# On Benchmarking of Deep Learning Systems: Software Engineering Issues and Reproducibility Challenges

**Inauguraldissertation**

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Antonio Maffia

aus Italien

Basel, 2023

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. em. Dr. Helmar Burkhart, Fakultätsverantwortlicher
Prof. Dr. Florina M. Ciorba, Fakultätsverantwortliche
Prof. Dr. Michael M. Resch, Korreferent

Basel, den 13.12.2022

Prof. Dr. Marcel Mayor, Dekan

*To Rossana and Edoardo. You have filled my life, giving it a greater meaning.*

# Abstract

Since AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, Deep Learning (and Machine Learning/AI in general) gained an exponential interest. Nowadays, their adoption spreads over numerous sectors, like automotive, robotics, healthcare and finance.

The ML advancement goes in pair with the quality improvement delivered by those solutions. However, those ameliorations are not for free: ML algorithms always require an increasing computational power, which pushes computer engineers to develop new devices capable of coping with this demand for performance. To foster the evolution of DSAs, and thus ML research, it is key to make it easy to experiment and compare them. This may be challenging since, even if the software built around these devices simplifies their usage, obtaining the best performance is not always straightforward. The situation gets even worse when the experiments are not conducted in a reproducible way. Even though the importance of reproducibility for the research is evident, it does not directly translate into reproducible experiments. In fact, as already shown by previous studies regarding other research fields, also ML is facing a reproducibility crisis.

Our work addresses the topic of reproducibility of ML applications. Reproducibility in this context has two aspects: results reproducibility and performance reproducibility. While the reproducibility of the results is mandatory, performance reproducibility cannot be neglected because high-performance device usage causes cost. To understand how the ML situation is regarding reproducibility of performance, we reproduce results published for the MLPerf suite, which seems to be the most used machine learning benchmark.

Because of the wide range of devices and frameworks used in different benchmark submissions, we focus on a subset of accuracy and performance results submitted to the MLPerf Inference benchmark, presenting a detailed

analysis of the difficulties a scientist may find when trying to reproduce such a benchmark and a possible solution using our workflow tool for experiment reproducibility: PROVA!. We designed PROVA! to support the reproducibility in traditional HPC experiments, but we will show how we extended it to be used as a "driver" for MLPerf benchmark applications. The PROVA! driver mode allows us to experiment with different versions of the MLPerf Inference benchmark switching among different hardware and software combinations and compare them in a reproducible way.

In the last part, we will present the results of our reproducibility study, demonstrating the importance of having a support tool to reproduce and extend original experiments getting deeper knowledge about performance behaviours.

# Acknowledgments

All these years spent working on my PhD have not always been easy, and if I reached this point, the first thanks I owe to Prof. Burkhart. From the moment he reassured me about starting this experience when we first met to the guidance he gave me over the years regarding research and life advices. He always supported me and continues to do so even in this last tough period. I didn't learn much German tough, but I wanted to say: Danke vielmals!

Thanks are also due to Prof. Dr. Michael M. Resch, who was always kind and helpful. I'm glad he invited me to Stuttgart to present my work to his research group and willingly accepted the request to be the co-referee for my PhD.

I want to thank the members of the former HPWC research group. Alexander, who helped me settle in Basel. I don't know how well this worked, but I certainly appreciated the attempt (especially the beers on Friday afternoon). Moreover, he honoured me with a great speech (in Italian) at my wedding, which is a moment I won't forget. Dominic, who helped me with the skeleton and the first version of PROVA! web UI, initiating me at Node.js :). Robert, with his fundamental help in managing and troubleshooting our cluster (a big part of my experience with HPC clusters comes from him). Bas supported me in the university and outside (I'm glad the football evenings keep going even after the PhD). And Yvonne, with her aid in both university and general administrative matters.

I had the opportunity to meet great people also in the new HPC group. Prof. Dr. Florina Ciorba has always been there when I needed help, making me feel an integral part of the group. In this last part of my PhD she did even more becoming my second supervisor to support me and prof. Burkhart in an unforeseen situation. Thank you so much!

Thanks also to the other guys in the HPC group: Ali and Ahmed, with whom I had the pleasure of many interesting conversations, as well as with Aurelian (with also a shared passion for Formula 1) and Jonas, an authentic Brazilian (especially for the way he cooks).

Outside of the University environment, I would like to thank Valentina and Francesco: they were always kind, took an interest in the progress of the thesis, and made themselves available on every occasion.

And, almost at the end, I thank the one who has been with me since day one. I moved to Basel because of him, and I have to say this was a perfect call. He helped and advised me countless times; I feel lucky to have such a friend. Grazie Danilo.

I thank my family who have always been close to me. The best thing that ever happened to me was having Rossana next to me. She has shared with me the weight of this doctorate with all its difficulties; I know it wasn't easy for her. She constantly pushed me and gave me the strength to reach my objective. This milestone is as much mine as it is hers. However, she was assisted by my little Edo: he is the only one who can get a smile out of me in every situation, even the worst. An infinite thanks to you two!

Finally, one last thought goes to Martin. I could write pages about all the times he supported me, was kind and helpful, and all the good times we had together, as well as how deeply sad it makes me that he is gone, but I'll just write one simple sentence: he was a really good friend.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| Abbreviation | Description | Definition |
|---|---|---|
| AGI | Artificial General Intelligence | page 19 |
| AI | Artificial Intelligence | page 19 |
| ANI | Artificial Narrow Intelligence | page 19 |
| ANN | Artificial Neural Network | page 21 |
| ASI | Artificial Super Intelligence | page 19 |
| ASIC | Application-Specific Integrated Circuit | page 13 |
| BF16 | Brain Floating Point 16 bit | page 24 |
| CD | Continuous Delivey/Deployment | page 140 |
| CEQIP | Cryogenic Electronics and Quantum Information Processing | page 11 |
| CI | Continuous Integration | page 45 |
| CNN | Convolutional Neural Network | page 22 |
| CUDA | Compute Unified Device Architecture | page 14 |
| DL | Deep Learning | page 19 |
| DNN | Deep Neural Network | page 21 |
| DSA | Domain-Specific Architecture | page 12 |
| DSL | Domain-Specific Language | page 14 |
| DSP | Digital Signal Processor | page 26 |
| FPGA | Field-Programmable Gate Arrays | page 13 |
| GAAFET | Gate-All-Around FET | page 10 |
| GPGPU | General-Purpose Graphics Processig Unit | page 13 |
| GPU | Graphics Processig Unit | page 13 |
| HDL | Hardware Description Language | page 13 |
| IC | Integrated Cirtuit | page 9 |
| ICH | Image Classification Heavyweight | page 112 |
| ICL | Image Classification Lightweight | page 112 |
| iGPU | Intel Integrated Graphics Processig Unit | page 26 |

| Abbreviation | Description | Definition |
|---|---|---|
| IPU | Intelligence Processing Unit | page 25 |
| IRDS | International Roadmap for Device and Systems | page 10 |
| ITRS | International Technology Roadmap of Semiconductors | page 10 |
| LoadGen | Load Generator | page 38 |
| ML | Machine Learning | page 19 |
| OCI | Open Container Initiative | page 51 |
| ODH | Object Detection Heavyweight | page 112 |
| ODL | Object Detection Lightweight | page 112 |
| ORD | Open Research Data | page 148 |
| SUT | System Under Testing | page 38 |
| TDP | Thermal Design Power | page 9 |
| TF | TensorFlow | page 26 |
| TF32 | TensorFloat-32 | page 24 |
| TPU | Tensor Processing Unit | page 13 |
| TTA | Time-To-Accuracy | page 36 |
| UDSS | User-Defined Software Stack | page 52 |
| VPU | Vision Processing Unit | page 24 |
| WfMS | Workflow Management Systems | page 55 |
| WSE | Wafer Scale Engine | page 25 |

# Chapter 1

# Introduction

Over the last decade, the interest in machine learning and AI had a significant rise. Thanks to the improvement reached by the quality of its solutions, nowadays, ML/AI is used to solve problems in many different fields. Their adoption goes from transportation to finance to manufacturing to healthcare and more.

The increase in ML algorithms accuracy is linked to the growth in complexity and, thus, the computational power required. Since AlexNet won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, marking a breakthrough in the field of ML, the number of petaflop/s-days needed by AI applications to train changed its trend, passing from a doubling every two years (similar to Moore's law) to a doubling every 3/4 months [1]. To cope with this increase in computational power, we are witnessing a proliferation of hardware and software in the form of domain-specific architecture (DSA) devices and frameworks trying to provide the necessary performance while traditional hardware struggles as Moore's law slows down. However, those devices are usually built upon novel architectures, making them tricky to operate and even more problematic when it comes to squeezing the best performance out of them.

Making it easy to experiment with those domain-specific devices and frameworks is fundamental to fostering their evolution and, therefore, advancements of ML research, but it can become extremely challenging when the experiments are not conducted in a reproducible way. In fact, as for other research fields, ML is also facing a reproducibility crisis, mainly due to the lack of proper documentation and explanation of experiment con-

figuration and execution. The efforts for helping reproducibility in ML are focusing on experiment outputs but not considering the differences in performance obtainable by using different hardware and software. Reproducibility in this context has two aspects: result reproducibility and performance reproducibility. While reproducibility of the results is mandatory, performance reproducibility cannot be neglected because high-performance devices usage causes cost.

Another critical factor for advancements in ML is to have a benchmark that can push for performance improvements, just as the Linpack benchmark led to the evolution of HPC systems while drawing up the TOP500 ranking. The designing of a good benchmark for ML is not an easy task. On the one hand, it involves the selection of applications that are relevant for such a rapidly evolving field and, on the other, the definition of metrics that can reward the performance that does not go to the detriment of quality. There has been much effort to define an ML benchmark, but MLPerf seems to be the one that became state of the art, promoting experiment reproducibility and fair comparison.

Furthermore, the execution of the experiment can also become quite complex, especially when it involves several steps in addition to its configuration and the communication with the execution system, which, in the case of HPCs, is most likely to be only remotely accessible. In this case, it is helpful to use tools like workflow management systems supporting the whole execution flow [2]. To address those reproducibility issues, we start defining a **taxonomy** of the experiment, which we use as the base to build our reproducibility **level** definitions (as theoretical support), and our experiment management tool: PROVA! (as practical support).

PROVA! was born to manage traditional high-performance computing experiments [3]. Still, we could extend it to support a different research field, i.e., Machine Learning, thanks to its simple and flexible design. This new PROVA! version allows us to study the performance reproducibility in ML by running the MLPerf benchmarks.

However, the range of devices and frameworks used in different MLPerf benchmark submissions is massive, and fully reproducing all of them would be dispersive. Instead, focusing on one benchmark, i.e., the MLPerf Inference, and only on a subset of the submissions, we could have a deeper analysis of the performance behaviour and show the many experiment customization possibilities given by using PROVA!.

Nevertheless, predefined experiments like a benchmark can be repetitive in terms of configuration and parameters to set. In this situation, it

is helpful to have a template that allows users to have a base working experiment that can be customized and used as a starting point for possible experiment variations and, once created, can be shared with others having similar needs.

In PROVA!, we created a specific "driver" (template) for the MLPerf Inference benchmark, which helped us with the reproducibility difficulties we faced during our performance reproducibility study. Finally, PROVA! assisted us to run the experiments using different hardware and software combinations and across multiple versions of the benchmark in a reproducible way, allowing us not only to reproduce the original experiments but also to expand the results and get more insight.

## 1.1 Research Questions

- The importance of the experiment's reproducibility is undebatable. Still, we face a reproducibility crisis in different research fields, including machine learning. In this field, most efforts aim at the results' reproducibility without paying much attention to the reproducibility of the performance. How can we support performance reproducibility in a fastly evolving field like machine learning?

- Accelerators overperform legacy CPU systems in specific domains. Their performance highly depends on optimized configuration and programming, which are not always obvious. How can we simplify accelerator experimentation and lower the learning curve by promoting greater collaboration between experts and machine learning scientists?

- Several machine learning benchmarks strive to compare hardware and software performance in the best way with the ultimate goal of pushing their improvements. How can we help manage predefined applications like benchmarks in a reproducible way so that scientists can use them as a base for creating custom experiments generating new insights?

## 1.2 Outline

The thesis is structured as follows. The first part discusses the motivations that led the HPC field towards domain-specific architectures and specialized frameworks for supporting machine learning applications and the difficulties encountered while carrying on experiments in a reproducible way. Our

three-level taxonomy of performance reproducibility defined in 2014 will be used in the experiments reported in Part IV. The second part debates the importance of benchmarks for machine learning hardware and software evolution with a survey on the main existing ML benchmarks and a more detailed introduction to the MLPerf benchmark suite, complemented by a user journey when ML Inference benchmark results are inspected. To address the challenges discussed in Part I and experiment with the benchmark applications presented in Part II, in the third part, we introduce our experiment management tool, PROVA!. We will provide a comprehensive description of it, including the architectural aspects and the approach used to extend it to support MLPerf benchmark experiments. Part 4 is the central part of the thesis. A well-defined sequence of experiment variants compares the performance values gathered to the original benchmark data. Finally, in Part 5, we summarize the outcomes of our study and possible future research steps.

## 1.3 Publications

The following list represents publications I contributed which are covering part of the contents of this doctoral dissertation

[4] A. Maffia. Reproducing ML Benchmarks: What Works What Doesn't Work. In review on the *32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, –, 2023.

[3] D. Guerrera, A. Maffia, and H. Burkhart. Reproducible Stencil Compiler Benchmarks Using PROVA!. Journal of *Future Generation Computer Systems (FGCS)*, 92:933–946, 2019.

[5] D. Guerrera, H. Burkhart, and A. Maffia. Reproducible Stencil Compiler Benchmarks Using PROVA!. In Proceedings of the *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 108–115, 2016.

[6] H. Burkhart, D. Guerrera, and A. Maffia. No more Believe Me : Make Your Informatics Experiments Reproducible. Presented as a poster in the *11th European Computer Science Summit (ECSS)*, 2015.

[2] A. Maffia, H. Burkhart, and D. Guerrera. Reproducibility in Practice: Lessons Learned from Research and Teaching Experiments. In Pro-

ceedings of the *Euro-Par 2015: Parallel Processing Workshops*, pages 592–603, 2015.

[7] H. Burkhart, D. Guerrera, and A. Maffia. Trusted High-Performance Computing in the Classroom. In Proceedings of the *Workshop on Education for High-Performance Computing (EduHPC)*, pages 27–33. 2014.

[8] D. Guerrera, H. Burkhart, and A. Maffia. Reproducible Experiments in Parallel Computing: Concepts and Stencil Compiler Benchmark Study. In Proceeding of the *Euro-Par 2014: Parallel Processing Workshops*, pages 464–474, 2014.

# Part I

# High-Performance Computing Meets Deep Learning

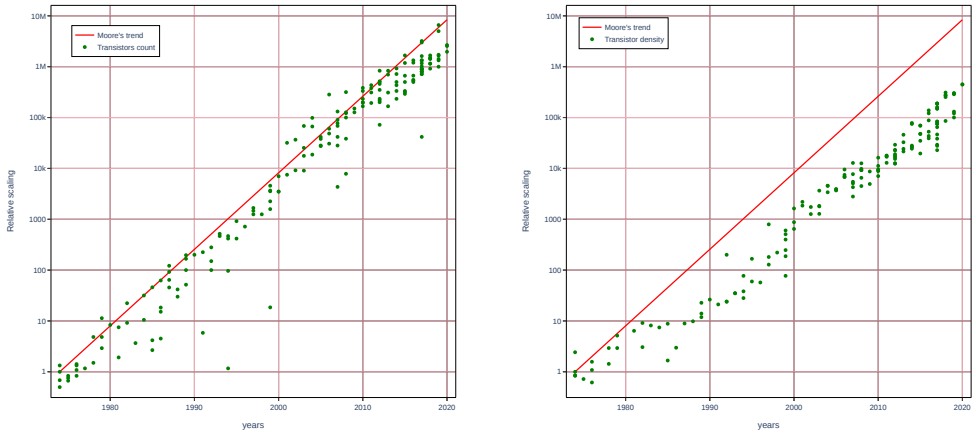
# Chapter 2

# Beyond Moore's Law

## 2.1 Semiconductors' trends

Moore's law drove performance improvements in the High-Performance Computing (HPC) field for more than 50 years. Even though the end of Moore's law has been wrongly announced many times, the gap between the predicted and the actual chip performance is rapidly growing, especially in the last decade, showing a concrete slow down [9]. The power of this law is given not only from the possibility of providing a performance prediction for semiconductor chips but also for their manufacturer to plan production costs. For this reason, there has always been a significant effort to try to keep it alive.

Nowadays, the total number of transistors in ICs is still growing at a pace of 2x every two years, as shown in Figure 2.1a. Still, the transistor density graph tells a different story: the semiconductor industry could not keep transistor density improving at the same speed, and the situation got even worse after the end of Dennard's scaling. Figure 2.1b shows a prominent slowdown after the mid 2000s leading to a difference of almost 19 times between actual and Moore's trend transistor density. The main problem is still related to containing power dissipation while increasing the transistor density, and this brings ICs into the "dark silicon" era where processors must reduce their clock frequency or even turn off components to keep the device working below its thermal design power (TDP) [10]. It follows that manufacturers can increase the number of components, but this does not necessarily correspond to the same increase in performance.




**(a)** *Transistors count evolution vs. Moore's law since 1974.*

**(b)** *Transistors density evolution vs. Moore's law since 1974.*

**Figure 2.1:** *Transistor evolution vs. Moore's law. The trend for Moore's law has been based on the Intel 8080 (introduced in 1974), and the density derived from the number of transistors and chip area. Data source* [11].

A further signal that Moore's law is approaching its end is the closing in 2016 of the International Technology Roadmap of Semiconductors (ITRS). Since the 1990s, ITRS has provided biannual reports about the semiconductor technology situation and future views that helped keep alignment between the semiconductor industry and Moore's law. However, ITRS mainly concentrated on silicon technology [12]. For this reason, thanks to the IEEE Rebooting Computing initiative, the International Roadmap for Device and Systems (IRDS) was created to focus on different aspects of computer technology evolution like application benchmarking, systems and architectures, regarding not only further developments of Moore's law but also CMOS alternatives.

The most promising directions for the "More Moore" IRDS category include using different materials and evolving finFET transistor design. For example, using materials from group III-V of the periodic table to replace or combine with silicon can improve electron mobility allowing a faster gate switch and a reduced operating voltage, thus reducing generated power and heat [13]. On the transistor design side, the next step is to move from finFET to GAAFET (gate-all-around FET), like nanosheet technology, in which the gate wraps around the channel, further improving its control compared to what happens with finFET. To move those approaches forward,

the semiconductor industry needs an improvement its manufacturing process; in a recent conference, Intel presented the road to a 50-fold transistor density improvement using a 3D design with stacked GAAFETs [14] and even IRDS, in its latest report [15], forecasted a transistor technology with a feature size equivalent to 0.7nm (using the 3D design) by 2034.

GAAFET will most likely not only be the next technology chosen to advance Moore's law but also the last one [13]. What is instead less clear is if shrinking the transistor until the size forecasted by Intel and IRDS will eventually happen. Even though a path to keep shrinking transistors seems to exist, reducing feature sizes is generating more and more an explosion of design and manufacturing costs in the face of a not so significant performance improvement [16]. The economic aspect is key for Moore's law survival: in his paper [17], Moore emphasized on being able to increase the number of components in an IC, reducing the cost per transistor, and this seems to be not the case anymore [18].

Alternatives presented by IRDS in the "Beyond CMOS" category are going from Analog Computing which is meant to exploit physical phenomena (like in the case of neuro-inspired computing) to Probabilistic/Stochastic circuits, which implement "real" nondeterminism and randomness in hardware, helping with problems like Monte Carlo simulation and Simulated Annealing to Quantum Computing. Quantum Computing uses quantum mechanics phenomena like superposition, entanglement, and tunnelling to solve specific computational problems in cryptography, quantum optimization, machine learning, search, and quantum chemistry.

Using this kind of machine is not a recent idea [19], but, only in the last decade, the approach started to make good progress. Several manufacturers are building a quantum computer with an increasing number of quantum bits (qubit) with the final aim of demonstrating the advantage against a classical computer, the so-called quantum supremacy [20]. Quantum computer vendors and researchers are now in this race; whether we have already reached quantum supremacy [21] or not [22] is something still under discussion, but there is no doubt this is a real promising path to follow, even the IRDS since 2018 dedicates a separate category to Cryogenic Electronics and Quantum Information Processing (CEQIP).

Though different candidates are vying to be the next leading technology, there is no clear winner yet; moreover, a solution demonstrated in the lab today would probably need at least a decade to meet industry standards and start to be used for mass production as it happened in the case of the FinFET[23]. The IRDS alternatives are pretty different, but they seem to have

a common denominator: being engineered with a specific problem or class of problems in mind. The Domain-Specific Architecture (DSA) approach looks the most promising one, not only if applied to novel technologies for the future "post-silicon" era but also to drive the design of CMOS-based architecture, which can already bear substantial performance advantages.

## 2.2 DSAs and HW accelerators

The idea of using supplementary hardware to offload specific operations not suited for the "main" processor (or too expensive for it) is not really new. Already in the 1960s, General Electric was selling an "Auxiliary Arithmetic Unit" (AAU) to compute operations with a higher arithmetic precision (floating point). This was considered an extension of the basic arithmetic unit present in the central processor and was provided as a "two-cabinet" size component (AAU and its controller)[24]. To execute FP operations, the central processor was granting the AAU access to the instructions stored in the main memory, and the AAU was accessing those using an I/O channel controller and subsequently decoding/executing them without further central processor intervention, similar to the way a modern CoProcessor uses the DMA. Floating-point units continued to evolve from AAU to smaller components, first as "Arithmetic Processing Unit" (APU) with the Intel 8231/AMD Am9511 and "Floating-point Processor Unit" (FPU) with the Intel 8232/AMD Am9512, and later on as "Math CoProcessor" with the Intel 8087, before getting eventually integrated into the CPU chip starting with the Intel 80486 processor.

In the beginning, keeping the CPU executing only the most common operations also had a "complexity" motivation. In fact, the number of transistors of a CoProcessor (ex. 45k for the Intel 8087) could be higher than the ones of the CPU (ex. 29k for the Intel 8086). Later on, with the evolution of the semiconductor industry and thanks to Dennard's scaling, integrating specific hardware in the same IC became possible without significant effort and, therefore, an obvious choice[25].

Clearly, CPU architectures improved during the years not only due to Dennard's scaling and clock frequency increment but also due to the introduction of hardware and software optimization techniques like instruction pipelining, speculative execution, and very long instruction word (VLIW), cache hierarchies, and more. Even the idea of integrating components to exploit data-parallelism, one of the most common approaches in the mod-

ern DSAs to boost performance, has been successfully applied to the world of High-Performance computing (HPC) since the 1970s by Cray with their Cray-1 (1976)[26], which, thanks to the help of an integrated “vector processor", become the fastest supercomputer of its time.

Nowadays, the power consumption and the costs for designing improved traditional general-purpose ICs are not sustainable anymore. In addition, the renewed idea of separating hardware capabilities in different specialized devices can help deliver performance in a more energy-efficient way. Those ICs are, most of the time, devices working in combination with the main CPU of a computer only when there is the need to accelerate a specific problem they are designed for. That is why they can also be called accelerators.

An accelerator does not necessarily need to be a single-purpose device (even though that is true most of the time). Its main goal is to target the performance improvement of a specific aspect of a problem, which may be helpful in multiple fields, that is, for example, the idea behind the graphics processing units (GPUs). Born to address computer graphics workloads related to the manipulation of images and videos and their visualization on display devices, GPUs started moving towards the era of general-purpose computing on graphics processing units (GPGPUs) when, in the early 2000s, NVIDIA (currently the leading GPU vendor) added some “programmable” components to its devices[27]. In fact, GPGPUs are currently used for a vast range of applications like Artificial Intelligence, finance, climate, molecular dynamics, and more, with a clear focus on extensively using data-parallelism through SIMD/SIMT(single instruction, multiple data/threads) execution model.

Other accelerators can be more tailored to a specific application, as in the case of the ASICs (Application-Specific Integrated Circuits), which, focusing just on a tight aspect of a problem, can deliver a tremendous performance per watt like the tensor processing units (TPUs) designed by Google. TPU v1[28], in fact, implements the systolic array execution model to improve only matrix-matrix multiplication and speed up machine learning inference. An accelerators’ architecture approach, somehow between GPGPU and ASIC, is the field-programmable gate array (FPGA), which can be programmed at the hardware level, usually through a hardware description language (HDL), and provide both flexibility and a specialized/optimized IC. If, on the one hand, FPGAs can also reduce the time and cost of the chip design, on the other, they cannot provide a complete optimization since the hardware provided by the device can be “just” configured but not

| | GPGPU | FPGA | ASIC |
|---|---|---|---|
| **Description** | General-purpose computing on graphics processing units | Field-programmable gate arrays | Application-specific integrated circuit |
| **Focus** | General-purpose, Data-parallelism (SIMT) | General/Single-purpose | Single-purpose |
| **Strenghts** | ++ Programmability<br>++ Flexibility<br>+ Memory Bandwidth<br>+ Peak performance | + Performance/Watt<br>+ Flexibility<br>+ I/O Bandwidth<br>- I/O Latency<br>- Clock Frequency | ++ Peak performance<br>++ Energy efficient |
| **Weaknesses** | + Power consumption | - Clock Frequency<br>- Peak performance<br>- Programmability | + Design costs<br>+ Design complexity<br>- Flexibility<br>- Programmability |

**Table 2.1:** *Overview of main accelerator categories.*

designed ad-hoc as it happens with an ASIC. Another aspect of FPGAs is that they usually operate at a low clock frequency and, thus, have no high peak performance. This can be a weakness if targeting ML training but can be a strength when used for ML inference since this case does not request high computation, and FPGA, using less power, provides a higher performance/watt. Table 2.1 reports an overview of the main accelerator categories.

The possible need for implementing the applications using an approach different from the one used for the traditional CPUs could be a drawback of using specific architectures: reaching the optimal performance of an accelerator may require deep and specific expertise. For this reason, the success of an accelerator may depend on the software ecosystem built around it, and this was for sure one of the reasons behind NVIDIA GPU's success. Their Compute Unified Device Architecture (CUDA) language allowed them to hide the GPU's execution model complexity and integrate it with well-established languages like C and Fortran.

For example, in [29], they show how, starting from an algorithm written in a high-level language like Python, it is possible to easily get up to 62k-fold improvement using a lower-level language and applying architecture-specific optimization. A domain-specific language (DSL) simplifies how a developer can program the device and provides abstract functions and operations implemented in the most optimized way to get the best performance from the hardware. Hennessy and Patterson [9] consider DSL one of the four reasons motivating the use of DSAs; the further three are, instead, related to the specialization of the components. Optimization techniques like out-of-order speculative execution, implemented to improve performance on the

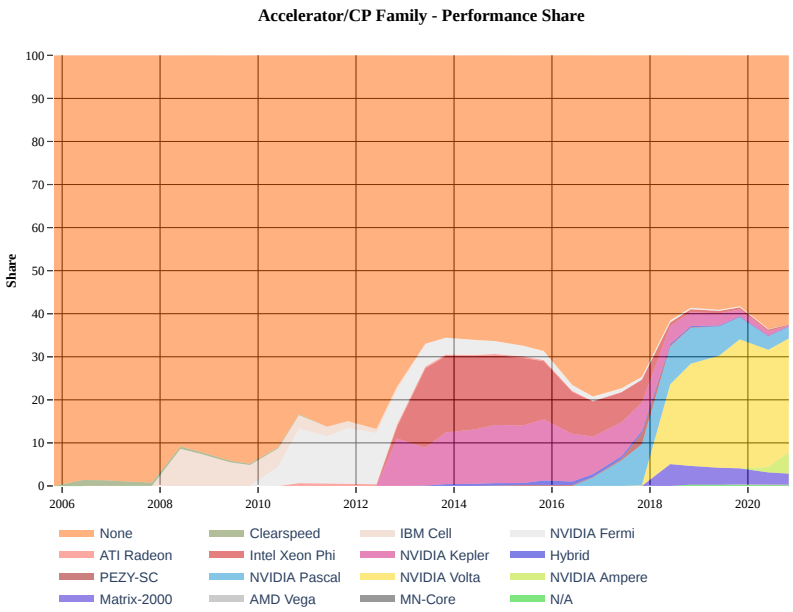



**Figure 2.2:** *Evolution of accelerators' performance contribution to TOP500 machines since their first adoption in June 2006. Data source [30].*

traditional CPUs, may cause a significant overhead when unnecessary instructions get executed and need to be reverted. The specific nature of the DSA architecture narrows down the possible instructions making the execution flow more predictable and allowing, for example, to use of a different instruction set architecture (ISA) like VLIW to exploit ①more effective parallelism. A similar discussion is valid for the cache management effort, which can be reduced by favouring a ②memory bandwidth increment (lowering the costly memory access). Finally, also the ③data representation may require a reduced amount of bits (lower precision) like 4-,8- and 16-bit integers for machine learning inference and 16- to 32-bit floating-point for the training, so 64-bit double-precision FPs (and in most cases even 32-bit single-precision FPs) are not needed.

An example of how accelerators can be effective for the growth of computers and HPC's performance evolution can be observed by looking at the TOP500 list. This list, updated twice a year, represents the existing HPC systems ranked by their performance on the Linpack benchmark. Since the accelerator's inception in 2006, the HPC systems' performance depends more and more on those devices. Figure 2.2 shows the evolution of the total

performance given by all systems in the list based on the accelerators used: todays' performance of the list is given by accelerators for almost 40% of the total, and, looking at the trend, this value is going to keep increasing in the next future. Different accelerators were added to HPC systems during the years, but it seems Nvidia GPUs are dominant today. Figure 2.2 also shows that the only other accelerator able to compete with GPGPUs, so far, was Intel Xeon Phi [31]. This specialized architecture focuses on exploiting data parallelism (like GPUs) but keeping compatibility with CPU architecture and just extending the x86 ISA with some specific instructions. In principle, the code did not need any change to run on this new device (because of the instruction compatibility), but it did need some changes to get a performance boost. This, together with the Intel difficulties with the 10 nm process (planned for the Xeon Phi evolution), made the project move towards a more specialized architecture to address machine learning [32] and eventually discontinued in favour of a more diversified DSA-oriented strategy. In fact, Intel lately acquired AI chip companies [33] and developed a GPGPU architecture(Intel Xe PonteVecchio [14]), adding those to their existing FPGA products.

Other fields are requesting high computational power that not even HPC systems can provide or, at least, not in an efficient way. A great example comes from molecular dynamics, where an ASIC device like Anton can deliver impressive results. In 2014, the enhanced version of Anton, Anton 2[34], provided a speedup that, for different molecules, could go from a minimum of 21x up to 800x compared to different GPUs and HPC systems using an even lower power consumption.

A similar situation is that observed in ML. Although machine learning algorithms such as neural networks already existed at the end of the 1950s, until the adoption of DSAs for their resolution, the results generated were not sufficiently accurate and, for example, applications such as image recognition had a percentage of error of five times worse than that of a human [25]. A specific architecture provided the necessary performance to add more layers to neural networks, called deep neural networks (DNNs), to generate much more accurate results, making these algorithms finally useful. The ability to cope with the computation required by deep learning (DL) has led to the evolution of algorithms that are becoming increasingly demanding from a computational point of view (Figure 2.3), which in turn has seen a proliferation of new ML- and DL-specific architectures. There is a clear need for tools to manage device complexity and specific benchmarks to compare these different architectures.

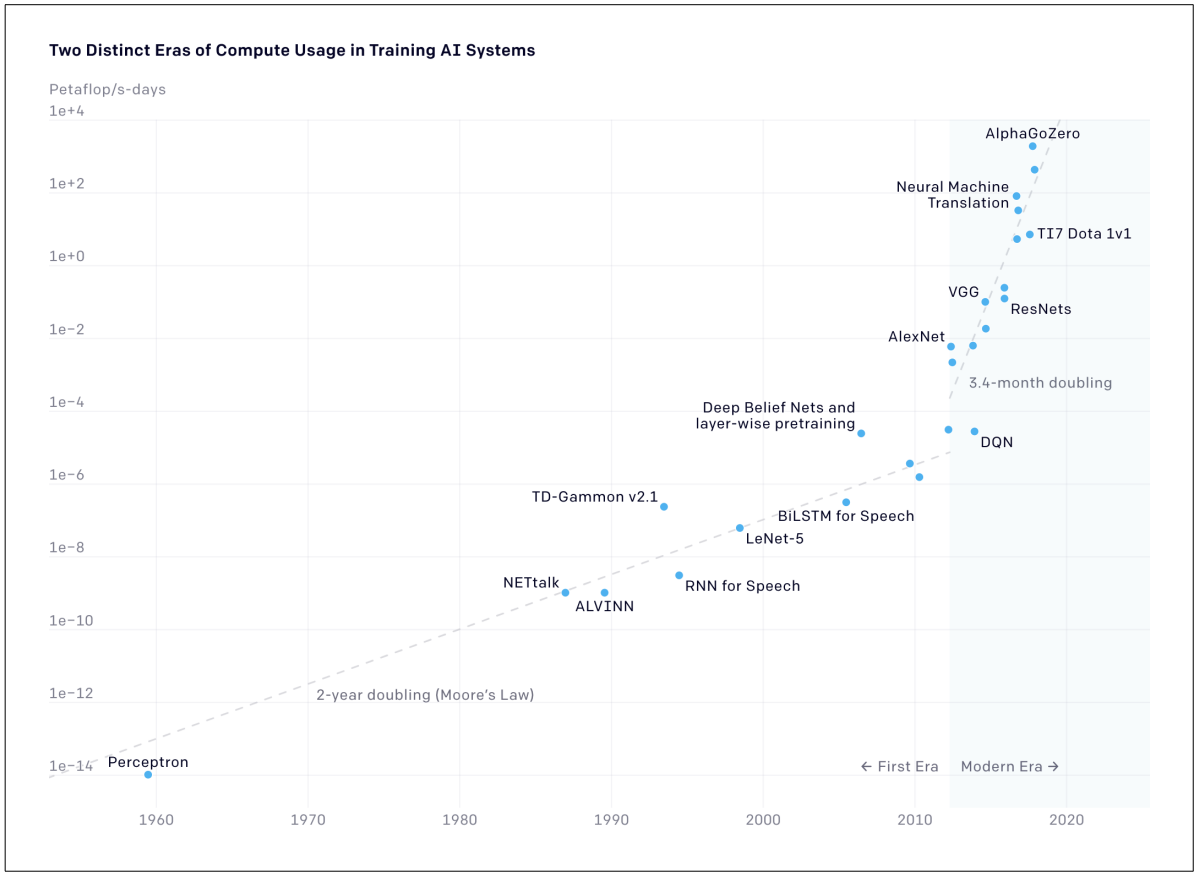



**Figure 2.3:** *The total amount of compute, in petaflop/s-days, used to train selected AI applications. A petaflop/s-day (pfs-day) consists of performing* $10^{15}$ *neural net operations per second for one day, or a total of about* $10^{20}$ *operations. Image courtesy of [1].*

# Chapter 3

# Deep Learning and the Need for HPC

## 3.1 AI, machine learning, deep learning: Definitions

The terminology around artificial intelligence is often a source of confusion. Deep Learning (DL), Machine Learning (ML) and Artificial Intelligence (AI) themselves represent "nested" sets (figure 3.1). So, on the one hand, it is true that DL is also ML, and ML is also AI, on the other, it is not always true the opposite.

AI is the largest set and includes all the "science and engineering of making intelligent machines, especially intelligent computer programs" [35]. The current AI applications can only solve specific target tasks and, thus, are categorized as Artificial Narrow Intelligence (ANI). Instead, we talk about Artificial General Intelligence (AGI) and Artificial Super Intelligence (ASI), referring to AI solutions that can reach or even surpass the human intelligence level. Even though research and projects about AGI exist [36], there are currently no concrete examples of AGI or ASI.

Within AI, ML represents algorithms that are not directly programmed to solve a specific problem but can instead "learn" how to do it: they extract knowledge from the input data to predict outcomes. The final behaviour results from the gradual improvements achieved after each "learning" process step, similarly to what happens for a living being.

ML consists of three main categories:

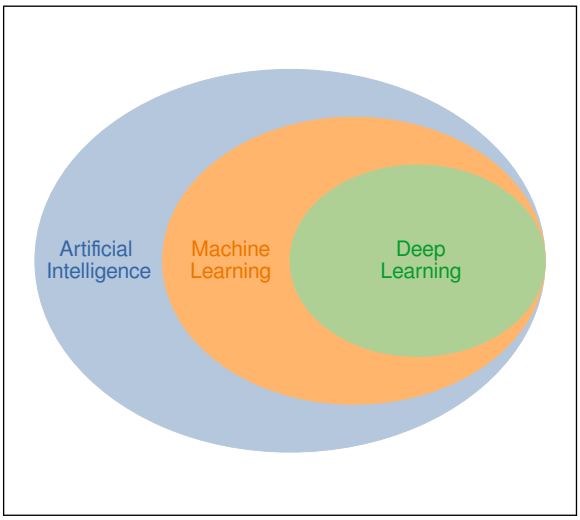



**Figure 3.1:** *Venn diagram for Artificial Intelligence, Machine Learning and Deep Learning concepts*

- **Supervised Learning**: ML algorithms trained on *labeled data*. The labels are information added to the data that define the input values corresponding to the desired output to be used during the training to refine the model. Those algorithms can classify input data into predefined categories (classification) or find equations that better approximate the relation between the provided input and output data (regression).
- **Unsupervised Learning**: ML algorithms trained to identify patterns on *unlabeled data*. Those algorithms can group input data based on common characteristics (clustering) or discover relationships between variables in a dataset (association), or also reduce the number of features in a dataset without losing helpful information (dimensionality reduction). The last example is mainly used as data preprocessing.
- **Reinforcement Learning**: ML algorithms trained on *reward* got for correct behaviour. The "agent" to be trained will take some actions in a specific "environment" and, based on the correctness of the action taken, he may get a reward. After several iterations, the agent should be able to perform the task he was trained for correctly. Examples of this category are the algorithms used in automatic game playing (e.g. AlphaGO [37]) or in self-driving cars.

For instance, in the case of the image classification task, the so-called

**feature engineering** can improve the quality of the results: a domain expert "extracts" the features, i.e., a set of characteristics, which the algorithm should focus on during the training. Nonetheless, this process becomes impracticable when the number of features increases or when those features are not clear a priori. Using the artificial neural network (ANNs) will help in such situations. The ANNs, or just neural networks (NNs), are inspired by the human brain functioning and define **artificial neurons** (nodes) which can communicate with each other using connections called **edges**. The data in the NNs pass through different layers that represent the steps of the algorithm. The structure of the NNs, in figure 3.2, includes an input layer that acquires the data, an output layer that generates the results and a variable number of *hidden* layers. An NN that has more than one hidden layer is considered a DNN.

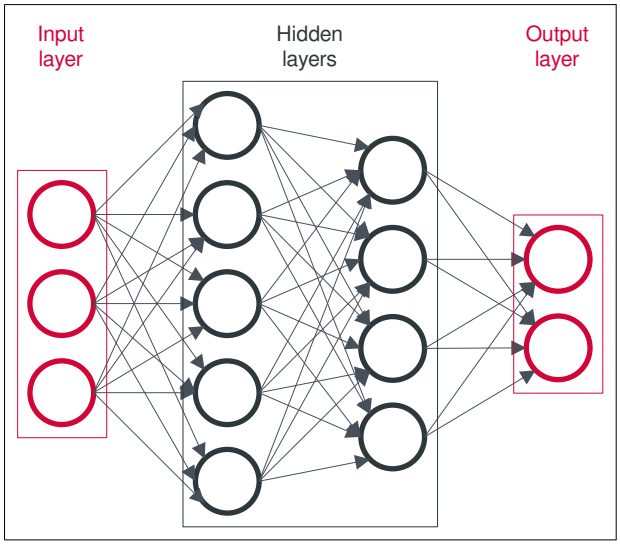



**Figure 3.2:** *Example of a Deep Neural Network (DNN)*

NNs do not require human intervention for the feature extraction step, as they will learn the feature only starting from the input data, which means they can use unstructured input data. Nevertheless, to provide a good result, they need to train on a much more significant amount of data than the classic ML, which translates into higher complexity and, thus, computational power required. The computation required gets even higher in the case of DL, where the larger number of layers can further increase the algorithm complexity.

Research involving NN originated in the 1950s [38] and got an increasing interest in the 1980s [39] [40] [41] but the learning process was still too slow

for the hardware available at that time. The situation changed in the last decade.

In 2012, AlexNet [42] was the first Deep Learning algorithm (CNN*) to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a considerable improvement compared to the previous solutions. This was achieved thanks to the algorithm's optimization and a hardware accelerator (Nvidia GPU), which provided a relevant speed up to the training. Since then, every competition winner has used Deep Learning algorithms and accelerators.

## 3.2 Deep learning applications' landscape

The increasing quality of results provided by machine learning algorithms, especially deep learning, pushed the adoption of those solutions in many diverse fields, from transportation to finance to agriculture to manufacturing to healthcare.

Even though the fields applying machine learning solutions are quite diverse, the same problems may be reoccurring in more than one field. For example, ML applications, like object detection from the Computer Vision area (table 3.1), can be used for social media and automotive. The actual difference usually resides in the requirements and constraints of a solution, which leads to the choice of one model instead of another: an object detection model recognizing the people we want to tag on a social media platform has, clearly, different asks in terms of accuracy or result latency compared to a model solving the same object detection tasks used by a self-driving vehicle to recognizing a pedestrian walking on the zebra crossing.

As discussed in [43] and [44], the different models trained to solve the same ML application, like image recognition, show a tradeoff between accuracy and complexity: a more accurate model will need more features and computational and memory requirements. Moreover, the possible datasets utilized to train a single model can vary and affect the results of the learning process. Sometimes, using different datasets could be a requirement of the application: in the aforementioned self-driving vehicle example, recognizing a road sign is be more important than spotting a fish breed, which may pull weight in other research areas.

*Convolutional Neural Network

| Area | Application | Dataset |
|---|---|---|
| Computer Vision | Image Classification, Object Detection, Semantic Segmentation, OCR, Facial recognition, Pattern Detection, Content-based Tmage Retrieval | ImageNet, CIFAR, MNIST, Cityscapes, ADE20K, PASCAL VOC, COCO, LFW, Adience, WIDER FACE, FDDB, INRIA Holidays Dataset |
| Automatic Speech Recognition | Audio Recognition, Audio Segmentation, Audio Generation, Computer Human Dialogue Systems, Text-to-Speech | LibriSpeech, TIMIT, MediaSpeech, SLUE, TUDA, PersonaChat, UDC, Reddit, LJSpeech |
| Natural Language Processing | Natural Language Generation, Text Summarization, Topic Analysus, Sentiment Analysis/Opinion Mining, Question Answering | WikiText, Penn Treebank, WMT, SQuAD, WikiQA, bAbI, SST, IMBd, MATH, DART |
| Recommendation Systems | Collaborative Filtering, Content-based Filtering, Context-aware Recommender Systems, Hybrid Recommender Systems | MovieLens, MIND, Criteo |
| Medical | Medical Image Segmentation, Drug Discovery, Medical Diagnosis | BraTS, CVC-ClinicDB, Kvasir, ACDC, Tox21, QM9, MIMIC-III |
| Robotics | Motion Planning, Robot Navigation, Human-Robot Interaction, Visual Odometry, Visual Navigation | Gibson 3D scenes, Matterport3D scenes, R2R |
| GameAI | Object Tracking, Continuous Control, Atari Games, OpenAI Gym | BIRDSAI, PyBullet, DMCS, Atari 2600, LunarLander |

**Table 3.1:** *A list of deep learning applications and datasets grouped by research area.*

However, selecting the best model in terms of accuracy is not always the obvious choice. We may have further restrictions coming from the device we want to use, like keeping a low power consumption. In this case, the most accurate model will probably be prohibitive in terms of complexity. That is why the hardware selected for the learning task has a crucial role.

## 3.3 HW and SW for DL

### 3.3.1 ML Architecture

As discussed in the chapter 2.2, machine learning applications push adoption and performance improvements in DSAs. Within the machine learning field, deep learning plays a key role, as shown by the many applications using it (see section 3.2). Deep learning algorithms are computational hungry

because of larger models, i.e., more features and layers, and datasets (see (section 3.1)) and that is confirmed by OpenAI, which showed an increase of 300,000x for computing need of algorithms developed between 2012 and 2018 (see figure 2.3)

Table 3.2 shows examples of DSAs for neural networks, and thus deep learning, from ASIC to FPGA to devices targeting specific application areas, i.e., Vision Processing Unit (VPU), or even more innovative architectures. They can focus on the learning process in general or only on specific phases, e.g., inference. Those devices will work with low data representation precision and include hardware components that can speed up the main operations present in the neural network in terms of computation, like matrix and vector multiplications.

For example, the Tensor Processing Unit (TPU), launched by Google in 2015, was designed for inference-only and working with int8 data format. It was built around a **systolic array** to reduces the "slow" read/write access to memory: a Unified Buffer (UB) feeds a Matrix Multiplication Unit (MXU) of 256x256 Multiply Accumulate (MAC) units performing 8-bit integer operations with a performance of 92 Teraops per second (running at 700MHz).

Working with 8-bit integers is only possible for inference after a process of quantization that approximates a model trained with a higher precision floating-point data format. Even the training process does not usually need high precision, but this process requires at least half-precision (FP16) to produce acceptable results.

To increase the device performance by reducing the data format precision without significantly loose model accuracy, device manufacturers are proposing additional custom data formats. With the TPU v2, Google introduced the Brain Floating Point (BF16), which is a "truncated" FP32 (IEEE 754 single-precision) with the same number range (8-bit exponent) and reduced the precision (mantissa reduced from 23 to 7-bit). The BF16 format showed to be effective for deep learning applications [45], and also CPU and GPU added the support to this data type in their latest architectures [46][47].

Nvidia also added the support for another custom floating-point format used by the *Tensor Cores* on their GPUs: the TensorFloat-32 (TF32). Despite the name, it is a 19-bits format that uses the same "truncation" approach of the BF16: the exponent is, again, represented using 8 bits, while the mantissa uses, instead, 10 bits. The format can represent the same precision as the FP16 (IEEE 754 half-precision standard) (see figure 3.3).

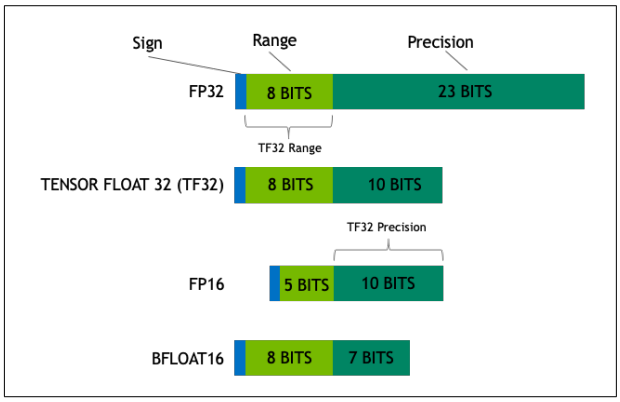



**Figure 3.3:** *Visual representation of TensorFloat-32 (TF32) data format. Original picture from* [48].

Besides, we have devices mainly focusing on accelerating the training phase using some different approaches like the Intelligence Processing Unit (IPU) [49] from Graphcore and the Wafer Scale Engine (WSE) [50] from Cerebras. Those devices have a massive amount of processing elements (PE) equipped with on-chip memory to guarantee a higher bandwidth and the possibility to be separately programmed following a MIMD approach.

All the devices mentioned have specific characteristics that may make them tricky to operate and even more problematic when trying to get the best performance out of them. For this reason, they must come with software frameworks optimized and easy to program.

### 3.3.2 ML Frameworks

With the complex landscape of specific hardware and the multitude of applications and models, having an easy way of developing deep learning algorithms was one of the main drivers for the evolutions in the field.

Those frameworks provide some **building blocks** from basic operations to functions, possibly optimized for specific hardware, to simplify the neural network implementation. At low-level, they usually leverage other optimized math libraries, for example, like MKL when running on CPU, cuDNN for GPU and even framework-specific optimization for execution on multiple hardware like the Accelerated Linear Algebra (XLA) from TensorFlow[†].

There are few commonalities among the frameworks, such as providing support to Python and C++ as programming languages or CPU and GPU architectures as hardware (see table 3.3). Specific hardware may provide

[†] https://www.tensorflow.org/xla

| Category | Name | Data format | Learning phase | Software / Framework | Supported Framework |
|---|---|---|---|---|---|
| ASIC | TPUv1, Edge TPU | INT8 | Inference | TFLite | ONNX |
| | TPUv2/3/4 | FP32, BF16, INT32 | Training, Inference | TensorFlow | PyTorch |
| | Habana Goya | FP32, INT32, INT16, INT8 | Inference | SynapseAI | TensorFlow, MXNet, Caffe2, CNTK, PyTorch, ONNX, Glow |
| | Habana Gaudi | FP32, BF16, INT32, INT16, INT8 | Training, Inference | SynapseAI | TensorFlow, PyTorch |
| | Hailo-8 | INT8 | Inference | Hailo AI | TensorFlow, ONNX |
| | Alibaba HanGuang | INT16, INT8 | Inference | HanGuangAI | Caffe, MxNet, TensorFlow, ONNX |
| DSP | Qualcomm Hexagon | INT8 | Inference | SNPE SDK | Caffe, Caffe2, ONNX and TensorFlow |
| FPGA | Xilinx Alveo | INT8 | Inference | Vitis AI | TensorFlow, PyTorch, Caffe |
| | FuriosaAI Warboy | INT8 | Inference | FuriosaAI SDK | - |
| VPU | Intel Movidius | FP16 | Inference | OpenVINO | TensorFlow, PyTorch, Caffe, ONNX, MXNet |
| IPU | Graphcore | FP32, FP16 | Training, Inference | Poplar SDK | TensorFlow, PyTorch, ONNX, Paddle Paddle |
| WSE | Cerebras | FP32, BF16 | Training, Inference | Cerebras SDK | TensorFlow, PyTorch |

**Table 3.2:** *Examples of machine learning hardware grouped by category.*

optimized libraries and SDK to help using them, like with Cerebras and Graphcore devices, but they also provide support/integration with the most used ML frameworks such as PyTorch and Tensorflow (see table 3.2).

Having this level of integration is essential to cope with device diversity. For example, if we want to run a model on an Intel integrated GPU (iGPU), we may need to use OpenVINO (see table 3.3), but thanks to the integration with TF, we can reuse a model trained on TF with a simple translation step.

Especially for inference, a concrete joint effort from the principal ML companies to have a unified way of expressing ML models is made with

ONNX [65]. This makes it possible to directly port models from one framework to another if the framework accepts the ONNX format as input or indirectly after a translation process.

Still, even if almost all devices are compatible with the significant ML frameworks, they can be supported differently by the framework in terms of optimization, making it important to have a performance comparison. Moreover, managing that software and its installation and configuration can be present challenges that we will address in the next chapter.

| Name (Affiliation) | Launch | Language | API | Supported HW | Note |
|---|---|---|---|---|---|
| TensorFlow (Google) | 2015 | Python, C++, CUDA | Python, C++, JavaScript, Java, more from the community [51] | CPU, GPU, TPU | TFLite variant used for Inference on Mobile devices |
| PyTorch (Facebook) | 2016 | Python, C++, CUDA | Python, C++, Java [52] | CPU, GPU, TPU | |
| ONNX RT (Microsoft) | 2018 | Python, C++, CUDA, C# | Python, C/C++, Java, C#, WinRT, Objective-C, JavaScript [53] | Variuos devices throught external backends [54] | Supports inference, initial support for training [55] |
| OpenVINO (Intel) | 2018 | Python, C++ | Python, C/C++ [56] | Intel devices: CPU, GPU, iGPU, VPU, FPGA | Provide a unified interface for inference on Intel devices |
| TensorRT (Nvidia) | 2016 | Python, C++, CUDA | Python, C++ [57] | GPU | Speedup inference on Nvidia GPU devices |
| Theano (University of Montreal) | 2007 | Python | Python [58] | CPU, GPU | Deprecated, forked into Aesara library [59] |
| Caffe (UC Berkeley) | 2013 | C++ | Python, MATLAB, C++ | CPU, GPU | Inactive (since Feb-2020) |
| Caffe2 (Facebook) | 2017 | C++ | Python, C++ [60] | CPU, GPU | Active as part of PyTorch |
| Chainer (Preferred Networks) | 2015 | Python | Python [61] | CPU, GPU | Deprecated, company moving to PyTorch |
| MXNet (Apache) | 2015 | C++, Python, CUDA | Python, Java, C++, R, Scala, Clojure, Go, Javascript, Perl, Julia [62] | CPU, GPU | |
| CNTK (Microsoft) | 2016 | C++ | Python, C++, C#, Java [63] | CPU, GPU | Deprecated, it contributes to ONNX/ONNX runtime |
| Mindspore AI (Huawei) | 2020 | C++, Python | Python, C++ [64] | CPU, GPU, Huawei NPU | Provide support to Huawei NPU devices |

**Table 3.3:** *List of main machine learning frameworks.*

# Part II

# Deep Learning HW/SW Analysis

# Chapter 4

# Reproducibility Challenges

The importance of reproducibility for the research is evident but not directly translating into reproducible experiments. As already shown by previous studies regarding other research fields [66], ML also faces a reproducibility crisis [67].

If, on the one hand, reproducibility in machine learning experiments can be challenging because of dealing with the randomness found in the data initialization or some atomic operation or the numerical approximation or even in the machine learning framework itself, on the other hand, good practices like defining and documenting the specific seed used for initialization can help reproducibility [68]. Besides the intrinsic randomness, other elements affecting reproducibility the most can be found in the lack of code (or pseudo-code) or poor explanations or detailed documentation of the experiment configuration and execution [69][70].

To improve results reproducibility, in the machine learning field, some scientific conferences started requesting fulfilment of submission policies and checklists when submitting a paper [71]. Along this line, the *Paper With Code*[*] initiative, currently supported by Meta AI (formerly Facebook AI), collects research papers in the machine learning field together with the code implemented for each paper and the results achieved to help understanding the state of the art of machine learning algorithms.

Those efforts mainly focus on the reproducibility of an ML experiment in terms of final results but do not consider the differences in performance obtainable by using different hardware and software.

[*] https://paperswithcode.com

## 4.1 Experiment taxonomy

To help address reproducibility challenges for a computational experiment, it is essential to precisely identify the fundamental characteristics of an experiment: this is the driver behind the experiment taxonomy we defined. Computational problem solving, in general, can be described as follows: A computational **problem** is solved by an algorithmic **method** on a compute **system**. As we presented in [8], a *micro-experiment* can be thought of as a 3-tuple <*Problem, Method, System*>, which can be considered as being one point in the space of experiments (see figure 4.1):

- ***Problem***: Solve a (random) dense system of linear equations in IEEE double-precision arithmetic.
- ***Method***: A two-dimensional block-cyclic data distribution using the right-looking variant of the LU factorization with row partial pivoting (see [72]).
- ***System***: Distributed-memory computer with Message Passing Interface (MPI 1.1 compliant) and Basic Linear Algebra Subprograms (BLAS) installed.

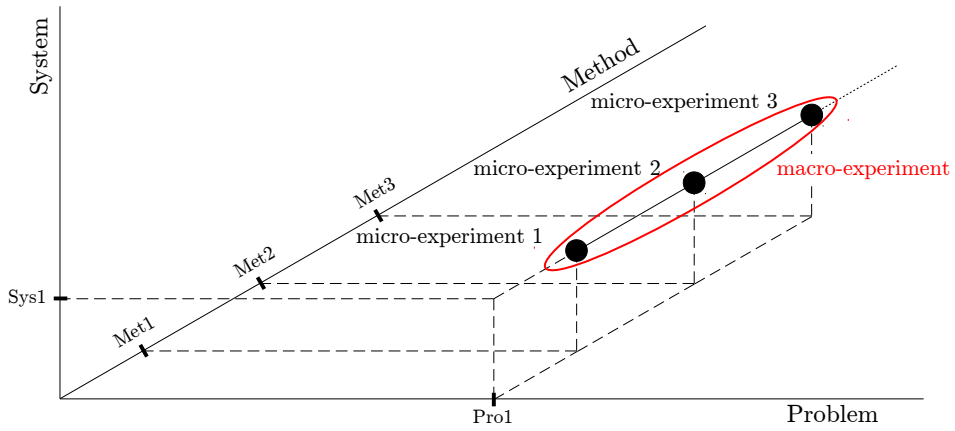



**Figure 4.1:** *Space of Computational Experiments*

The common experimentation need is to compare data resulting from more than a single micro-experiment. Keeping two out of the three dimensions fixed, we get an experiment that is a function of the third one: the red line shown in Figure 4.1 identifies such a *macro-experiment*, which is a

collection of micro-experiments (e.g., the black dots in Figure 4.1). Macro-experiments can be categorized as being either system-oriented, method-oriented, or problem-oriented.

## 4.2 Reproducibility levels

Based on the experiment taxonomy presented, in 2014, we initially defined 3 levels of reproducibility for an experiment [8], which we later refined in [5] as follows:

- **Repetition**: run the original *micro-* or *macro-experiment* without any variation of the parameters used. This should drive to the same results, which guarantees a certain level of credibility (**completeness** of documentation).
- **Replication**: run the original experiment on a different system. An experiment should not be bound to a specific computing environment (**portability**).
- **Re-experimentation**: run the original experiment changing only the method used. When this drives to the same outputs, the scientific approach is proven (**correctness** of the approach).

In 2018, the Association for Computing Machinery (ACM) also felt the need for terminology that could standardize the concepts around "reproducibility", proposing the first version of their Repeatability, Reproducibility, and Replicability definitions [73] lately reviewed following the convention explained in [74].

Even though the ACM terminology is now aligned with the National Information Standards Organization (NISO) Recommended Practice [75] and is actively used to review and assign badges to scientific research papers, its focus is mainly on the team running the experiment and the experimental setup. Instead, in our classification, we focus on the experiment elements *<problem, method, system>* (see figure 4.1), which may change in a usual "reproduction excercise": from simply re-executing an experiment to changing the computational environment or the approach used to run it.

In [69], the authors present 3 degrees of reproducibility for experiments in the AI field based on the experiment factors they identified: from the basic experiment reproducibility (R1), applying no variation to the original

experiment, to the data reproducibility (R2), changing the method implementation, to the method reproducibility (R3), changing both implementation and data used for the experiment. While R1 degree overlaps with our repetition level of reproducibility, both R2 and R3 degrees map to our re-experimentation; in fact, we make no differentiation based on the data used. Instead, our classification focuses on the "system" (hardware/software) performances defining a replication level of reproducibility when changing the system, which is missing in the degrees of reproducibility introduced in [69].

Clearly, it doesn't matter if an experiment runs faster when the results are wrong. For this reason, we consider the reproducibility of results embedded in the "performance reproducibility" and, thus, a prerequisite when comparing experiments in terms of performance: ensuring reproducibility of performance should automatically guarantee reproducibility of results.

By the way, our idea is not to stick with a specific reproducibility levels terminology but rather discuss the fundamental components that need to be considered to achieve reproducibility (and its acceptance) as a broader concept. First, re-run an experiment and get the same results (or, more generally speaking, insights) is a basic need for a scientist to ensure experiment value, as already stated by Karl Popper in 1934, "Non-reproducible single occurrences are of no significance to science" [76]. Second, setting up an experiment in a different computational environment is key on the one hand to increase trust and, on the other hand, to demonstrate that the experiment does not rely only on a specific system. Finally, an experiment that is not the outcome of a single occurrence and is trustable can be used to compare and validate other approaches that solve the same initial problem.

To compare the performance of different ML-specific hardware and software, it is fundamental to reproduce the experiments built by the DSAs experts to have a solid starting point for the comparison. A comparison of different experiments is what we call macro-experiment, which means to move an experiment in the space of experiments changing problem, method or system components. If we cannot reproduce the base experiment, it will be impossible to change the experiment components and create a new experiment to study the performance behaviour.

# Chapter 5

# Deep Learning Benchmarks

## 5.1 Benchmarks overview

As discussed in chapter 3, the accuracy improvement for AI solutions does not come for free but translates into a higher complexity: working with an increasing number of parameters and more extensive datasets makes the ML algorithms eager for performance improvements. In effect, to increase AI adoption, it is fundamental not only that the solution reaches a higher accuracy but also that it gets it in a reasonable amount of time.

The performance improvements can come from both algorithmic and system (both hardware and software) enhancements: it is critical to have a benchmark that can help adequately evaluate these ML solutions and, eventually, foster AI adoption and evolution.

In 2012, one of the first ML benchmarks, BenchNN [77], reimplemented a subset of Recognition, Mining, and Synthesis (RMS) applications from the PARSEC benchmark suite [78] replacing some target tasks with neural networks (NN) and comparing the quality of the "approximate" results given by the NN version with the original ones. BenchNN results showed how the NN version of the implemented applications was reaching a good quality in terms of output. However, poor execution time performance demonstrates, on the one hand, the potential of ML and, on the other, the need for specific hardware tailored to NNs. Nevertheless, the applications evaluated were not relevant as ML solutions [79].

Afterwards, benchmarks like DeepBench [80] have been explicitly created for evaluating ML considering the basic operations predominant in ML

algorithms, e.g., GEMM or convolution. Even though it is essential to improve low-level operations, this benchmark does not clearly show how good the approach solves a problem. For this reason, other ML benchmarks appeared to cover tasks solving specific ML problems, like image recognition, object detection, recommendation systems, language processing, and more (see section 3.2). In other cases, a benchmark may go further and even cover end-to-end scenarios composed of both AI and non-AI tasks [81].

Even when running complete ML apps, a benchmark may focus on a specific ML framework (see section 3.3) as in Fathom [79] or PerfZero [82], limiting the possibility of comparing how specific hardware behaves using different software. The same can happen on the hardware side with benchmarks targeting a single device type, e.g., the GPU for the TBD benchmark.

Another critical element for a benchmark is the metric used. Most benchmarks look at the performance using the throughput (e.g., sample/s) or time (e.g., time-per-epoch), while others look at the accuracy (e.g., top-5 accuracy) but both in an isolated way. However, an optimization may improve a specific "proxy" metric while adversely affecting another [83]. To remedy this situation, benchmarks like DAWNBench [84] started using the time-to-accuracy TTA metric, which measures the time needed by a system to reach a predefined accuracy value.

Beyond the differences mentioned, the aim of a benchmark is not to run an experiment merely to get a performance number. Instead, it should allow a fair comparison of systems and algorithms to foster a healthy competition, similar to how the Linpack [85] benchmark (HPL[72]) has been used since 1993 to draft the list of the faster 500 HPC systems in the world, driving their evolution.

Considering all the points about AI benchmarks we discussed, our desired benchmark should:

- Include workloads representative of problems relevant to the field.
- Evolve hand in hand with enhancements in both algorithms and systems.
- Use a metric that measures performance concerning accuracy.
- Enable a fair comparison of hardware and software technologies.
- Take into account reproducibility as a fundamental requirement.

Table 5.1 presents a list of AI benchmarks and their main characteristics. Among the ML benchmarks proposing a competition, MLPerf [86] seems the

most prominent, promoting experiment reproducibility and fair comparison. Some teams leading other benchmarks decided to stop and move to MLPerf, like DAWNBench, while in other cases, such as TBD, DLBS, and MLMark, they are still active though also contributing to the MLPerf benchmark.

Another newsworthy benchmark suite for AI is the AIBench [87], which covers a broader range of ML tasks, i.e., 19 (as of today), and workload types, i.e., basic operations to end-to-end scenarios. In this work, we will focus on MLPerf and, in particular, on the Inference benchmark. However, we believe the approach is generic enough to be applied to other MLPerf benchmarks and different benchmark suites like AIBench or even benchmarks in other domains where performance reproducibility is equally important [88].

## 5.2 Case study: MLPerf

MLPerf is a collection of machine learning benchmark suites started in 2018 as a joined effort from industry and researchers [89] to provide representative benchmarks for a fair system performance evaluation [90]. Since its formation in 2020, the MLCommons Association has maintained MLPerf to help the adoption of AI/ML by providing benchmarks, extensive open data sets and best practices [86]. As of today, MLPerf includes Training [90], Inference [44], HPC [91] (training), Tiny [92] (inference) and Mobile [93] (inference) benchmark suites.

MLPerf defines multiple categories and subcategories for each suite to allow a fair comparison, requesting them to comply with specific submission rules. Training and inference benchmarks are organized into two **divisions**: Closed and Open. The Closed division is defined to enable a fair comparison of software and hardware, fixing the model the benchmark need to use, which should be the same adopted in the reference implementation. While the Open division, relieving the model constraint, enable novel solutions which can reach the same target quality [90].

MLPerf divides the systems as well into different categories based on their availability to general users: **available** systems only include purchasable or rentable components, a system can be in **preview** if will become "available" in the following benchmark round, **Research**, **Development**, or **Internal** (RDI) category involves hardware or software which is experimental, underdevelopment or for internal-use only.

On the workload side, the applications part of the benchmarks may change from one suite to the other. However, they mainly solve tasks from

common areas like vision, language processing, commerce, game AI or, in the case of the HPC training benchmark, tackling particular tasks from a so-called **Scientific** area. MLPerf selected a specific ML model for each workload (fixed for the Closed division), which should represent the state-of-the-art solution for the ML task in terms of both performance and accuracy; that is why a chosen model may change with the benchmark versions. Both Closed and Open divisions define a fixed dataset.

The metrics may change from one benchmark to the other, but they need to reach a specific quality/accuracy target to be considered valid; that has applied to the time and throughput metrics of the training benchmarks as well as the latency and throughput of the inference ones (see table 5.2).

Finally, differently from the training, for the inference benchmarks, MLPerf does not only define the tasks to run and the rules to fulfil but also a complete benchmark framework for the inference. The structure is composed of the **LoadGen** in charge of generating the inference queries and the **System Under Testing** (SUT), which will receive and solve the requests. Such a benchmark architecture allows the LoadGen to simulate different realistic situations, the so-called **scenario**, which the SUT may deal with [44]:

- **Single-Stream**: the LoadGen sends a one-sample query after another as soon as the SUT completes them.
- **Multi-Stream**: the LoadGen sends an n-sample query based on a latency constraint, counting the queries the SUT can complete within the assigned constraint.
- **Server**: the LoadGen sends a new one-sample query to the SUT according to a Poisson distribution.
- **Offline**: the LoadGen sends all samples in a single query to the SUT at the start.

Based on the scenario, the metrics considered are different. For the *stream* scenario, it is essential to look at the latency the system can achieve while processing those streams. Instead, the other scenarios will focus on the throughput reached by the SUT.

Starting with the version v1.0 of the inference benchmark, MLPerf added measurements and metrics regarding **Power** consumed by the system to evaluate its efficiency. The metric depends on the specific scenario considered for these results (Table 5.2).

To promote a fair comparison, MLPerf defines some more rules for the results submission, e.g., schedule for the submission or mandatory scenarios for a specific device category or hyperparameters tuning [94]. After a couple of years from the launch of the benchmarks, we can already see some exciting trends for hardware and software improvements. For example, in the training benchmark, with the fourth submission round (v1.0) from June 2021, the performance improvement was 6.8 to 11 times higher than Moore's law advancing [95], which is an excellent example of how a benchmark contest and specific hardware and software usage can advance research fields like machine learning.

| Benchmark | Phase | Metric | Rank | Workload | Model | Dataset | Framework | Device | Release |
|---|---|---|---|---|---|---|---|---|---|
| DeepBench | T/I | Perf | no | ML operations (conv, GEMM, etc.) | NA | NA | NA | - NVidia/AMD GPU, Intel KNL (train)<br>- NVidia GPU, Embedded devices (infer) | 2016-2020 |
| DAWNBench | T/I | TTA | yes | Apps | Open | Fixed | Open | Open | 2018-2020 (stopped) |
| Fathom | T/I | Perf | no | Apps | Fixed | Fixed | TensorFlow | CPU, GPU | 2016-2019 |
| TBD | T | Perf, Acc | no | Apps | Fixed | Fixed | TensorFlow, MXNet, PyTorch, CNTK (deprecated) | GPU | 2018-2020 |
| DLBS | T/I | Perf | no | Apps | Fixed | Fixed | Caffe, Caffe2, PyTorch, TensorFlow, TensorRT, MXNet, OpenVINO (experimental) | CPU, Nvidia GPU, AMD GPU (limited) | 2017-2021 |
| AIMatrix | T/I | Perf | no | Ops, Apps (Alibaba DCs) | Fixed | Fixed | TensorFlow, Caffe | NVidia GPU, Cambricon | 2017-2020 |
| MLMark | I | Perf, Acc | yes | Apps | Fixed | Fixed | OpenVINO, TensorRT, ARMNN, TensorFlow, TFLite | CPU, GPU, ARM, TPU | 2019-2020 |
| TF bench (PerfZero) | T/I | Perf | no | Apps | Fixed | Fixed | TensorFlow | CPU, GPU, TPU | 2016-2021 |
| AIBench | T/I | Perf, Acc | yes | Ops, Apps, E2E scenarios | Fixed | Fixed | TensorFlow, Pthread, PyTorch, Open | CPU, NVidia GPU, TPU, Open | 2018-2021 |
| MLPerf | T/I | Perf, Acc | yes | Apps | Fixed | Fixed | Open | Open | 2018-2021 |

**Table 5.1:** *A list of AI benchmarks.* ***Benchmark*** *is the name of the benchmark,* ***Phase*** *is the training face the benchmark focus on, which could be training (T) or inference (I),* ***Metric*** *is the metric type considered (performance/accuracy),* ***Rank*** *set to "yes" for the benchmarks used to build a ranking,* ***Workload*** *lists the categories of workloads in the benchmark,* ***Model*** *and* ***Dataset*** *are set to Open if decided by the user or Fixed when defined by the benchmark,* ***Framework*** *is Open when defined by the used for performance ranking or a specific one if the benchmark provides a fixed reference implementation,* ***Device*** *is the list of the devices supported by the benchmark,* ***Release*** *shows when the benchmark creation date and what is the current stable release.*

| Name | Workload | | | | | |
|---|---|---|---|---|---|---|
| | **Area** | **Task** | **Model** | **Dataset** | **Target quality** | **Metric** |
| Training | Vision | Image classification | ResNet-50 v1.5 | ImageNet | 75.90% classification | • Time-to-train |
| | | Image segmentation (medical) | 3D U-Net | KiTS19 | 0.908 Mean DICE score | |
| | | Object detection (heavy weight) | Mask R-CNN | COCO | 0.377 Box min AP and 0.339 Mask min AP | |
| | | Object detection (light weight) | SSD | COCO | 23.0% mAP | |
| | Language | Speech recognition | RNN-T | LibriSpeech | 0.058 Word Error Rate | |
| | | NLP | BERT-large | Wikipedia 2020/01/01 | 0.72 Mask-LM accuracy | |
| | GameAI | Reinforcement Learning | Mini Go | Go | 50% win rate vs. checkpoint | |
| HPC (training) | Scientific | Climate segmentation | DeepCAM | CAM5+TECA simulation | IOU 0.82 | • Time-to-train (strong scaling)<br>• Throughput, i.e., number of trained model instances (weak scaling) |
| | | Cosmology parameter prediction | CosmoFlow | CosmoFlow N-body simulation | Mean average error 0.124 | |
| | | Quantum molecular modeling | DimeNet++ | Open Catalyst 2020 (OC20) | Forces mean absolute error 0.036 | |
| Inference (Datacenter / Edge) | Vision | Image classification | ResNet-50 v1.5 | ImageNet | 99% of FP32 (76.46%) | (Scenario-dependent)<br><br>Performance metrics:<br>• 90%-ile Latency (SingleStream)<br>• 99%-ile Latency (MultiStream)<br>• Queries/s throughtput (Server)<br>• Samples/s throughtput (Offline)<br><br>Power metrics:<br>• Energy-per-stream (Single/MultiStream)<br>• System power consumption (Server, Offline) |
| | | Image segmentation (medical) | 3D U-Net | BraTS19 | 99% of FP32 and 99.9% of FP32 (0.85300 mean DICE score) | |
| | | Object detection (heavy weight) | SSD-ResNet34 | COCO | 99% of FP32 (0.20 mAP) | |
| | | Object detection* (light weight) | SSD | COCO | 99% of FP32 (0.22 mAP) | |
| | Speech | Speech recognition | RNN-T | LibriSpeech dev-clean | 99% of FP32 (1 - WER, where WER=7.452253714852645%) | |
| | Language | NLP | BERT-large | SQuAD v1.1 | 99% of FP32 and 99.9%** of FP32 (f1_score=90.874%) | |
| | Commerce | Recommendation** | DLRM | 1TB Click Logs | 99% of FP32 and 99.9% of FP32 (AUC=80.25%) | |
| Mobile (inference) | Vision | Image classification | MobileNetEdgeTPU | ImageNet | 98% of FP32 (Top1: 76.19%) | |
| | | Object detection (light weight) | MobileDETs | COCO | 95% of FP32 (mAp: 0.285) | |
| | | Segmentation | DeepLabV3+ (MobileNetV2) | ADE20K | 97% of FP32 (32-class mIOU: 54.8) | |
| | Language | NLP | Mobile-BERT | SQuAD v1.1 | 93% of FP32 (F1 score: 90.5) | |
| Tiny (inference) | Vision | Image classification | ResNet-8 | CIFAR-10 | 85% (Top 1) | |
| | | Visual Wake Words | MobileNetV1 0.25x | Visual Wake Words Dataset | 80% (Top 1) | |
| | Speech / Audio | Keyword Spotting | DS-CNN | Google Speech Commands | 90% (Top 1) | |
| | | Anomaly Detection | Deep AutoEncoder | ToyADMOS | 0.85 (AUC) | |

***** Edge only. ****** Datacenter only

**Table 5.2:** *MLPerf benchmark suites.*

# Chapter 6

# Benchmarking with MLPerf

To understand ML experiments' challenges, we tried to reproduce some of the results submitted during the first "round" of the MLPerf benchmark and, in particular, the Inference one.

MLPerf provides access to the benchmark in a git repository which includes:

- the Load Generator (LoadGen) used to feed the inference query to the model.
- information regarding the model calibration process.
- the tools for validating the submission.
- the information for all the benchmarks in the suite, including reference implementations and scripts, to validate the model accuracy.

MLPerf LoadGen is provided as a C++ library with Python bindings, while the reference implementations and their helper scripts are only meant to "familiarize" with the benchmark.

After the benchmark round is open, the participants will get the benchmark and reimplement the code, optimizing it for their specific software and hardware. Before the deadline, they will submit all the requested files, peer-reviewed by the other submitters. Finally, the accepted submissions will be published in a separate git repository.

The submission structure, defined by the benchmark policies[94], includes information about the system used, code and instructions to run

the submission, scripts and configurations used for the submission, and benchmark results. Looking into these submissions was the first step to try reproducing the results for the MLPerf Inference benchmark v0.5.

## 6.1 Reproducing MLPerf Inference: A user journey

Among the submissions, the Intel OpenVINO framework provides an interesting case. It is an inference engine optimized for Intel devices (CPU, GPU, VPU, FPGA) [96] which aims to run transparently on any supported devices with minimal configuration effort.

The original OpenVINO experiments run on Windows or Linux OS systems. Some README files document the steps to configure the environment and the experiment to run: reproducing the submission requires an entirely manual process.

Using the instructions provided, we decided to build the environments as containers. Starting from the container definition file, it was possible, on the one hand, to always initiate our tries from a clean environment and, on the other, to have better confidence about the correctness of the environment. However, with our little knowledge about the OpenVINO software and a not completely clear list of actions, e.g., there was no clear indication about the OpenVINO version to use, it was not easy to understand the different configurations. For this reason, the first decision was to try getting more familiar with the tool, installing it following the official documentation (instead of the MLPerf submission instructions), and adding the MLPerf LoadGen on top of it.

MLPerf provides the model as part of the inference benchmark, but in the case of OpenVINO, the user needs to translate the model in a different format. Still, creating an OpenVINO model starting from the one provided is well documented and relatively straightforward, thanks to the official prebuilt OpenVINO Docker containers.

With this first combination of software, i.e. standard OpenVINO + Intel code + ML LoadGen, the build of the experiment failed because of some incompatibilities with the OpenVINO version. After acquiring more knowledge regarding OpenVINO and the errors, we could slightly change the code to have a first working version.

Nevertheless, changing both hardware and software simultaneously made it impossible to compare the experiments in terms of performance. So, we

decided to look at the code which Intel submitted to the sequent MLPerf inference round: v0.7. This time, the instructions like software versions and steps were more explicit. Thanks to these instructions and the increased OpenVINO knowledge, it was possible to reproduce the same environment. In this submission, Intel used the OpenVINO code only to run the Offline scenario. Still, since the scenario is just a parameter in their code, we could use it to run the Single-Stream scenario and compare it with our previous experiment.

At this point, we could go back to the original experiment and complement some information we missed in our first try: the MLPerf LoadGen used was not the standard one, but a version "patched" with two pull requests in the GitHub repo. Using this different version of MLPerf LoadGen and building from the OpenVINO version available at the submission time, we could reproduce the environment configuration of the original experiment for version v0.5 of MLPerf.

While a machine learning scientist would probably only look for (1) running the code on his hardware and (2) understanding results compared with the ones available in the MLPerf submission, would he/she be able or even interested in going through all these steps to reproduce this experiment?

For each container, we generated both a Docker and a Singularity version. A CI pipeline connected to a GitHub repository builds the containers and pushes them to a container registry. Providing the environment as a container is helping to reproduce the experiments, but a user would still need to manage changes manually to both experiment configuration and systems to use. Those aspects can be simplified using a workflow system.

## 6.2 Support tools

As discussed in section 7.3, many tools address experiments complexity and assist reproducibility. In addition, people have already used some of them to reproduce machine learning experiments and MLPerf benchmarks.

**Popper**[*] is a framework that helps define and run scientific workflows leveraging container technology. The tool is built upon the *Popper Convention* [97], which suggests following a DevOps approach: (1) select a DevOps tool for each stage of the experiment, (2) use a Version Control System to for all the scripts involved and (3) document the experiment changes in the version control commits. The tool looks mature and follows a similar

[*]https://getpopper.io

approach and motivation we had for our work on [8]. Still, Popper was not available back then. There is an example on GitHub[†] using Popper for the MLPerf training benchmark, but it does not seem more than a try.

Another tool focused on machine learning experiments is the **Collective Knowledge** framework (CK) [98]. The idea behind CK is to consider an experiment as a collection of components that capture the experiment artifacts (code, scripts, datasets, models, ...) to be managed through CLI/API and, thus, easily reused. Moreover, the components are organized as a database to enable the FAIR[‡] principles. Finally, CK officially collaborates with MLPerf/MLCommons and provides reproducibility studies about the benchmarks and is actively used for some submissions.

Even the MLPerf community itself has realized the need to have a way to simplify the execution of the benchmark and help reproducibility. So they started the development of **MLCube™** [99]. [100] describes MLCube as "a consistent interface to machine learning models in containers like Docker. Models published with the MLCube interface can be run on local machines, on various major clouds, or in Kubernetes clusters, all using the same code". MLCube aims to help researchers and developers build ML models that can be easily shared and reproduced. However, even though MLCube is promising and is currently growing with more and more "runners" added (ssh, Docker, Singularity, Kubernetes, cloud), it is still at its early stage.

The next chapter will discuss how we overcome the issues faced during the MLPerf inference benchmark reproduction using our workflow tool: PROVA!. PROVA! have many aspects in common with other tools managing the experiment "flow". Nevertheless, the main difference is for PROVA! to have micro-/ and macro-experiment as central concepts: reproducing an experiment and helping users vary one or more of the experiment *dimensions* to characterize the software/hardware performance [8]. Furthermore, on the one hand, its web interface helps manage the experiments and connections to different remote systems. On the other hand, it shows how it can interact and extend the PROVA! backend [101].

[†]https://github.com/getpopper/mlperf-training-workflows
[‡]findable, accessible, interoperable, and reusable

# Part III

# PROVA! 2.0: A Benchmark Driver

# Chapter 7

# Experiment Challenges in HPC

## 7.1 Software Stack

With the increasing complexity of the architecture, the software needed to use and manage them is getting more and more complex. This complexity comes not only from the software itself but also from all its dependencies.

### 7.1.1 Environment modules

To help deal with the software complexity in the HPC world, HPC centers are extensively using Linux modules. A user can quickly "load" and "unload" a module for specific software and version, being sure that the "environment" required gets set in the right way. System admins can write the file describing Environment modules, called modulefile, using the Tool Command Language (Tcl) or later alternatives. Among those Linux modules alternatives, it is worth mentioning Lmod: the solution developed at Texas Advanced Computing Center (TACC) and based on Lua language.

Even though Environment Modules/Lmod helps manage software in an HPC environment, they are not trivial to write and update. Furthermore, manually writing modules representing complex software with many dependencies can become very tedious. For this reason, tools like EasyBuild and Spack have been built. Both EasyBuild and Spack help automatically install the software (with all its dependencies) and produce the correspondent modulefiles for configuring, cleaning, or changing the environment. They also provide templates for standard software build and installation proce-

dures and an extensive database of software "recipes" which can be used as-is or customized based on the need.

### 7.1.2 Linux containers

Using environment modules, the user must install different software and versions on a system to switch from one version to another and run various experiments. This process may take some time and even end up in conflicts when loading multiple software with different versions of the same dependency. In this case, a cleaner approach to software management can be achieved using Linux containers.

#### From Chroot to Docker revolution

Containers are an OS-level virtualization technology. Unlike Virtual Machines, they do not require a complete OS and hardware virtualization but share OS kernel (with other containers) and include all the packages, binary, and libraries the software users want to run.

The first container technology appeared back in 2000 with FreeBSD adding their container concept, the *Jails*, to the OS; this followed the first OS feature usually considered part of the container world: chroot. Chroot was introduced in 1979 within the Unix V7 development and added to BSD in 1982 to test its installation. It allows changing the root directory for the calling process and its children processes to a different path, creating a separate and isolated environment called *chroot jail*.

Later on, the container technologies evolved: the processes running within a Linux container are isolated from the rest of the system using **namespaces** and **control groups** features provided by the kernel. The different namespaces will limit the resources, e.g. process IDs (pid), mount points (mnt), network stack (net), a process can *see*, while the cgroups limit the amount of a particular resource (e.g. Memory, CPU, I/O) the process can *use*.

If, on the one hand, container solutions like OpenVZ [102], Solaris Containers [103] and LXC [104] raised the interest in containers, on the other, the real breakthrough came with Docker.

Docker is an open platform for developing, shipping, and running applications [105]. It started in 2013, focusing on containers running a single application/service (application container) instead of combining multiple

services like in OpenVZ or LXC (OS container), making Docker containers lighter and more suitable for a microservices architecture.

While previous container technology like LXC may need machine-specific configuration [106], Docker containers are both hardware and platform agnostic, allowing great portability. Docker containers are built from a "Dockerfile", including the steps (mainly shell commands) needed to configure the environment in the container. Each of the Dockerfile steps will produce a *layer* of the final Docker image, which stores the changes (diff) to the previous layer. After being built, the container can be easily shared through a remote registry, both public or private, and re-used or extended by others.

Docker is based on a client-server architecture composed of a **daemon** and a **client**: a Docker daemon process runs on all the hosts used by Docker, serving the client requests for building, running and distributing the container. Only a user with elevated privileges can use Docker CLI commands by default.

Despite being the most adopted container technology is a clear sign of its reliability in managing software applications, its usage in the HPC field shows some concerns like the need for a running daemon and root privileges, the system overhead [107], and the missing support for workload managers and parallel storage driver [108].

### Containers in HPC

Several technologies appeared to cope with the HPC needs, which Docker lacks.

**Podman** Podman [109] is defined as an open-source, "daemonless container engine for developing, managing, and running Open Container Initiative (OCI) [110] containers and container images on your Linux System" [111]. Podman is fully compatible with both the Docker CLI and the Docker container image format. However, unlike Docker, it does not need a daemon and, using user namespace mapping of UID/GID, can run in *rootless* mode. Podman also has more features not related to the HPC field, like integration with **systemd**, which, for example, allows enabling the Podman API [112] to use Docker-compatible remote container management. At the same time, it lacks some HPC-related features like easy integration with MPI or the rootless mode for distributed filesystems [113].

**Shifter** Shifter [114] is one of the first attempts to enable Docker containers in HPC environments, particularly Cray systems. Started back in 2015 by NERSC, it builds a complete architecture to manage the acquisition and conversion of Docker containers to a common format and be then used through the workload manager leveraging the Linux **chroot** operation. With such an architecture, Shifter shields the system from the security implication of Docker execution without requiring the user to change their Docker images. Still, its installation/configuration is not straightforward and highly tight to Cray systems, making it hard to use in other environments.

**Charliecloud** Charliecloud is a lightweight, open-source user-defined software stack (UDSS) implementation for HPC centers with the design goals of (1) providing a standard, interoperable and reproducible workflow, (2) running on existing HPC hardware and software with minimal changes, and (3) be straightforward [115].

The UDSS get built as a container starting from a standard Dockerfile using Docker or other container builder tools, e.g. Buildah [116], as an independent Linux filesystem tree. To manage the UDSS container, Charliecloud leverages only two Linux namespaces, i.e. user and mount, and no control groups at all: the **user** namespace allows to map UID/GID used in the container to the real UID/GID for the user on the host, while the mount can bind-mount path from the host to the container. This approach guarantees the minimum functionalities needed in HPC environments without needing a daemon or elevated privileges.

**Singularity** Singularity is an open-source project started in 2015 at Lawrence Berkeley National Laboratory to "bring containers and reproducibility to scientific computing" [117]. The Singularity container technology specifically targets HPC systems providing support for MPI, Infiniband, GPU, integration with workload managers, and avoiding privilege escalation. As for the other Docker HPC alternatives, it does not need a daemon.

Singularity containers are usually built starting from a Singularity definition file, equivalent to a Dockerfile, and stored as a single file using the so-called Singularity Image Format. Moreover, Singularity provides support to Docker containers, which gets automatically translated prior to their execution.

It supplies three configurations for non-root execution:

- **setuid**: the runtime binary "temporarily" gains root privileges to execute operations that need privileges like filesystem loop mounts.
- **user namespace**: it maps all files in the container filesystem to the same unprivileged user running on the host and all the bind-mounted paths belonging to other users to *nobody/nogroup*. It only works with a Linux filesystem tree container (sandbox mode).
- **fakeroot**: still uses user namespace but need UID/GID mapping for the user to allow acting as a different user (including root) in the container. It only works with a Linux filesystem tree container (sandbox mode).

**SARUS** A more recent effort for an HPC-specific container technology is called Sarus [108]. Developed at CSCS in 2019, Sarus aims to run Linux containers compatible with open standards, i.e. Open Container Initiative (OCI), and address HPC systems needs.

Like Shifter, Sarus can start from Docker containers (based on OCI standard), converting them to a custom format (Sarus-specific in this case). On top of this, it adds the OCI Hook to the OCI Runtime to add support for HPC-related needs like support for MPI, GPU, SSH or workload manager.

Even though the described container technologies can present different pros and cons, they all have an explicit interest in integrating this technology into the HPC world, showing a firm trust in this approach for software stack management.

## 7.2 HPC Systems Interaction

Beyond the complexity given by configuring and programming specific hardware used for deep learning experiments (see section 3.3), a challenge for an experiment can be related to how users access and interact with those devices. These specific deep learning devices are typically available in data centers or HPC clusters and accessible via a remote connection. The most common way to access them is through an SSH connection and interact using shell commands which not all users may be familiar with and probably not even something they would like to deal with. Scientists working with machine learning are used to powerful workstations (possibly equipped with GPU accelerators) under their desks and high-level support such as given by

Python and R environments. Using a remote system can be challenging for them. For instance, tools like Jupyter Notebook and RStudio can expose a web application as an "interface" to the execution system to alleviate those issues. In this case, the users can access and program systems remotely using their favourite application through a simple web browser.

The other challenge related to the execution environment is managing the resources needed for the experiment. In the case of an owned system and with exclusive access granted, this issue may not be critical. However, it may still be helpful to have a way to limit resources used by the experiment so that the user can, for example, test how effectively the resources are being used (scaling analysis). Instead, when dealing with HPCs, the massive amount of the computational power provided will be assigned to various workloads. Rarely a single experiment will need to access the entire pool of resources and certainly not in a continuous manner. For example, even if the single phases of an experiment may require different amounts of resources: generating data is likely to have a greater demand for resources than analyzing it.

The access to these shared resources is usually managed through a workload manager, also called a scheduler, and the user is asked to interface with it, which may not be an easy task for all users. In addition, more workload management solutions exist, and the user may have to handle different schedulers when interfacing with multiple systems. Even though Slurm seems to be the scheduler taking the lead [118], not all users might find it trivial to deal with it. Moreover, the growing interest in Linux containers, discussed in the previous section, is driving the adoption of container-focused workload managers, such as Kubernetes, also in the HPC field [119][120][121][122]. A possible solution is to provide a scheduler-agnostic way to allocate resources to help users interact with current HPC systems and support the possible evolution of workload managers.

## 7.3 Experiment Workflow

We can manage the steps needed to carry on an experiment in different ways based on their complexity: we could use from a few simple scripts to complicated tool-managed workflow. A usual experiment workflow includes multiple phases, from the preparation (data, code, parameters) to the actual execution, to gathering the results and generating a report. Furthermore, each phase may be composed of multiple steps and need to exchange infor-

mation, raising the workflow complexity. Moreover, we need to store the precise configuration of the experiment so that we can (1) reproduce, (2) analyze, and (3) customize it. In fact, if we want to understand the performance behaviour of an experiment, we cannot only analyze the results for a specific fixed configuration but need to test various combinations.

There are many workflow tools available with different characteristics and focus. In our previous work [2], we analyzed several workflow management systems (WfMS) proposed to describe the workflow of an application and make it reproducible.

Some WfMSs, mainly spread in natural science, like Galaxy [123], can manage complex computational biology and bioinformatics workflows by integrating data acquisition, derivation, analysis, and visualization as executable components throughout the scientific exploration process, while others may have a focus on the HPC field: Pathway [124] is a tool for designing and executing performance engineering workflows for HPC applications, DataMill [125] is a community-based easy-to-use services-oriented open benchmarking infrastructure for performance evaluation, which facilitates producing robust, reliable, and reproducible results. It provides a platform for investigating interactions and composition of hidden factors affecting the performance measurements, such as binary link order, process environment size, compiler-generated randomized symbol names, or group scheduler assignments. Besides the ones discussed in our previous work, also in the machine learning field, there are many workflow tools available, such as Airflow [126] or MLFlow [127], which are defined as "platforms" created to manage the workflows and enhance reproducibility.

Since working with complex systems and architectures, a workflow tool should also help to collect system information during the experiment execution, which can be later used to (1) understand the experiment outcomes, (2) detect possible sources of misbehaviours. While these tools identify many useful features one needs, none of them enables one to undertake performance experiments, targeting both reproducible results and reproducible performance and easy access to the resources.

# Chapter 8

# PROVA! 1.0: Performance Reproduction of Various Applications

## 8.1 Definition and Motivation

The challenges regarding experiments (Chapter 7) and their reproducibility (Chapter 4), together with the necessity of having a way to manage different and increasingly complex post-moore hardware (Chapter 2), are the main reasons behind the tool we developed.

The PROVA! project aims to help the user in his journey to deliver reproducible research by hiding the complexity of the environment and the maintenance of the software stack, storing valuable information about the system used to carry on an experiment and the experiment configuration details.

PROVA! was born to manage high-performance computing experiments like stencil experiments [3][128]. Still, we recently added some new features such as the **container** (see Section 9.2) support and the **driver mode** (see Section 9.3) to make it more flexible for other research fields.

### 8.1.1 Contributions to the Project

PROVA! is a project started within the high-performance and web computing team with the collaboration of Danilo Guerrera. The definitions of

the experiment taxonomy and reproducibility levels were produced together with him. In the initial version of PROVA!, he concentrated on managing the software environment through Linux modules and scientific software management tools and executing the jobs on a parallel machine. At the same time, I implemented the code around the collection of results and performance graph generation for the experiment and the design and implementation of the PROVA! web application (Section 8.2.2). Chapter 9 will discuss the new version of PROVA!, which I designed and implemented, improving some existing features and adding new ones like container support and driver mode.

## 8.2 Architecture

PROVA! does not require any specific configuration of the remote systems and can interface different systems: from an HPC cluster with both traditional and accelerators resource requestable through a job scheduler to a single node machine possibly, as well, connected to some accelerators to an HPC cluster with only CPU nodes (see figure 8.1).

The PROVA! architecture is mainly composed of an **Experiment Management Tool** and a **web application**. The Experiment Management Tool represents the core of PROVA!. It must be installed on each system the user wants to run an experiment and needs to be accessible through an SSH connection, usually to a front-end machine. The web application comprises an Experiment and Analysis Server, which manages the experiment by communicating with the remote PROVA! installation, and a web interface. Differently from the Experiment Management Tool, the deployment of the PROVA! Experiment and Analysis Server is not needed for each remote system since it can access multiple of those, and, likewise, the same remote system can be accessed by multiple PROVA! web servers. Finally, a PROVA! web application can serve multiple users, which means that, in principle, it could even be deployed by a third-party and provided as-a-Service so that users only need to create an account and start setting up their remote connections. Even though the PROVA! web application provides a default web UI, this is just a "view" for the information generated by the Experiments and Analysis Server. That is why it would be possible to replace the web UI with a different existing tool that communicate to the web server using HTTP requests. We presented this possibility in [101] using Jupyter [129] as a proof-of-concept interface integrated with PROVA!.

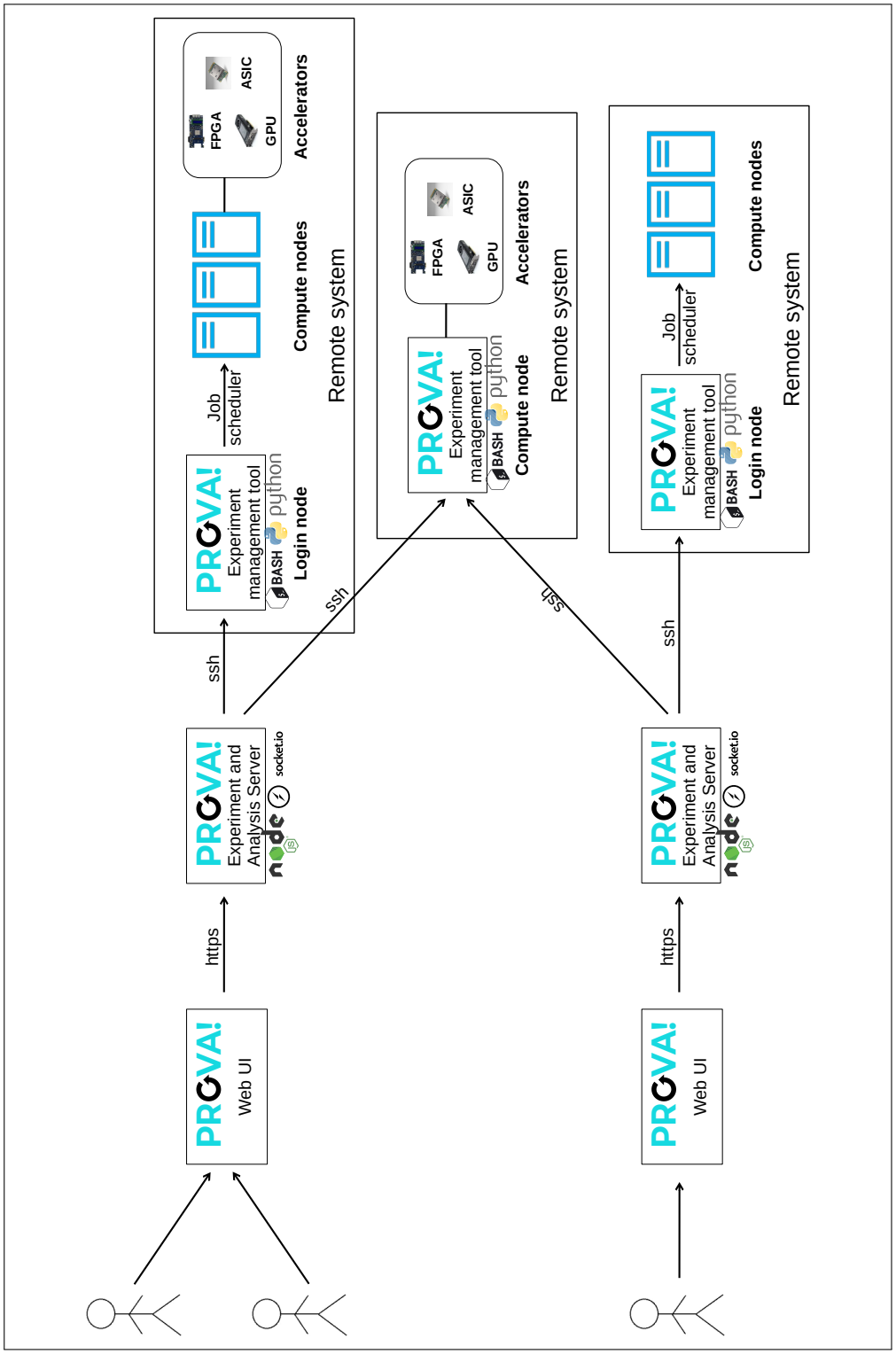



**Figure 8.1:** PROVA! *architecture: High-level view*

### 8.2.1 The PROVA! Framework

The remote system where the user wants to run his experiment needs to have the PROVA! framework installed in a path accessible by all the machines which will need to run the experiment (usually a shared filesystem). The user with a valid account on the remote system can connect and use the PROVA! CLI to manage his projects and experiments.

The framework consists of a collection of bash and python scripts that can be installed at the system level (by a superuser/admin) and shared with all the users or in a private path (by a normal user). The user needs to specify if he wants to use PROVA! with EasyBuild modules (section 7.1.1): he can either install EasyBuild through PROVA! or specify another existing EasyBuild installation. Instead, if the user only wanted to use software through Linux containers (section 7.1.2), the PROVA! installer will download its dependencies as a container. The details for the container support in PROVA! will be discussed in Section 9.2.

After the PROVA! framework installation is over, the user can start executing the *workflow* command and its subcommands. As shown in figure 8.2, the main PROVA! folder contains the workflow command, which will call a script from the *scripts* folder based on the action a user wants to execute. Using the web UI, the user can execute most of the commands. The only commands which need to be executed from the PROVA! CLI are the ones related to the methodTypes' installation.

The role of a methodType is to manage the software needed by a specific method. Apart from installing the software and its dependencies, it defines the scripts used by PROVA! to set up, compile and run the method with that methodType (see figure 8.2). For example, suppose a user develops a method using the C language, the correspondent methodType could (1) install a GCC compiler through a procedure defined in the **install** script, (2) define a **setup_params** script which creates a header file with the parameters' values, (3) include a **compile** script which runs the GCC command to compile the code and (4) a **run** script which runs the executable generated by the compile script.

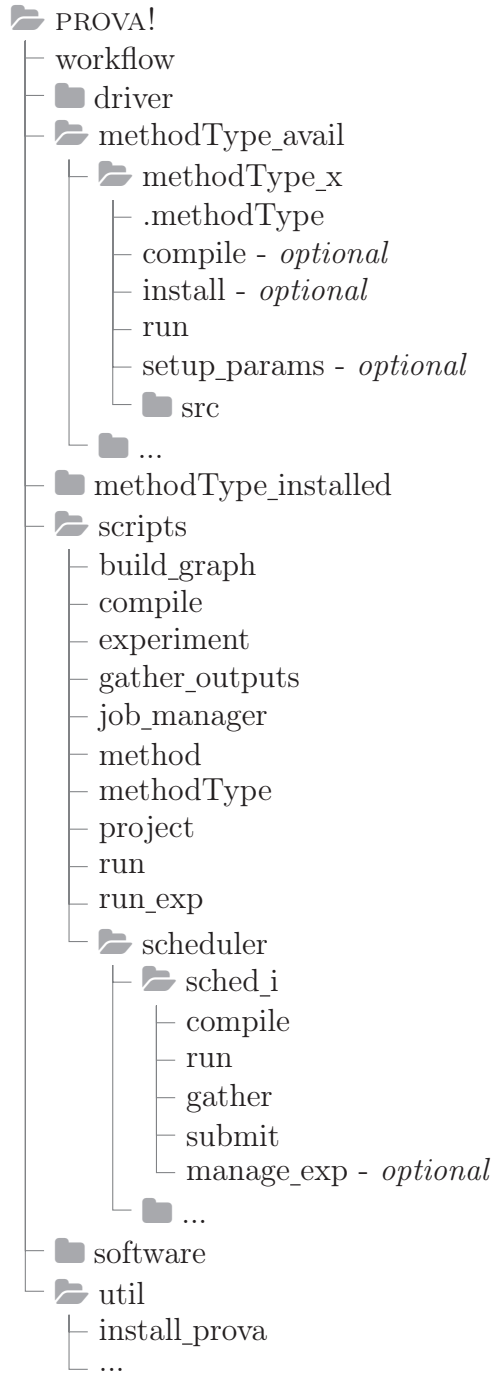



**Figure 8.2:** *Structure of the* PROVA! *framework*

The methodType information used by those scripts are stored in the methodType descriptor file, named *.methodType*, which is structured as follow:

- **name**: A unique name for the methodType.
- **eb_modules**: The list of EasyBuild modules needed by the methodType.
- **container**: The object containing the information about the container needed by the methodType (Section 9.2).
- **version**: A string representing the methodType version.
- **comment**: A comment/description for the methodType.

The methodType subcommand, by default, acts on methodType_avail and methodType_installed folders in the PROVA! root directory and, thus, may need to be run by the system admin. Any user on the system may utilize the methodTypes installed in the PROVA! root directory, but PROVA! also gives the possibility to create them in the user workspace (figure 8.3). In this case, the methodTypes will be private to the user and not shared.

A user can also execute the other commands after being connected to the remote system. However, in the majority of the cases, those run from the PROVA! web application (Section 8.2.2): based on the value selected in the PROVA! UI, the PROVA! commands get built and sent for execution on a remote system. This way, the user will be shielded from the commands' complexity. The only requirement is for the remote system to be accessible by the web application. We can logically divide the possible workflow subcommands (see scripts folder in figure 8.2) into groups based on the functionalities provided: development (project, method), experiment (compile, run, gather_output, run_exp), scheduler (job_manager, scheduler) and visualization (experiment, build_graph).

The commands used for developing an experiment are **project** and **method**, which serve the CRUD actions. Following our experiment taxonomy (Section 4.1), to solve a **problem**, a user can create a project in PROVA!, including the specification of its parameters. The project information is stored in a descriptor that has the following fields:

- **name**: A unique name for the project.
- **type**: A string to define if the project is using a driver.
- **parameters**:
  - **names**: An ordered list of the parameters' names.

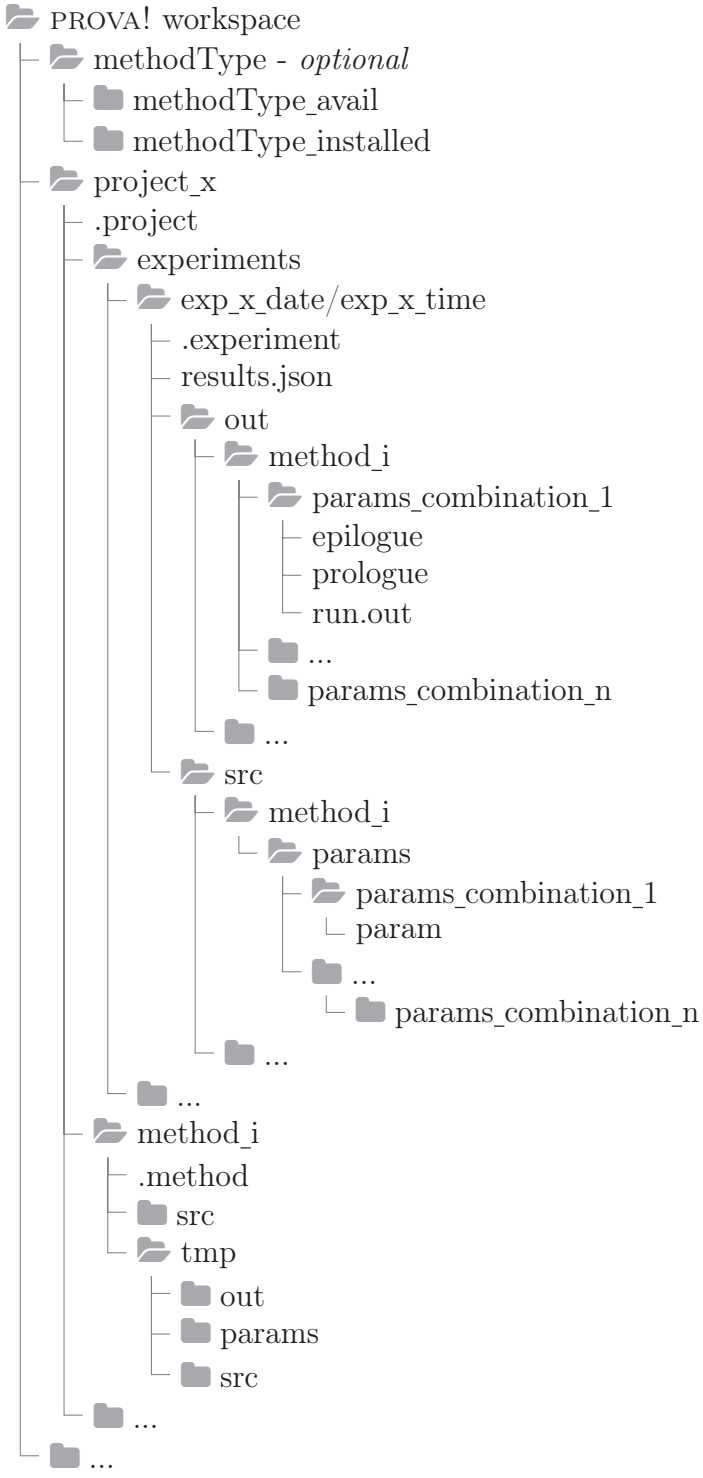



**Figure 8.3:** *Structure of a* PROVA! *workspace*

  - **defaults**: An ordered list of the parameters' defult values.

- **threads**: The default value for the total number of processes requested by the experiment.
- **metrics**: The list of possible metrics available as output of the experiment execution.
- **lineselector**: A string used to identify the line containing the output results.
- **comment**: A comment/description for the project.

The second step is to create one or more methods that represent the solution to the problem, which we identify with a **method**. A method in PROVA! includes a specific methodType which defines the software needed by the method (software part of a **system**) and the scripts to manage the phases of an experiment on the remote system (hardware part of a **system**). Apart from the actual implementation, which completely depends on the user, the descriptor of a method includes the follows:

- **name**: A unique name for the method.
- **type**: A string defining the name of the methodType used.
- **local**: A "true"/"false" string defining if the method is using a local or system methodType
- **comment**: A comment/description for the method.

After the user adds a method, he can test it by running a compile/run command: PROVA! will use the compile and run scripts together with the corresponding compile and run scripts specific to the methodType and execute the test against the default parameter values specified in the project descriptor. The user can repeat this for all the methods added to the project and start a micro-/macro-experiment.

The command to run the experiment, i.e., **run_exp**, will take the user-defined list of parameters and values and run the experiment workflow (compile → run → gather_outputs) for each possible combination of the parameters' values. The **gather_outputs** script will use the *lineselector* string and the *metrics* list to build a JSON object with the results, while the last step of the run_exp script will be saving an experiment descriptor including all the experiment details needed to reproduce it in the future (Section 9.1.2):

- **date**: A unique "date/time" string to identify the experiment
- **project_name**: The name of the project used for the experiment.

- **methods**: The list of methods used for the experiment.
- **par_names**: An ordered list of parameters' name used for the experiment.
- **parameters**: An ordered list of the parameters' values used for each of the experiment's parameters.
- **nthreads**: A list of the total number of processes to request for different runs on the experiment.
- **pin_strat**: A string defining different possible pinning strategies for the threads/processes to the system cores.
- **executions**: The number of times to repeat an experiment execution.
- **scheduler**: The name of the scheduler available on the system.
- **use_mt**: A "0"/"1" string specifying if the experiment used (1) or not (0) multithreading.
- **reproid**: A "date/time" string identifying the master experiment if the current experiment has been reproduced from another experiment.
- **reproduced**: An array of "date/time" strings identifying the experiments reproduced from the current one in case the this is a master experiment.

If a remote system requires a job scheduler, PROVA! can run all the experiment-related commands through the scheduler adding the "job_" prefix to the command. In this case, the **job_manager** script will figure out the scheduler configured for the remote system and use the proper job scheduler interface from the **scheduler** folder (see figure 8.2). A job interface should include a wrapper for the compile, run, and gather commands to create the corresponding job script and submit it to the job queue. Moreover, PROVA! can provide advanced job management whose logic can be added to the **manage_exp** script for some job schedulers. More details about the advanced job scheduler management will be discussed in Section 9.1.1.

Once the experiment terminates and its results are written, it is possible to use the **experiment** command to visualize, delete or reproduce it (Section 9.1.2) and the **build_graph** command to visualize the experiment results (Section 9.1.3).

### 8.2.2 The PROVA! Web Application

The PROVA! web application uses the Node.js runtime environment [130], allowing the server and client-side of a web application to use one programming language: Javascript. The web application architecture is based on MVC pattern: the user accesses the web application through a web browser, the HTTP requests get routed (Express.js [131]) to the appropriate **controller**, which will get the **model** information either from the DB (user information) or sending a request to the PROVA! backend installed on the remote system (experiment information). The controller uses then the model and the view template to build the final **view** and send the response to the browser (figure 8.4).

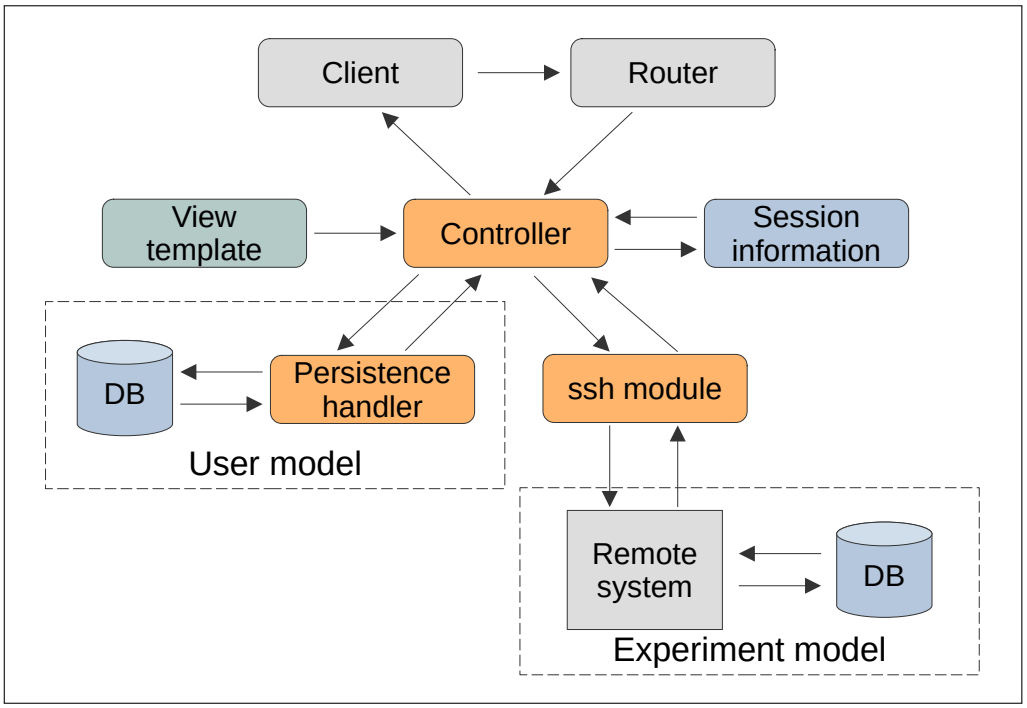



**Figure 8.4:** PROVA! *web application architecture: MVC pattern.*

To access the PROVA! web application, the user needs to create an account: the user credentials and the remote system connections configurations are the only information stored in the web application DB.

The primary function of the web application is delivered by the Experiment and Analysis Server (server-side of the application), which needs to communicate to the remote system to send commands to execute and receive back the data to visualize. After an authenticated user configures an ssh connection to the remote system, the Node.js server can use it to

send both synchronous and asynchronous commands. The synchronous commands get "static" information regarding the remote PROVA! installation, e.g., the methodTypes installed, and the user workspace, e.g., existing PROVA! projects. In contrast, the asynchronous ones are used to start commands which require a longer execution time, e.g., experiment execution.

The outputs of the synchronous commands appear directly into the view. However, for the asynchronous commands, we implemented a publish-subscribe pattern using socket.io [132]: after firing the remote command, the view only receives a confirmation that the command was started and joins a *room* (subscribe action), in the meantime, the server keeps listening to the command output forwarding the outputs as messages in the room (publish action) as soon as it receives it. The view gets the messages from the room and updates the page content. Even if the user closes the browser and comes back later to the web application, the first action done by the view is to subscribe again to its room to check if there is any previous command still executing.

The web application has seven view pages:

- **Profile**: CRUD operations for a user
- **Configurations**: CRUD operations for an ssh remote connections
- **Project**: CRUD operations for a project
- **Method**: CRUD operations for a method
- **Experiments**: Configure and submit an experiment
- **Visualization**: Visualize, reproduce or delete an existing experiment
- **RemoteShell**: Open a shell view on the remote system

The profile and configuration pages map to the user object stored in the web application persistence layer, while the other pages are specific to each connection configured by a user and populated with the information from the remote system. When the user gets access to the web application, he should first configure a remote connection entering the information in the configuration view page (see figure 8.5). The ssh module of the web application will try to establish a connection using the configuration provided and, in case of success, store the configuration for future usage.

After the user configures a proper connection to a remote system and connects to it, he can start working on his remote workspace directly from the UI, creating a project from the and then adding new methods using the correspondent views. Figure 8.6 shows an example of the method view.

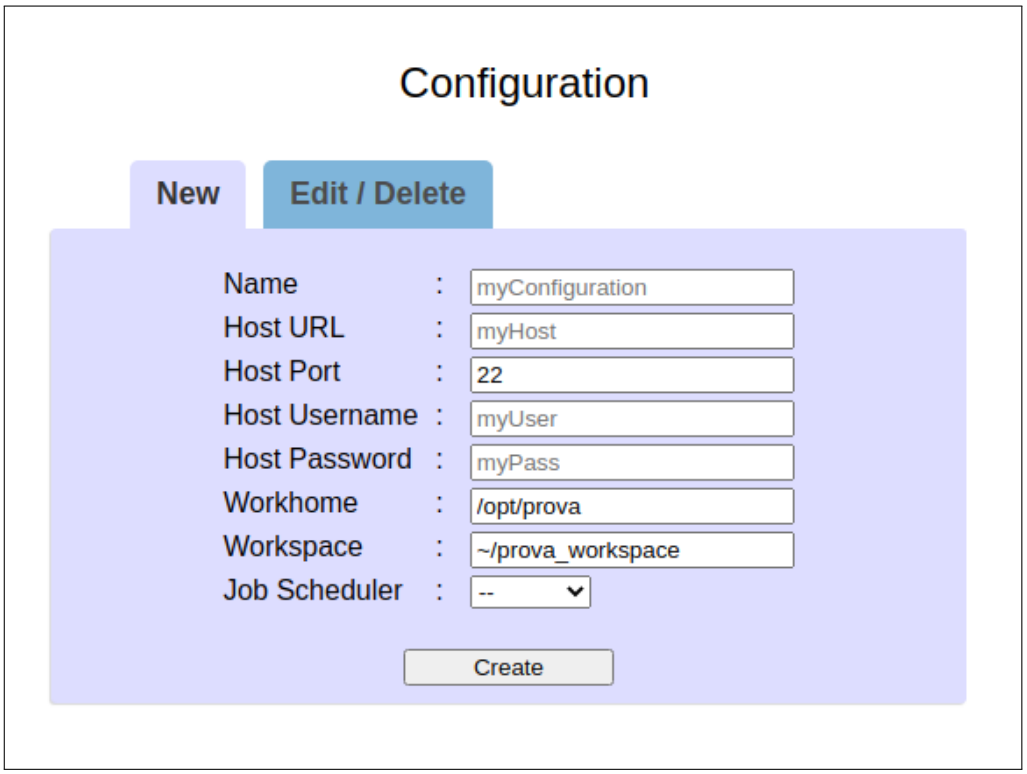



**Figure 8.5:** PROVA! *web UI: Configuration view*

There is information about the current systems in use on the right side, while on the left side, there is the main content of the page: the method. The user needs to specify the name for the project to which he wants to add the method and the method type. That information is based on the current connection and retrieved from the remote system.

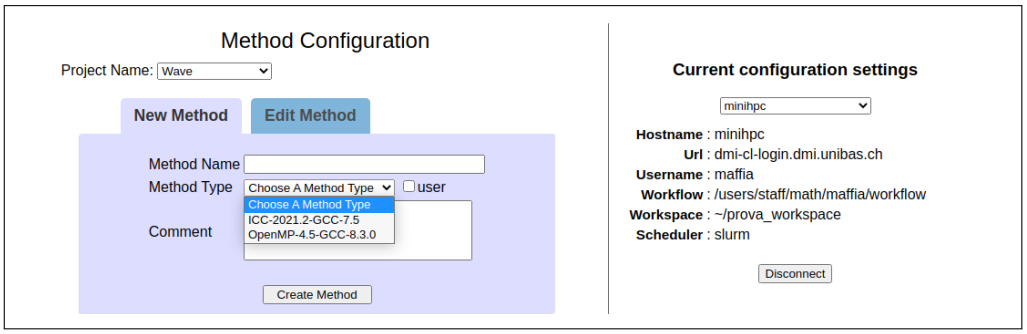
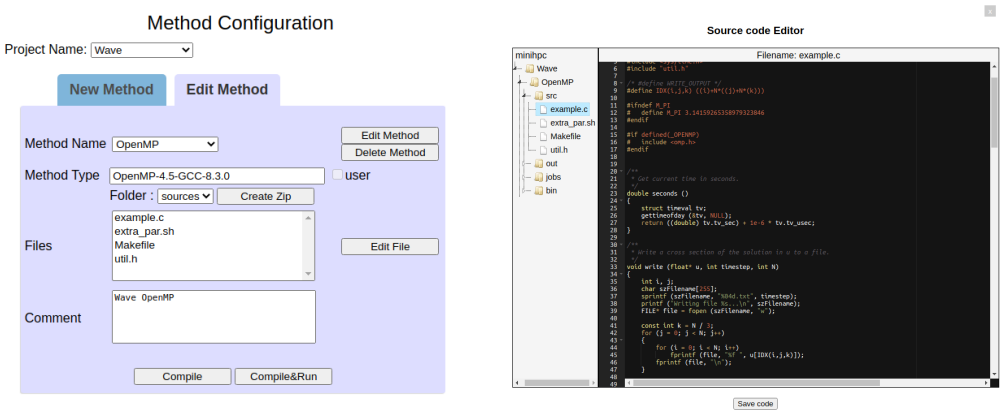



**Figure 8.6:** PROVA! *web UI: Method view for the method creation*

From the "Edit Method" tab in the method view (see figure 8.7a), the user has the details of the method he created and can edit the sources in the method directly in the web browser (see figure 8.7b). Instead, it is not

possible to change the methodType used during the method creation since the default files and configurations depend on it: in this case, the user would need to create a new method.




**(a)** *Method view for editing an existing method.* **(b)** *File editor in the method view.*

**Figure 8.7:** *Example of method settings and sources edit in the* PROVA! *web UI.*

In order to test the method he created, the user has "Compile" and "Compile&Run" buttons in the same method view (figure 8.7a) he can use. The web application will build the correspondent commands described in the previous section and send them for execution on the remote machine showing the results on the page in an output area.

After implementing and testing the methods, the user can proceed with the experiment configuration and execution. The different sections of the experiment view page are mapping out experiment taxonomy: Problem (project + parameters), Method (one or more methods implemented) and System (threads configuration for the resources to request on the remote system) as shown in figure 8.8. The experiment configuration includes the threads/processes pinning configuration (already discussed in [128]) and the job scheduler. The latter allows the user to define the settings for the scheduler specified during the remote connection creation:

- #**ofNodes**: The total number of nodes to request.
- **Configuration**: A string in ["Thread only", "MPI only", "Hybrid"] to manage the processes/threads configuration.
- **Multithread**: A "ON"/"OFF" string to activate/deactivate simultaneous multi-threading.

- **Partition**: A string representing the partition/queue name to use.
- **Walltime**: A string representing the time expected the job will need.
- **Memory**: A string representing amount of memory to request for the job.
- #**ofGPUs**: Total number of GPUs to request.

In case of direct execution of the experiment, the output will be prompted into the "Command Response" area while generated. Otherwise, in the case of using the job scheduler, PROVA! will show the IDs of the jobs created so that the user can check the job in execution from the interface to know when the experiment terminates.

The last step for the user is to check the experiment results from the visualization page. When selecting an experiment choosing a date and time, the experiment details show (figure 8.9a), and the user can configure the parameters/methods/threads combinations to visualize in the graph and the chosen metric. After selecting more graph options like the kind of values (min/max vs std.dev.) or what to show as Series or Category to use for the final plot, the graph view will look similar to figure 8.9b.

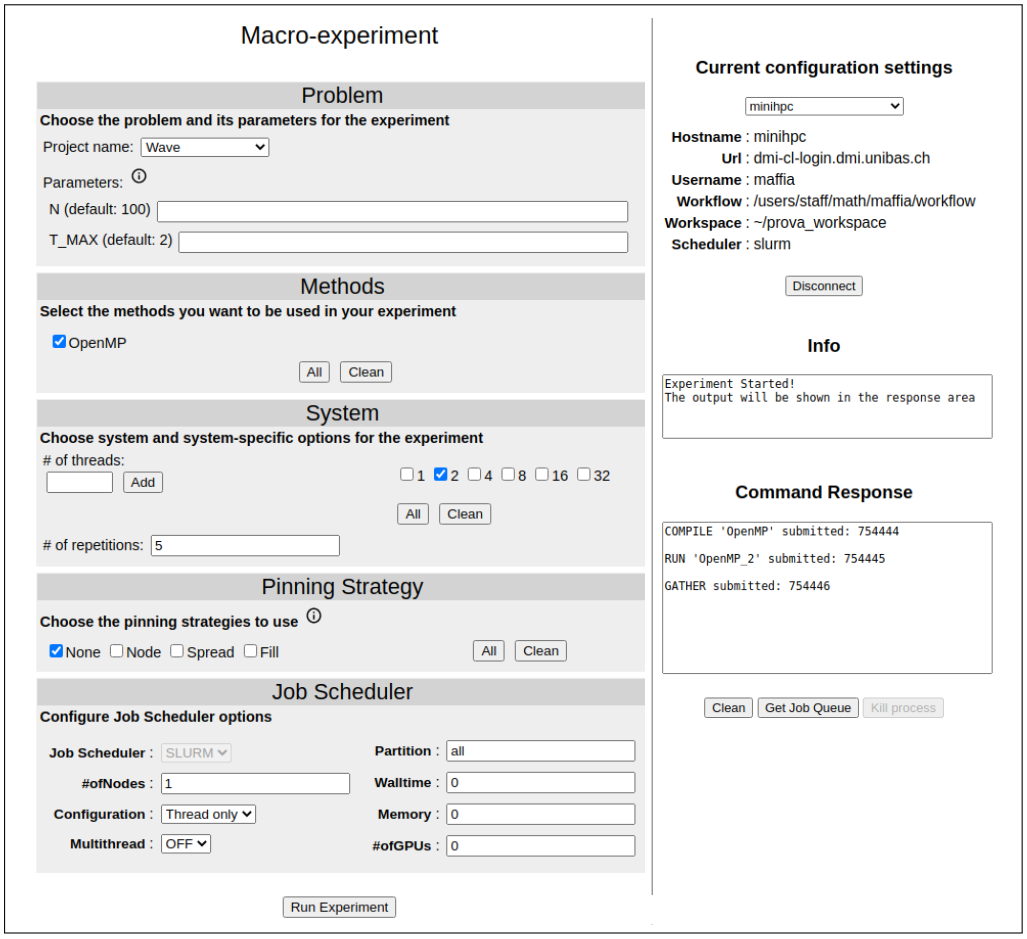



**Figure 8.8:** PROVA! *web UI: Example of an experiment's configuration*


**(a)** *Configure the experiment result data to show in the graph.* **(b)** *Configure the graph format.*

**Figure 8.9:** *Configuration and generation of the result graph from the* PROVA! *web UI.*

# Chapter 9

# PROVA! 2.0: Extensions

## 9.1 Feature enhancements

### 9.1.1 Job scheduler management

As shown in section 8.2, PROVA! can interface with different execution systems using a job scheduler. That is fundamental since the default way of interacting with an HPC system is through a job scheduler. PROVA! can pack the experiment to execute into a job script and submit it to one of the supported schedulers: Slurm, SGE/UGE and PBS. Since using only a few scheduler commands to submit, list, and kill jobs, it is trivial to add more schedulers.

Even though three job schedulers are supported, when using PROVA! with Slurm, it is possible to get some more advanced job scheduling behaviour for the experiment. Using job the dependency feature in Slurm, PROVA! splits the experiment steps for different methods and system configurations into multiple independent jobs, which improves (1) the system resource utilization since each job can customize the amount of resources requested and, because of the possibility of the jobs running in parallel, (2) the overall execution time (especially for experiments requesting a small subset of the complete system resources).

Following the experiment taxonomy of section 4.1, let us consider a problem $Exp$ consisting of two methods, $M_1$ and $M_2$ and two systems, $S_1$ and $S_2$. Let the problem also have two parameters, $P_x$ and $P_y$, which can take two values. Figure 9.1 shows how PROVA! would manage the job submission in such a situation.

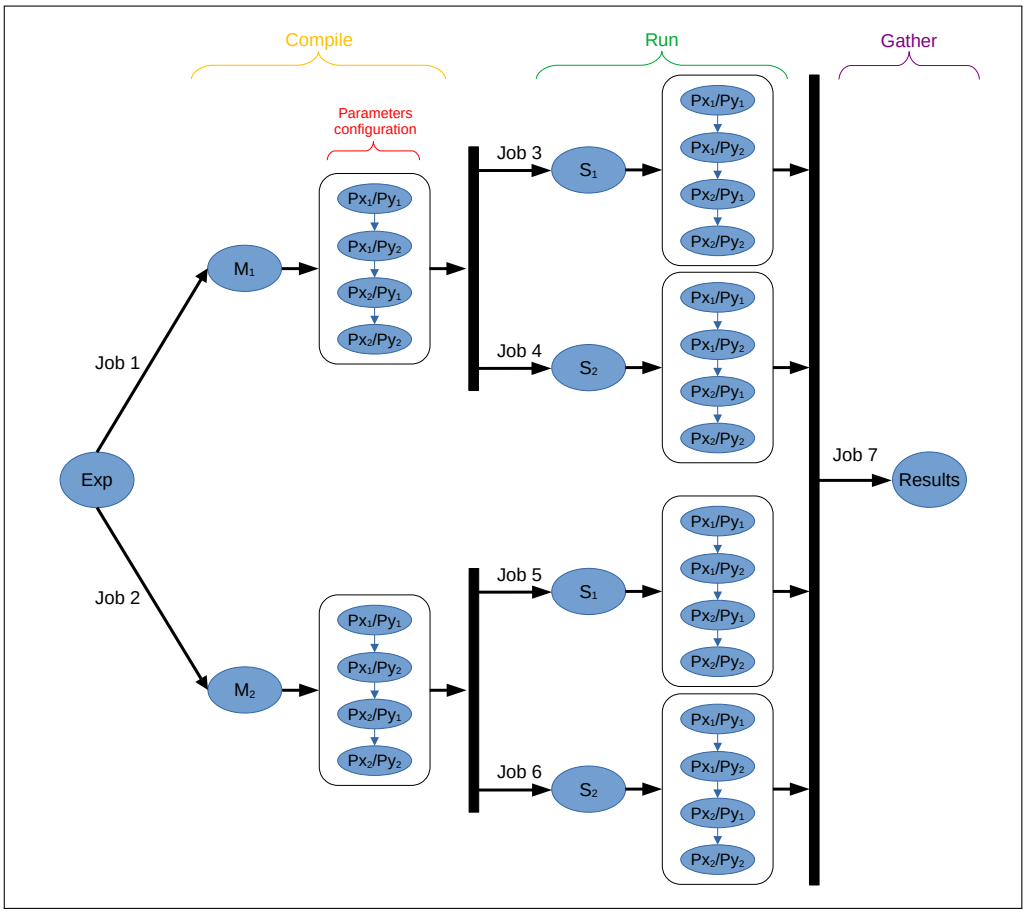



**Figure 9.1:** PROVA! *experiment job management in case of two methods ($M_1$, $M_2$), two parameters ($Px$, $Py$) with two possible values each, and two system configurations ($S_1$, $S_2$).*

The first two job scripts (*Job* 1 and *Job* 2) will contain the steps to compile each experiment's method using all the possible parameters combinations. Since the compilation may not require the same resources requested by the execution step of the experiment, these jobs can allocate a different amount of resources for each step, i.e., a lower amount in case of the compilation, which optimize the system usage.

*Job* 1, compiling the method $M_1$, is a dependency for *Job* 3 and *Job* 4, which will execute the same method $M_1$ only after *Job* 1 has successfully terminated. Also, for the run phase, PROVA! can set up different jobs based on the system configuration $S_1$ or $S_2$ and submit those jobs in parallel: if the system configurations require a different amount of resources, the experiment will have a better resource allocation and, most likely, a shorter execution time. Furthermore, for the method $M_2$, PROVA! will generate two

jobs for the run phase (*Job* 5 and *Job* 6) which depend on the job executing the compilation phase (*Job* 2).

Finally, the jobs submitted for the run step are considered a dependency for the *Job* 7, which will gather all the logs/outputs from the different execution and create a "results" file. To avoid an error occurring in one of the experiment runs, the dependencies defined for the last job (*Job* 7) is different from the rest: it get satisfied at the end of all the previous jobs, like in the previous case, but even if only one of the jobs it depends on end successfully.

### 9.1.2 Experiment reproduction

The way PROVA! manages the experiment and their reproduction has improved. Even though experiment reproduction has always been the main focus of PROVA!, in the first version of PROVA!, a scientist who wanted to reproduce a PROVA! experiment had to get the information contained in the PROVA! experiment descriptor and use them to reconfigure the experiment manually.

In the new version, it is possible to repeat (see section 4.2) an experiment with a single command and compare the reproduced experiments against the original and among them. PROVA! creates a bi-directional reference between the "master" and the "reproduced" experiment(s) to simplify their comparison through the experiment visualizer, as shown in figure 9.2.

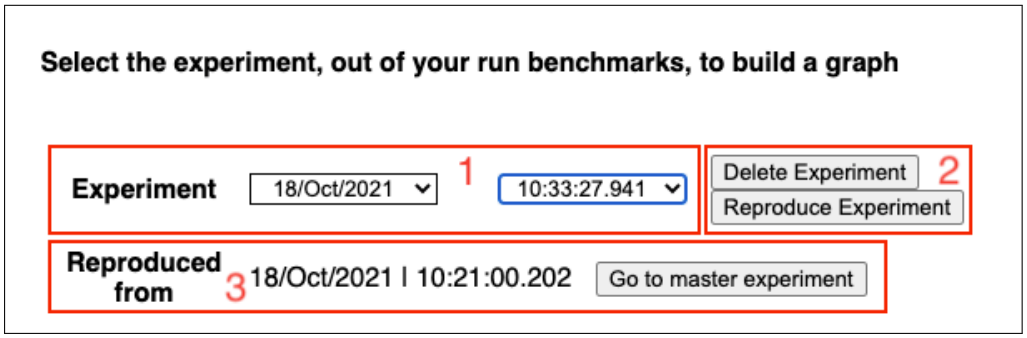



**Figure 9.2:** PROVA! *web UI: Experiment selection in the visualization page*

Figure 9.2 shows the different parts of the experiment selection in the PROVA! UI: after selecting an experiment based on the date and time of its execution (box 1), it is possible to delete or reproduce it using some action buttons (box 2). For reproduced experiments, the information appears on the page, and an action button can help the user switch to the master experiment (box 3).

When the user selects the "Reproduce Experiment" action, PROVA! clones the experiment structure (see figure 8.3) into a new experiment and executes it applying the same workflow commands used for the original experiment, including, in case of scheduled executions, the submission of the same job scripts used in the original execution flow. At the end of the execution, PROVA! updates the dependencies among the reproduced experiments, which can be then used in the visualization page to build a new performance graph.

### 9.1.3 Experiment visualization and graph builder

The enhanced version of the experiment visualizer allows the scientist to specify the experiment characteristics like parameters, methods, system (see figure 8.9a), and other possible executions of the same experiment allowing comparison as discussed in the previous section.

The experiment elements a user can choose for the graph builder are:

- **Parameters**: List of parameters' configurations run against the experiment.
- **Method**: List of methods executed during the experiment.
- **Threads**: List of processes number combinations requested by the experiment.
- **Reproduced experiments**: List of repetitions for an experiment (if present).

Since the graph builder will create a 2D histogram graph, only 2 of those elements can vary. For all the others, the user must select a specific fixed value. It is then possible to choose which "variable" element will be used as "category" (different groups for the X-axes) and which as 'series" (bars for each category on the X-axes) in the generated graph. Figure 9.3 shows how to configure the graph builder to compare different replication of the experiment; selecting a reproduced experiment (box 1), the graph builder will show the "Reproduced experiments" as a selectable entry for Series or Category (box 2).

Figure 9.4 shows an example of a graph generated using *reproduced experiments* as **series** and *methods* as **category**: at the top, we find the project parameters and their values (fixed elements) followed by the selected experiments (series), and at the bottom, on the X axes, the implemented methods (categories).

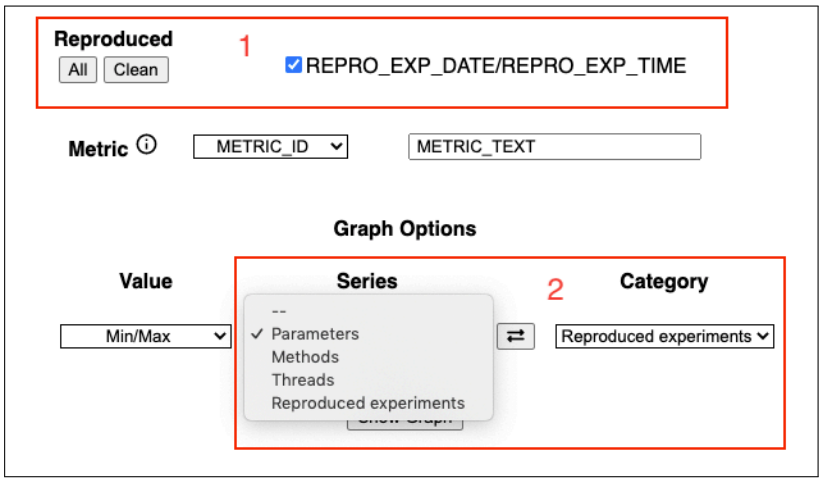



**Figure 9.3:** PROVA! *web UI: Reproduced experiment in the visualization page*

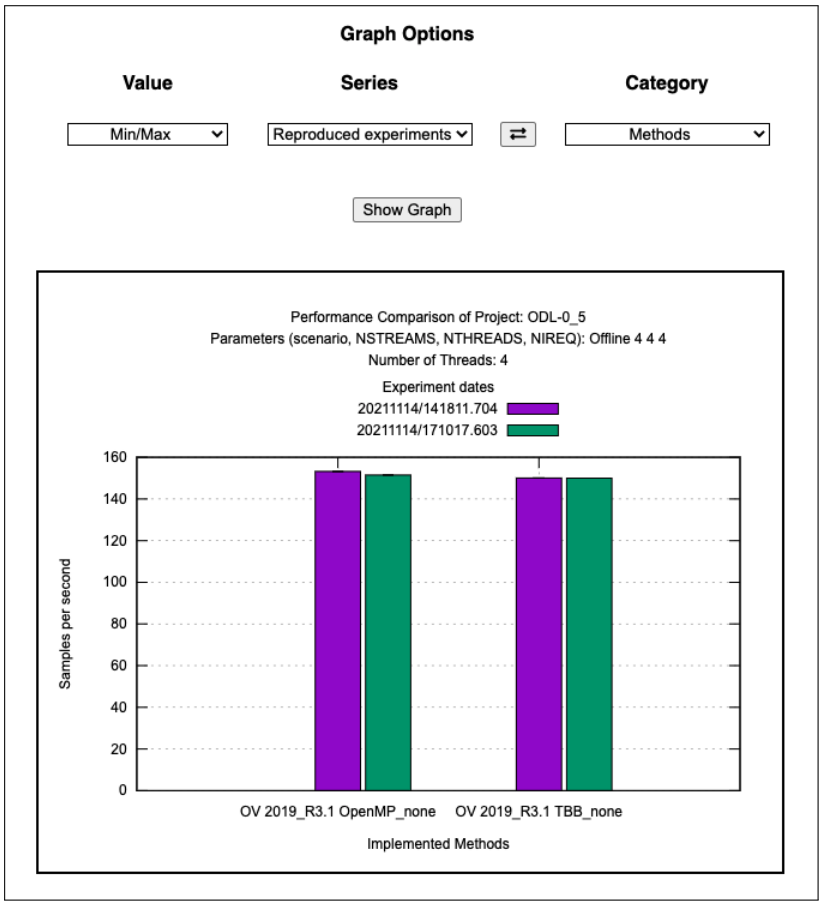



**Figure 9.4:** PROVA! *web UI: Example of graph comparing different reproduction of an experiment*

## 9.2 Containers support

The way PROVA! provide to handle the experiment software complexity (see section 7.1) is through the definition of a method type. Initially, a method type in PROVA! could manage the software needed by the experiment only using the Linux environmental modules leveraging EasyBuild (see section 7.1.1).

As discussed in section 7.1.2, there are quite some benefits in using containers when dealing with software complexity which convinced us to add container support in PROVA!. In the meantime, EasyBuild integration is still available, and, if needed, it is possible to use it in combination with containers (for example, if the container software requires to load a specific module for it).

In the methodType descriptor, we need a container section that includes the following information:

- **executable**: Container client executable (ex. docker, singularity).
- **cmd**: Command to pass to the container client executable (ex. run, exec).
- **runtime**: Option needed by a specific runtime (ex. "--nv/--gpus" to enable GPU usage in "Singularity/Docker").
- **options**: Other options to pass to the container command.
- **url**: Remote container location (ex. "docker://ubuntu" to use public docker image of Ubuntu OS in Singularity).
- **imgdir**: Path where to download the container image (Singularity only).
- **img**: Name of the downloaded container image (Singularity only).

Based on those values, PROVA! exports the environment variables used by *compile* and *run* scripts during the experiment compilation and execution phases.

For instance, PROVA! constructs and exports the *CONTAINER_CMD* environment variable (see listing 9.1), which a user can prepend to the command running the experiment and use other container variables like *IMAGE_URL* and *IMAGE_PATH* to pull the container before using it (see listing 9.2).

Listing 9.2 shows how to use the *CONTAINER_CMD* environment variable, constructed and exported by PROVA! (see listing 9.1), to adapt the command used to run the experiment and other container variables like

```
...
CONTAINER_EXE=`echo ${container_info} | jq -r '.executable'`
CONTAINER_CMD=`echo ${container_info} | jq -r '.cmd'`
CONTAINER_RTM=`echo ${container_info} | jq -r '.runtime'`
CONTAINER_OPTS=`echo ${container_info} | jq -r '.options'`
export CONTAINER_CMD="${CONTAINER_EXE} ${CONTAINER_CMD} \
${CONTAINER_RTM} ${CONTAINER_OPTS}"
...
```

**Listing 9.1:** *CONTAINER_CMD variable constructed by the* PROVA! *compile script.*

*IMAGE_URL* and *IMAGE_PATH* to pull the container before the command execution.

Since primarily targeting HPC systems, we focus on Singularity container technology, the most used in the field. Nevertheless, PROVA! container support has been developed to be generic and used directly (or with minor adaptations) with other container technologies like Docker or Podman (see section 7.1.2).

```
...
if [ ! -f ${IMAGE_PATH} ]; then
  echo "Singularity image not found, pulling it from library"
  if [ ! -d ${IMAGE_PATH%/*} ]; then
    mkdir -p ${IMAGE_PATH%/*}
  fi
  singularity pull ${IMAGE_PATH} ${IMAGE_URL} > /dev/null
  echo "singularity pull completed"
fi
${CONTAINER_CMD} ${IMAGE_PATH} mycommand
...
```

**Listing 9.2:** *An example of using Singularity command in a* PROVA! *run script.*

## 9.3 Driver mode

An experiment expressed as a ⟨Problem, Method, System⟩ 3-tuple can be implemented as a micro-/macro-experiment in PROVA! (Section 4.1). However, predefined experiments like a benchmark can be repetitive in terms of configuration and parameters to set. In this situation, it is helpful to represent the experiment as a "driver".

A driver in PROVA! is a predefined project type (similar to a template) that allows users to have a base working experiment that can be customized and used as a starting point for possible experiment variations. It includes

the default experiment configurations, the output/metrics generated by the experiment, and the scripts needed to manage the whole experiment workflow. Once created, the user can share the PROVA! driver with others having the same needs.

The driver structure is shown in figure 9.5: the mandatory elements for a driver are the descriptor and the script used to run the experiment, the other scripts are optional, and PROVA! will not check for their presence. For example, by default, PROVA! copies the driver files to the installation folder and possibly run the driver install script. If that driver does not require any particular installation step, the install script can be omitted.

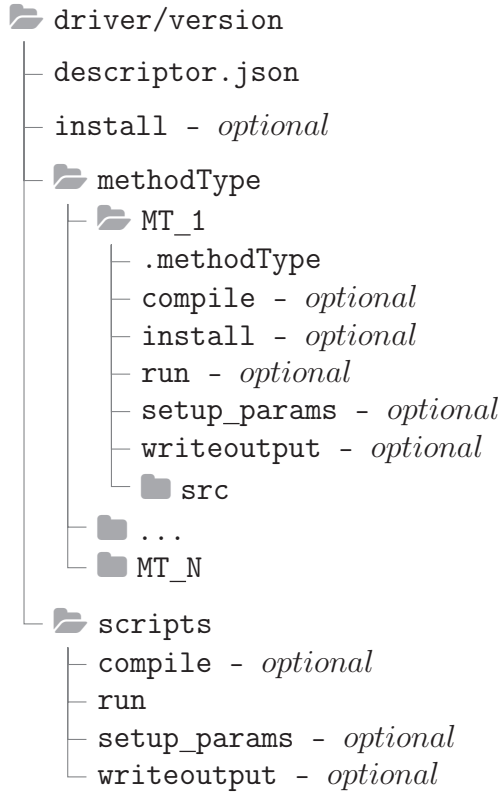



**Figure 9.5:** *Structure of a driver folder in* PROVA!

A driver descriptor includes all the information related to a driver. This JSON file extends the PROVA! project descriptor presented in section 8.2.1.

Besides the fields inherited from the project, i.e., name, version, parameters, threads, metrics and lineselector, a descriptor may contain more driver-specific information needed for handling the experiment, e.g., it can be helpful to define the different possible datasets to be used or also the code repository URL.

After the self-explanatory name and version in the driver descriptor, we have the parameters block, which includes all the parameters used as input or as configuration for the experiment and the details for each of them.

The parameters are, obviously, driver-specific, but their structure usually includes:

- **default**: A default value for the parameter.
- **values**: A list of predefined values for the parameter (optional).
- **mode**: A string defining how to use the parameter (optional). For example, use "export" to specify to export the parameter as an environment variable.
- **type**: A string defining the parameter's data type (optional).

The only parameter's information used directly by PROVA! are the list of possible **values** and the **default**: the first suggests to the user the possible selectable values in the PROVA! UI for a specific parameter, the second defines the default values for the parameter in case the user does not specify them. Even though the PROVA! workflow is not directly using other parameters' information from the parameter's block, it will read those values and make them available to the driver-specific experiment scripts.

The number of threads represents a particular parameter since it helps request the proper amount of resources for the experiment execution, that is why PROVA! treats it separately, and the driver descriptor defines a **default** value for it. Moreover, PROVA! will always expect the *Threads* parameter to be defined.

The driver descriptor continues with another critical driver-specific element: the *metrics* field. The way a metric gets extracted from the output depends only on the driver, and the structure of a metric cannot, as well, be generalized and need to be custom to the specific driver.

Finally, the *lineselector* field defines the string used to filter the output, telling PROVA! which output lines contain the results to parse.

Since the driver can be a collection of multiple applications (like in the case of a benchmark suite), all the software needed to run each driver application or different implementation of an application is managed by defining a **methodType**. Like for a usual PROVA! methodType (see section 8.2.1), it includes the installation script and the experiment workflow scripts specific to a software/hardware combination: setup_params, compile, run, but also a **writeoutput** script which makes use of the *lineselector* and *metrics* information to filter the "raw" logs and generate the final output file used during the gather step to produce the result data.

When a methodType does not require special steps for managing the experiment workflow, the script in the *methodType* path is not needed, and PROVA! will use the ones present in the *scripts* folder of the main driver path (see figure 9.5). In a driver, the methodType contains a basic implementation of the driver application managed by it, which PROVA! will copy inside the source folder of a method during its creation as the starting point for the final method implementation.

To create a project from an existing driver, the user will define the project type as "driver" and select the name and version for the driver he wants to use among the ones installed on the remote system, as shown in figure 9.6 (box 1).

At this point, based on the driver selection, the information stored in the driver descriptor is loaded into the PROVA! UI as shown in figure 9.6: they can be either configured as for the default parameters (box 2) or customized as for the user-defined parameters (box 3) or the "metric/output selector" (box 4).

The creation of a project of driver type will also affect the method creation, and the PROVA! method view in the web UI will only show the methodTypes available for the driver selected in the project definition (see figure 9.7)

Knowing all those configurations and information, PROVA! will distinguish its actions based on the project type following either the **base** or the **driver** experiment workflow.

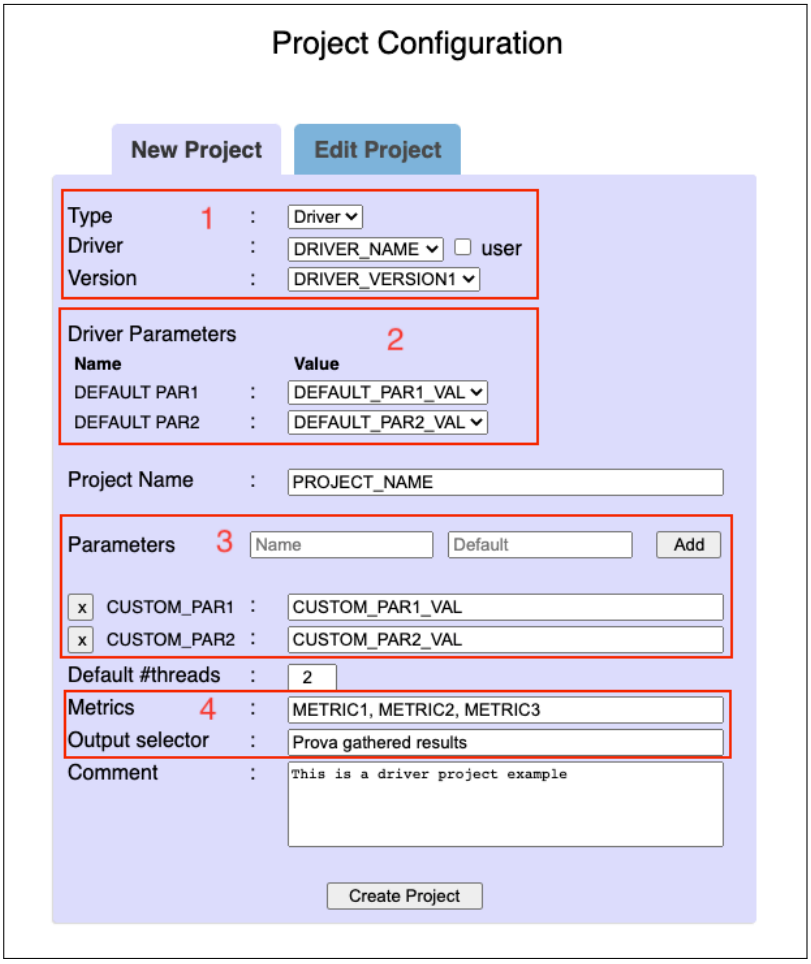



**Figure 9.6:** PROVA! *web UI: Driver mode configuration for a new project*

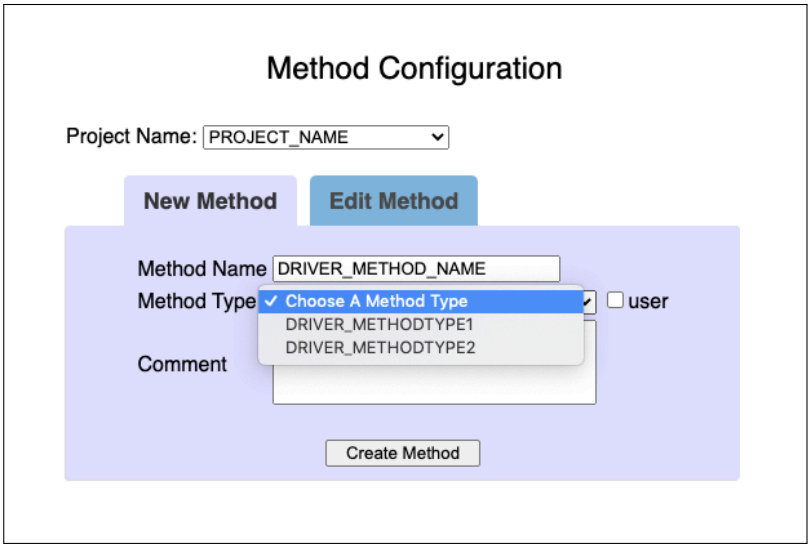



**Figure 9.7:** PROVA! *web UI: Driver method's type for a new method*

# Chapter 10

# PROVA! as Deep Learning Benchmark Driver: MLPerf Inference Example

## 10.1 Driver design

We needed to both adapt our framework add the driver mode support (see section 9.3) and define the driver in a "generic" way so that the same approach can be reused for other drivers in the future. The decision for our first PROVA! driver fell on the MLPerf Inference benchmark [44] for three main reasons: (1) Involving not only the code to benchmark but also the LoadGen component makes it more interesting from the "integration" perspective, (2) this benchmark provided, beyond the reference implementations, some "helper" scripts to run it which made it easier to start with and (3) inference requires less powerful resources and time to execute, this allowed us to test the new approach quickly.

We thought about different approaches with an increasing degree of integration with the PROVA! components: project, method (methodType) and system.

#### Approach 1: MLPerf as a project

In this approach, the project is generic for an MLPerf benchmark which could be any of the available suites, the method will represent the chosen

suite and the task, framework, device, and other settings are all considered parameters to configure.

Here is an example of the PROVA! components mapping:

- Project: MLPerf
- Method: MLPerf Inference
- MethodType: MLPerf Inference
- System: GPU workstation
- Parameters
  - Model: SSD-Mobilenet v1
  - Framework: Tensorflow
  - Device: GPU
  - Dataset: coco2017
  - Scenario: SingleStream

However, this mode is an evident "misuse" of PROVA! since we are not mapping the experiments' elements (Project, Method, System) to PROVA! but just adding all to a script and running the whole experiment. The other problem is to have a "golden" methodType that should be able to manage any possible combination of ML tasks, frameworks and devices, which would probably not be feasible, at least not if we want to have a minimum of flexibility.

### Approach 2: MLPerf Inference as project

In the second approach, we consider configuring a project for a specific benchmark suite and moving one element of the benchmark experiment into PROVA!, i.e., the ML task.

A possible mapping of the PROVA! components would look like this:

- Project: MLPerf Inference
- Method: Object Detection Light
- MethodType: Object Detection Light
- System: GPU workstation
- Model: SSD-Mobilenet v1
- Dataset: coco2017
- Parameters

  - Framework: Tensorflow
  - Device: GPU
  - Scenario: SingleStream

Since the ML task to execute is mapped as a **method**, both the *model* and the *dataset* are no longer "open" parameters and can be inferred from the method. Nevertheless, even in this second approach, we have an issue regarding a MethodType managing any possible ML framework to run, which is not quite handy and still does not have a good mapping with the PROVA! experiment elements.

#### Approach 3: MLPerf Inference task as project

In the last approach, we configure a PROVA! project to manage single ML tasks which represent the problem we want to solve, like in a usual PROVA! project, and fixes the *model* and *dataset* at the project level. The method is a single ML framework representing one possible solution to the problem, like the method in the experiment taxonomy, and, in the meantime, it fixes the MethodType, which can include a single software with a specific version configured for a particular device.

The example of PROVA! components mapping using the final driver approach would be:

- Project: Object Detection Light
- Method: Tensorflow-2.4.0
- MethodType: Tensorflow-2.4.0
- System: GPU workstation
- Model: SSD-Mobilenet v1
- Dataset: coco2017
- Framework: Tensorflow
- Device: GPU
- Parameters
  - Scenario: SingleStream

The only parameter left is the **scenario** we will configure during our experiment. In the case of the MLPerf Inference benchmark, other parameters for the ML tasks are in a couple of configuration files: $mlperf.conf$ and $user.conf$. Those files should contain values that produce the best performance and, for this reason, may not need to be treated as parameters.

That is probably not the same during the development phase when a user may need to add more custom variables on top of the parameters defined as default in the driver. Section 10.3 will show an example of how to use the driver during the development of an experiment.

## 10.2 Driver Configuration

### 10.2.1 Driver descriptor

After defining the structure of the PROVA! driver for the MLPerf Inference benchmark, we start configuring the components we discussed in section 9.3, which the driver needs. Apart from the general elements, i.e. name, version, parameters, threads, metrics and lineselector, the MLPerf Inference driver descriptor defines as well some the other specific elements like:

- repo: URL to the git repo for the driver version
- results: URL to the results git repo for the driver version.
- categories: List of ML tasks for the driver version.
- datasets: List of datasets allowed for the driver version.
- container: Details for the benchmark suite container used for submissions checks.

Except for the container element, which looks like the one of a methodType descriptor (see section 9.2), the other specific elements and the parameters and metrics have a custom structure based on the driver functioning. PROVA! uses *repo* and *results* fields to download the correspondent git repositories during the driver installation. Instead, each category of the categories object includes information about the model and the dataset usable by the benchmark application task.

Let us consider the **Object Detection lightweight** task: the task-related entries in the driver descriptor are shown in listing 10.1.

The *datasets* field in the driver descriptor contains the details of each dataset available for the driver, like the name, the version, the URL to download it (when available), the path under the driver folder where to store it and the script to check the accuracy of the results using that dataset. In the case of the MLPerf Inference driver, the only predefined parameter is the **scenario** which, for version v0.5 of the benchmark, could be one of SingleStream, MultiStream, Server or Offline, but if needed, the user will be able to add more parameters.

```
...
"categories": {
  "OD_L": {
    "area": "vision",
    "name": "Object Detection Light",
    "model": "ssd-mobilenet",
    "dataset": "coco",
    "dataver": "2017",
    "path": "v0.5/classification_and_detection"
  },
  ...
},
"datasets": {
  "coco": {
    "2017": {
      "path": "data",
      "downloadable": "true",
      "url": ["http://images.cocodataset.org/zips/val2017.zip",
        "http://images.cocodataset.org/annotations/
           annotations_trainval2017.zip"]
    },
    "acc_chk": "/inference-r0.5/v0.5/classification_and_detection/
       tools/accuracy-coco.py"
  },
  ...
},
"parameters": {
  "scenario": {
    "default": "SingleStream",
    "values": ["SingleStream", "MultiStream", "Server", "Offline"]
  }
},
"metrics": {
  "90.0": {
    "description": "90.0",
    "type": "f",
    "mult": "1",
    "dec": "0",
    "text": "90.0",
    "lineselector": ["90.00 percentile latency (ns)"]
  },
  "qps": {
    "description": "Throughput",
    "type": "f",
    "mult": "1",
    "dec": "2",
    "text": "qps",
    "lineselector": ["QPS w/o loadgen overhead",
      "Samples per query",
      "Samples per second",
      "Scheduled samples per second"]
  },
...
```

**Listing 10.1:** *MLPerf Inference driver descriptor in* PROVA!*: Object Detection Light section.*

The last part of the descriptor defines the possible metrics. The metric can have the following fields:

- Description: String representing the metric description.
- Type: Data type for the metric.
- Mult: Multiplier used to adapt the precision of the metric.
- Dec: Amount of decimal digits to show for the metric.
- Text: Default text to show in the performance graph for the metric.
- Lineselector: Array of strings to filter the output and retrieve the metric.

Every workload have multiple metrics generated by the LoadGen logger, and the user can gather all of them using PROVA!, but, depending on the scenario, the metrics required by the benchmarks are changing (see table 5.2). In the case of the **Single-Stream** scenario, the metric needed is the 90%-ile latency which, in the results file, is always identified with the same name that PROVA! can use as *lineselector*. While the *QPS* metric (throughput) has different meanings based on the scenario and the strings to identify it in the results file will change. In this situation, PROVA! will look for any possible strings specified as lineselector to gather the proper value (see listing 10.1) and store it.

### 10.2.2 Driver execution scripts

The information stored in the driver description will be used to run the PROVA! experiment workflow. The main steps of the experiment workflow for the driver are the same as for a standard PROVA! experiment: **Compile**, **Run**, **Gather output** (see section 8.2.1) with a few minor adjustments.

As described in section 9.3, the methodTypes used in a driver are local to the driver itself, i.e., stored in the driver folder, and not usable by method created in generic PROVA! project but only in projects declared as implementing the specific driver. Based on the approach chosen for the driver implementation, a methodType for the MLPerf Inference benchmark will be specifically managing a certain software and device, and it will be composed of a descriptor and the scripts to manage it. Let us consider a methodType for the OpenVINO framework (see section 3.3). Its PROVA! methodType descriptor is shown in listing 10.2.

The descriptor contains some general information like the framework name, the device type, the list of the driver categories the methodType can

```
"name":"OpenVINO-2019.R3.1_src_c_omp",
"categories": ["OD_H", "OD_L", "IC_H", "IC_L"],
"device": "cpu",
"framework": "openvino",
"eb_modules": [],
"container": {
  "executable": "singularity",
  "cmd": "exec",
  "runtime": "",
  "options": "--contain",
  "url": "https://github.com/provarepro/mlperf_inference/releases/
     download/0.0.1/provarepro-mlperf_inference.v0.5-OpenVINO-2019
     _R3.1_src_c_omp-py36-gcc75-ubuntu18.sif",
  "img": "mlperf_inference-v0.5-OpenVINO-2019_R3.1_src_c_omp-py36-
     gcc75-ubuntu18.sif",
  "imgdir": "container"
},
"version":"1.0",
"comment":"OpenVINO-2019.R3_1 (compiled from source with OpenMP
    support) with MLPerf Loadgen v0.5"
```

**Listing 10.2:** *Example of methodType descriptor for OpenVINO framework.*

run, and the Linux container (using Singularity technology in the example) specific to the ML framework. The workflow scripts will use all those information to carry on the experiment.

### Compile

In this first step, PROVA! uses the **setup_params** to read the experiment parameters and store them into a configuration file which will be exported prior to the actual compilation of the benchmark code.

PROVA! configures the parameters in several steps: the parameters read in one step can be used to build the ones of the following steps. First, it reads the "fixed" experiment characteristics from the project and methodType descriptors. Second, it reads the values assigned by the user to the default and custom project parameters for the experiment execution, and, finally, it copies the MLPerf configuration files defined by the benchmark (see figure 10.1).

### Run

The run script is specific to a methodType and may require different parameters apart from the ones configured in the previous step. In the case of OpenVINO, we use the execution command documented by Intel in the MLPerf Inference benchmark submission to define the required parameters:

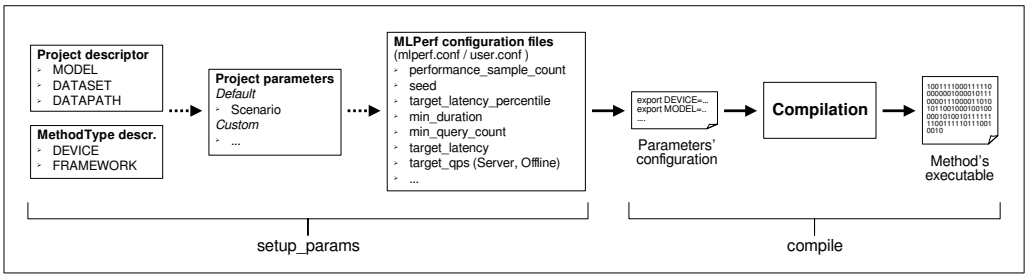



**Figure 10.1:** *MLPerf* PROVA! *driver: Compilation steps.*

```
$ ov_mlperf.exe \
    --scenario SingleStream \
    --mode Performance \
    --mlperf_conf_filename mlperf.conf \
    --user_conf_filename user.conf \
    --total_sample_count 50000 \
    --data_path dataset-coco-2017-val \
    --model_path ssd-mobilenet_int8.xml \
    --model_name ssd-mobilenet \
    --batch_size 1 \
    --nwarmup_iters 50 \
    --dataset coco \
    --device CPU \
    --nireq 1 \
    --nthreads 8 \
    --nstreams 4
```

**Listing 10.3:** *OpenVINO execution command for the Object Detection (lightweight) task[135].*

listing 10.3 shows the command used for the Object Detection (lightweight) benchmark task (model: SSD-MobileNets-v1 [133][134]).

The missing parameters get exported directly as part of the methodType run script (see figure 10.2), applying either a default value defined in the same script or a custom value. The user can add any method-specific parameters as a custom project parameter, and the setup_params script will export it, overwriting the default value.

To develop the driver mode, we add a change to the run script to preprocess the outputs generated before going to the final step of the workflow. The default PROVA! **gather_output** script expects the results file to have specific formatting. For this reason, the **run** script of a driver methodType needs to run another script to adjust the raw output generated by the MLPerf benchmark. A **writeoutput** script gets executed after the experiment execution and uses the line selector for the desired metrics, as mentioned in the previous section (see section 10.2.1). Finally, it writes the values fetched to an output file, ensuring compatibility with the input

format used by the gather script (see figure 10.2).

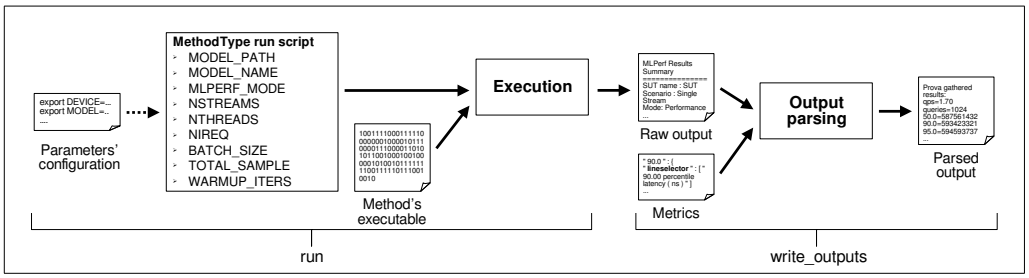



**Figure 10.2:** *MLPerf* PROVA! *driver: Execution steps.*

### Gather output

The gather output step used in the case of a driver project is the same as for a traditional PROVA! project. The script needs the information from the project parameter configuration to understand the methods executed during the experiment. It then looks for the experiment results in all the single outputs and merges the results in a JSON file. This final results file will be accessed by PROVA! when the user wants to generate a performance graph out of the experiment (see figure 10.3).

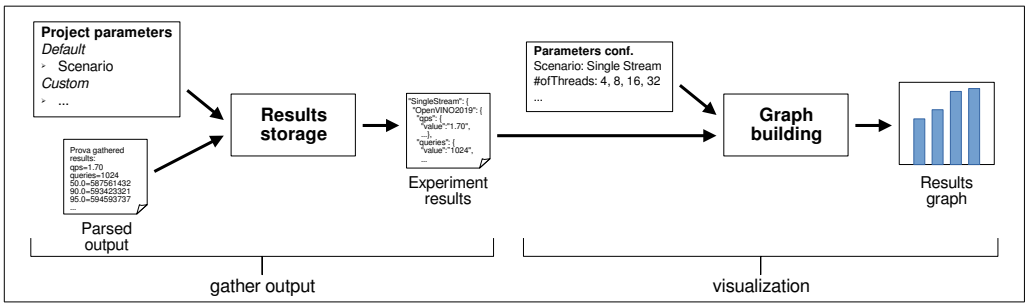



**Figure 10.3:** *MLPerf* PROVA! *driver: Results presentation steps.*

## 10.3 Driver Usage

As for the basic project, PROVA! can also carry on complex experiments (macro-experiments) in a driver project setup. In the case of a driver, some common parameters are added by default when creating a new project and can be integrated with more custom parameters based on the needs. While the final goal of a "driver" project is to compare different approaches on different systems, during the implementation of the code, it is helpful

to vary more method-specific parameters to tune them. Possible usage of PROVA! is to set up a separate driver project with "compatible" methods only and specify the parameters to tune as project parameters. After the tuning phase, the user can set up another driver project keeping out the "fixed" method-specific parameters, which he previously tuned, and use just driver-specific parameters. This way, the user has great flexibility since he can develop a method in an isolated way and later compare it with others based on the driver characteristics, e.g., the task scenario for MLPerf Inference benchmark.

As with the other PROVA! commands, the driver mode can be operated directly through the CLI or using the PROVA! Experiment and Analysis Server presented in Section 8.2. A user logged into the web server can connect to a remote system and start creating a project: when a driver is available, the user can select it in the interface and enter the base configurations and custom parameters.

Based on the driver used for the project, the interface will show the methodTypes available for the method creation (see figure 10.4).

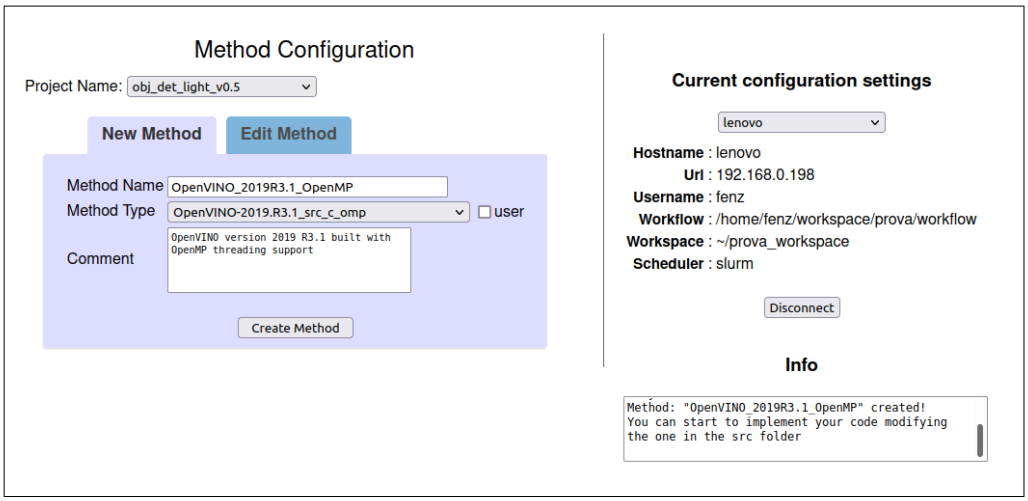



**Figure 10.4:** PROVA! *web UI: Method creation example for the OpenVINO framework.*

PROVA! will create the method from a base template included in the methodType, which, in the case of the OpenVINO methodType example, it is represented by the code Intel provided for the MLPerf Inference benchmark: this code can be modified or just used as-is to replicate the Intel submission. The same approach can be followed to provide the methodTypes representing all the implementations available in the MLPerf benchmark results.

As an example, let us consider a user who wants to evaluate the accuracy achievable by the ML model used for a benchmark task: since by default, MLPerf code runs in *Performance* mode, the user needs to overwrite the MLPERF_MODE parameter (see figure 10.2) by adding it as a project parameter and then set its value to *Accuracy*. Furthermore, to run an OpenVINO experiment, the user needs to convert the model to the OpenVINO Intermediate Representation (IR) composed of an XML and a binary file. The OpenVINO methodType template already has a valid model in the OpenVINO format. However, a user may want to experiment with how the accuracy changes with different converted models: the user can add the models to the default path (MODEL_PATH) and set the correspondent MODEL_NAME values after configuring it as a project parameter.

Figure 10.5 shows the experiment configuration matching our OpenVINO example: the experiment assesses the accuracy of the Object Detection (lightweight) benchmark task using two model versions, specified in the MODEL_NAME parameter, running a Single-Stream scenario. We generated the models by converting the official SSD-Mobilenet Tensorflow model to OpenVINO format using INT8 precision mixed with either single (FP32) or half (FP16) floating-point precision (for the operations not supporting the INT8 precision).

The experiment execution (see section 9.1.1) and the results visualization (see section 9.1.3) remain unchanged. The final performance graph generated using the PROVA! UI shows the accuracy achieved by the two models used, which, in this case, is relatively stable (see figure 10.6).

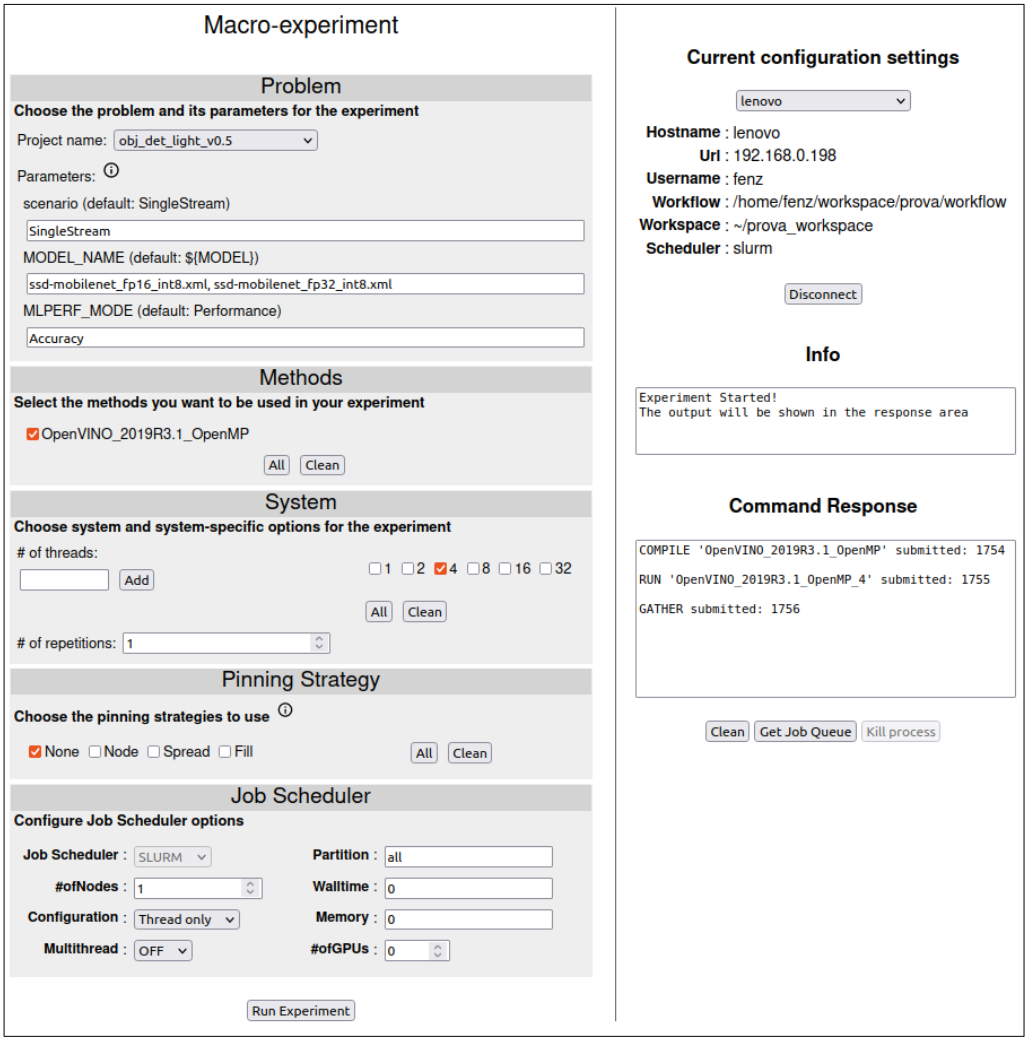
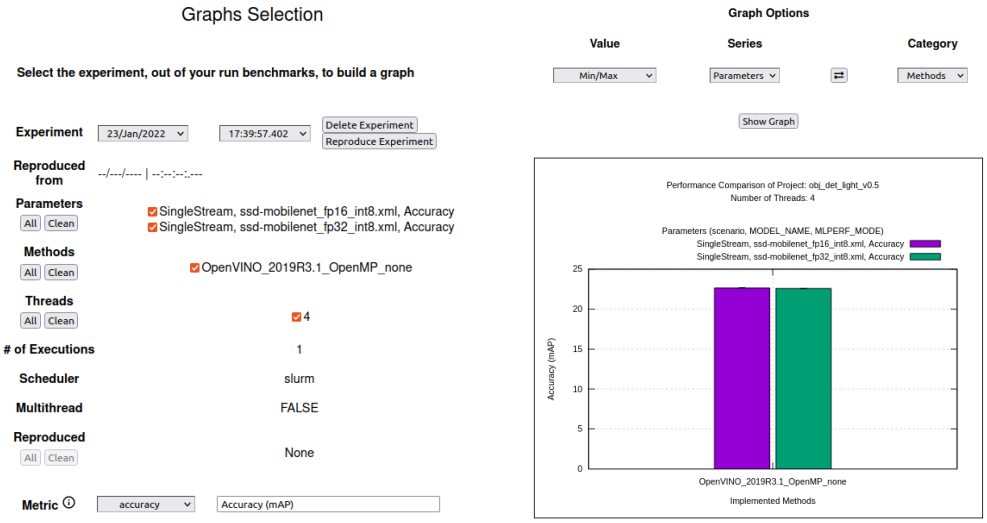



**Figure 10.5:** PROVA! *web UI: Experiment configuration and execution example for the object detection (lightweight) task running OpenVINO framework.*




**(a)** PROVA! *web UI: Results graph configuration for the OpenVINO Accuracy example.*

**(b)** PROVA! *web UI: Performance graph for the OpenVINO Accuracy example.*

**Figure 10.6:** PROVA! *web UI: Configuration and generation of a performance graph showing accuracy for the MLPerf Object Detection (lightweight) benchmarks task using OpenVINO framework.*

# Part IV

# Measurements and Results

# Chapter 11

# Experimental Testbeds

As for our experiment taxonomy (section 4.1) and PROVA! implementation (section 8.2.1), we consider a **system** the hardware and software used to carry on a specific micro-experiment. In this section, we will describe the system details for the devices used in our experiments and the ones used by the experiments we want to reproduce.

In terms of software, we set up a public GitHub project* to handle the repositories containing the *recipes*, i.e. Dockerfile and Singularity definition files, representing the containers used in our experiments. Each repository includes some GitHub Actions [136], which, using the container recipes committed, are automatically building new versions for both Docker and Singularity containers and pushing them respectively to a public project in DockerHub [137] and GitHub artifacts [138] within the same repository. Each GitHub repository represents a specific PROVA! methodType (see section 8.2.1) used to manage the software environment of an experiment on a specific device. Some examples or the containers recipes used are shown in Appendix A.

Whereas our work concentrates on running benchmarks from the MLPerf Inference suite, we are going to follow their device classification splitting between **Edge** and **Datacenter** devices.

*https://github.com/provarepro

## 11.1 Edge devices

We consider part of the edge category the devices that are isolated instances, in contrast with the datacenter ones. In the first round of MLPerf Inference, i.e. v0.5, there was no distinction between edge and datacenter and no other category for the Inference benchmark. With the following benchmark submission rounds and the creation of more benchmark categories, the policies were updated, and, for example, a device like a notebook would be submitting its results to the MLPerf Inference Mobile benchmark (under Notebook section). However, this is important mainly in the case of submitting the results during the benchmark run: for our experiments and discussions, we will consider the notebook part of the edge device category.

### 11.1.1 MLPerf submission

#### System: ICL i3 1005G1

Benchmark

- **Version**: MLPerf Inference benchmark v0.5
- **Submitter**: Intel

Hardware configuration

- **CPU**:
  - **Name**: Intel i3-1005G1 @ 1.20 GHz[†]
  - **Architecture**: Intel Ice Lake (mobile)
  - **Frequency**: 1.20GHz/3.40 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 1/2/2
- **Cache**:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 48 KiB/core (12-way set associative)
  - **L2**: 512 KiB/core (8-way set associative)
  - **L3**: 4 MiB (16-way set associative)
- **Memory**: 1x 4GB DDR4 2666 MHz (MT/s)
- **Accelerator**
  - **Name**: Intel UHD Graphics

[†] https://ark.intel.com/content/www/us/en/ark/products/196588/intel-core-i31005g1-processor-4m-cache-up-to-3-40-ghz.html

  - **Frequency**: 300/900 MHz (Base/Turbo)

Software configuration

- **Framework**: OpenVINO
- **Version**: 2019/pre-release (inferred from submission date)
- **Note**: OpenMP threading and GPU support

### 11.1.2 PROVA! reproduction

#### System: Flex i5 1035G1

Hardware configuration

- **CPU**:
  - **Name**: Intel Core i5-1035G1 @1.00GHz[‡]
  - **Architecture**: Intel Ice Lake (mobile)
  - **Frequency**: 1.00GHz/3.60 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 1/4/2
- **Cache**:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 48 KiB/core (12-way set associative)
  - **L2**: 512 KiB/core (8-way set associative)
  - **L3**: 6 MiB (12-way set associative)
- **Memory**: 2x 8GB DDR4 3200 MHz (MT/s)
- **Accelerator**
  - **Name**: Intel UHD Graphics
  - **Frequency**: 300 MHz /1.05 GHz (Base/Turbo)

Software configuration

- **Configuration 1**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release
  - **Note**: OpenMP threading and GPU support
- **Configuration 2**:

[‡] https://ark.intel.com/content/www/us/en/ark/products/196603/intel-core-i51035g1-processor-6m-cache-up-to-3-60-ghz.html

  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release
  - **Note**: TBB threading and GPU support
- **Configuration 3**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: OpenMP threading and GPU support
- **Configuration 4**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: TBB threading and GPU support

## 11.2 Data center devices

### 11.2.1 MLPerf submission

#### System: Xeon 8276

Benchmark

- **Version**: MLPerf Inference benchmark v0.5
- **Submitter**: Dell EMC

Hardware configuration

- **CPU**:
  - **Name**: Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz[§]
  - **Architecture**: Intel Cascade Lake-SP
  - **Frequency**: 2.20 GHz/4.00 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 2/28/2
- **Cache**[¶]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 1 MiB/core (16-way set associative)
  - **L3**: 38.5 MiB (11-way set associative)
- **Memory**: 760GB DDR4-2933 MHz (MT/s)

[§] https://ark.intel.com/content/www/us/en/ark/products/192470/intel-xeon-platinum-8276-processor-38-5m-cache-2-20-ghz.html

[¶] https://en.wikichip.org/wiki/intel/xeon_platinum/8276

Software configuration

- **Configuration 1**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release (inferred from submission date)
  - **Note**: OpenMP threading

### System: Xeon 6258R

Benchmark

- **Version**: MLPerf Inference benchmark v0.7
- **Submitter**: Intel

Hardware configuration

- **CPU**:
  - **Name**: Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz[‖]
  - **Architecture**: Intel Cascade Lake-SP
  - **Frequency**: 2.70 GHz/4.00 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 2/28/2
- **Cache**[**]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 1 MiB/core (16-way set associative)
  - **L3**: 38.5 MiB (11-way set associative)
- **Memory**: 6x 32GB DDR4-2933 MHz (MT/s)

Software configuration

- **Configuration 1**[139]:
  - **Framework**: TensorFlow
  - **Version**: v2.3.0
  - **Note**: Compiled with mkl optimization and AVX-512 arch support
- **Configuration 2**[140]:

[‖] https://ark.intel.com/content/www/us/en/ark/products/199350/intel-xeon-gold-6258r-processor-38-5m-cache-2-70-ghz.html

[**] https://en.wikichip.org/wiki/intel/xeon_gold/6258r

  - **Framework**: OpenVINO
  - **Version**: 2021/1.pre
  - **Note**: OpenMP threading

- **Configuration 3**[141]:
  - **Framework**: MXNet
  - **Version**: v1.8.0 (custom commit: 6ae469a)
  - **Note**: Compiled with MKLDNN and BLAS=mkl

### System: Xeon 8280

Benchmark

- **Version**: MLPerf Inference benchmark v0.7
- **Submitter**: Dell EMC

Hardware configuration

- **CPU**:
  - **Name**: Intel(R) Xeon(R) Platinum 8280M CPU @ 2.70GHz[††]
  - **Architecture**: Intel Cascade Lake-SP
  - **Frequency**: 2.70 GHz/4.00 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 2/28/2
- **Cache**[‡‡]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 1 MiB/core (16-way set associative)
  - **L3**: 38.5 MiB (11-way set associative)
- **Memory**: 376GB DDR4-2933 MHz (MT/s)

Software configuration Same as *Configuration 2* from **Xeon 6258R** system (11.2.1)

[††] https://ark.intel.com/content/www/us/en/ark/products/192478/intel-xeon-platinum-8280-processor-38-5m-cache-2-70-ghz.html

[‡‡] https://en.wikichip.org/wiki/intel/xeon_platinum/8280

### 11.2.2 PROVA! reproduction

#### System: Xeon 8280 (sciCORE)

sciCORE is a center of competence for scientific computing, providing infrastructures and services for high-performance computing and storage and processing of scientific data [142] located at the University of Basel. From this facility, we selected two different computing architectures for our experiments to compare the evolution of the performance in the cluster. Moreover, the AVX512 processor is the same as the Dell submission, which allows us to conduct a precise reproduction experiment.

##### Hardware configuration

- **CPU**:
  - **Name**: Intel® Xeon® Platinum 8280 Processor[§§]
  - **Architecture**: Intel Cascade Lake-SP
  - **Frequency**: 2.70 GHz/4.00 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 2/28/1 (HT disabled)
- **Cache**[¶¶]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 1 MiB/core (16-way set associative)
  - **L3**: 38.5 MiB (11-way set associative)
- **Memory**: 12x 32GB DDR4-2933 MHz (MT/s) (ECC on)

##### Software configuration

- **Configuration 1**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release
  - **Note**: OpenMP threading
- **Configuration 2**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release

[§§] https://ark.intel.com/content/www/us/en/ark/products/192478/intel-xeon-platinum-8280-processor-38-5m-cache-2-70-ghz.html

[¶¶] https://en.wikichip.org/wiki/intel/xeon_platinum/8280

  - **Note**: TBB threading
- **Configuration 3**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: OpenMP threading
- **Configuration 4**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: TBB threading
- **Configuration 5**:
  - **Framework**: OpenVINO
  - **Version**: 2021/1.pre
  - **Note**: OpenMP threading
- **Configuration 6**:
  - **Framework**: OpenVINO
  - **Version**: 2021/1.pre
  - **Note**: TBB threading
- **Configuration 7**:
  - **Framework**: TensorFlow
  - **Version**: v2.3.0
  - **Note**: Compiled with mkl optimization and AVX-512 arch support
- **Configuration 8**:
  - **Framework**: MXNet
  - **Version**: v1.8.0 (custom commit: 6ae469a)
  - **Note**: Compiled with MKLDNN and BLAS=mkl

### System: Xeon E5-2630 v4 (sciCORE)

Hardware configuration

- **CPU**:
  - **Name**: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz[***]
  - **Architecture**: Intel Broadwell-EP
  - **Frequency**: 2.20 GHz/3.10 GHz (Base/Turbo)

[***] https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e52630-v4-25m-cache-2-20-ghz.html

- **Sockets/Core per socket/Threads per core**: 2/10/1 (HT disabled)
- **Cache**[†††]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 256 KiB/core (8-way set associative)
  - **L3**: 25 MiB (20-way set associative)
- **Memory**: 8x 32GB DDR4-2667 MHz (MT/s) (ECC on)

Software configuration

- **Configuration 1**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release
  - **Note**: OpenMP threading
- **Configuration 2**:
  - **Framework**: OpenVINO
  - **Version**: 2019/pre-release
  - **Note**: TBB threading
- **Configuration 3**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: OpenMP threading
- **Configuration 4**:
  - **Framework**: OpenVINO
  - **Version**: 2019/R3.1
  - **Note**: TBB threading

### System: Xeon 6258R (miniHPC)

MiniHPC is a small high-performance computing (HPC) cluster [143] located at the University of Basel. For our experiments, we selected a processor which happens to be the same used in the Intel submission, i.e., the "1-node-2S-CLX" system, which allows us to try reproducing all the three different software configurations for that specific MLPerf Inference submission.

[†††] https://en.wikichip.org/wiki/intel/xeon_e5/e5-2630_v4

Hardware configuration

- **CPU**:
  - **Name**: Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz[‡‡‡]
  - **Architecture**: Intel Cascade Lake-SP
  - **Frequency**: 2.70 GHz/4.00 GHz (Base/Turbo)
- **Sockets/Core per socket/Threads per core**: 2/28/2
- **Cache**[§§§]:
  - **L1i**: 32 KiB/core (8-way set associative)
  - **L1d**: 32 KiB/core (8-way set associative)
  - **L2**: 1 MiB/core (16-way set associative)
  - **L3**: 38.5 MiB (11-way set associative)
- **Memory**: 12x 128GB DDR4-2933 MHz (MT/s) (ECC on)

Software configuration

- **Configuration 1**:
  - **Framework**: TensorFlow
  - **Version**: v2.3.0
  - **Note**: Compiled with mkl optimization and AVX-512 arch support
- **Configuration 2**:
  - **Framework**: MXNet
  - **Version**: v1.8.0 (custom commit: 6ae469a)
  - **Note**: Compiled with MKLDNN and BLAS=mkl
- **Configuration 3**:
  - **Framework**: OpenVINO
  - **Version**: 2021/1.pre
  - **Note**: OpenMP threading
- **Configuration 4**:
  - **Framework**: OpenVINO
  - **Version**: 2021/4
  - **Note**: OpenMP threading

[‡‡‡] https://ark.intel.com/content/www/us/en/ark/products/199350/intel-xeon-gold-6258r-processor-38-5m-cache-2-70-ghz.html

[§§§] https://en.wikichip.org/wiki/intel/xeon_gold/6258r

# Chapter 12

# MLPerf Inference Benchmark Experiments

Our experiments will focus on the MLPerf Inference Benchmark Suite, and in particular on the Computer Vision tasks: Image Classification and Object Detection. This decision was taken mainly based on the submissions available for the framework and the systems we wanted to experiment with but also since, especially for the first benchmark versions, these tasks represent most of the total submissions [44].

The majority of the experiments will concentrate on the first two versions of the benchmark, which were more interesting in terms of reproducibility challenges, but will as well provide results of the newer benchmark versions, thus investigating not only the evolution of frameworks/devices for machine learning but also the benchmark itself.

Moreover, for each of the considered submission codes, we analyze different experiment configurations using different systems to have a better understanding of the performance behaviour, which will cover the **Repetition** and **Replication** reproducibility levels (section 4.2). Even though we can run different models changing, for example, the data format precision, the *method* used to solve a task will remain unchanged (since defined by the benchmark), meaning there will not be real **Re-experimentation** covered.

Beyond reproducibility, the experiments will also demonstrate some properties of our PROVA! tool in terms of software engineering, such as portability, extensibility, reusability, adaptability.

## 12.1 MLPerf v0.5

Following the discussion started in Chapter 6, we moved the manual experiments regarding the OpenVINO submissions for the MLPerf Inference benchmark v0.5 in PROVA!, using the driver mode (see chapter 10).

For the first benchmark round, two participants run the OpenVINO framework to benchmark their devices: Dell EMC and Intel. The details of the subset of scenarios submitted are summarized in Table 12.1.

| System | Framework | Workloads | | | |
|---|---|---|---|---|---|
| | | Image Classification | | Object Detection | |
| | | ICL | ICH | ODL | ODH |
| ICL i3 1005G1 | OpenVINO | SS, O | SS, O | SS, O | — |
| CLX-9282-2S | OpenVINO | — | — | SS, S, O | — |
| R740xd8276 | OpenVINO | — | — | SS, O | — |
| R740xd6248 | OpenVINO | — | — | SS, O | — |

**Table 12.1:** *MLPerf Inference Benchmark v0.5 scenarios submitted for the Intel and Dell EMC systems selected (section 11.1 and 11.2): Single-Stream (**SS**), Server (**S**) and Offline (**O**).*

We selected the *edge* device used by Intel and the *datacenter* device used by Dell EMC to compare against because they are similar to our systems. This experiment reproduction will include three different benchmark tasks, CPU and accelerator (GPU) devices and different versions and configurations of the framework used, as shown in figure 12.1.

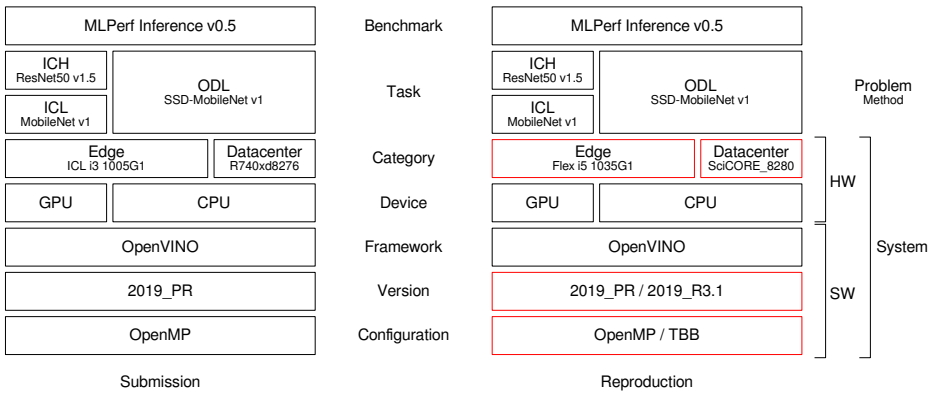



**Figure 12.1:** *MLPerf Inference v0.5 reproduction: High level view of the experiment elements we changed (red) for our reproduction. The method is defined by the benchmark itself and bound to the task.*

### 12.1.1 Object Detection Lightweight task

Table 12.2 shows a schematic view of the first MLPerf Inference experiment we try to reproduce: the *Problem* and the *Method* are the ones defined by the MLPerf Inference benchmark while the *System* changes from the original to the reproduced. Intel used a laptop with four cores (2 physical + HT), while our system has eight cores (4 physical + HT). For this reason, we decided to configure our system to use only two cores in our first reproduced experiment.

Still, in the second one, we used it without restriction, comparing the original OpenVINO version configured to use OpenMP threading with a version using TBB threading running the Object Detection task in a Single-Stream scenario. Finally, we run the same benchmark task in the Offline scenario to investigate the OpenVINO behaviour when changing some of the parameters.

| | **Original Exp.** | **Reproduced Exp. 1** | **Reproduced Exp. 2** |
|---|---|---|---|
| **Problem** | Object detection (Single-Stream/Offline scenario) [44]<br>Dataset: COCO [144] (300x300) | | |
| **Method** | Single Shot MultiBox Detector (SSD) [133]<br>Model: SSD-MobileNet-v1 | | |
| **System** | • Device: ICL i3 1005G1<br>• Framework: OpenVINO (Configuration 1) | • Device: Flex i5 1035G1*<br>• Framework: *Unchanged* | • Device: Flex i5 1035G1<br>• Framework: OpenVINO (Configuration 1-2) |

* Used in a 2-core/4-thread configuration

**Table 12.2:** *Reproduction of OpenVINO experiments submitted by Intel as part of the MLPerf Inference Benchmark v0.5: object detection (lightweight) task for Single-Stream and Offline scenarios. A detailed description of the systems is available in section 11.1.*

In the first reproduced experiment, we replicate the original experiment applying the same parameters configuration documented in the submission. The performance comparison is shown in Figure 12.2: the results are pretty different for such a small execution. When using eight threads as in the original experiment, our system performs much worse (-43%), which was not expected since the systems are similar.

The system used by Intel to run this benchmark has four cores (including HT): using more threads than the number of physical cores is not always providing better performance. In this case, the number of threads used is even higher than the number of virtual cores, which is not the best value expected.

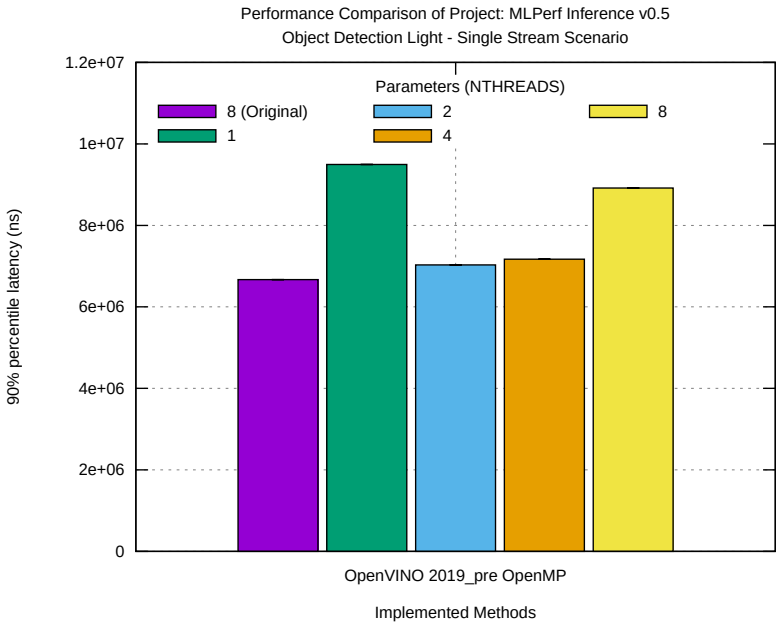


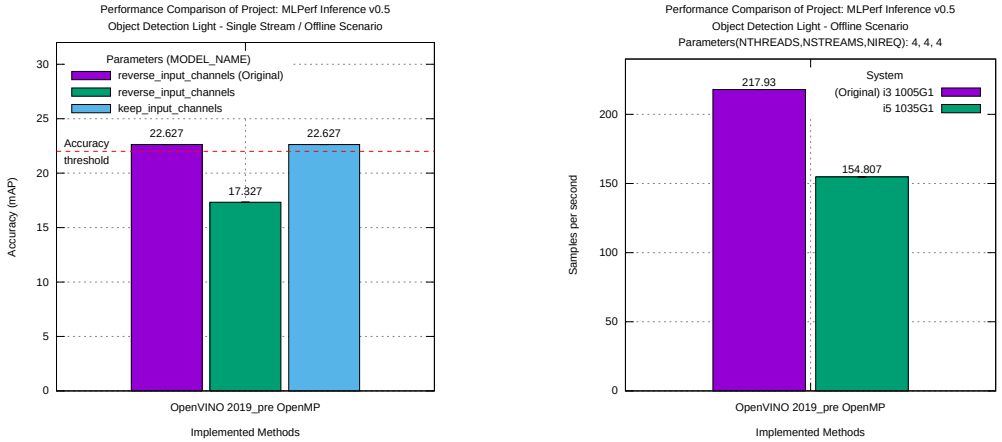

**Figure 12.2:** *MLPerf Inference v0.5: Object Detection lightweight, Single-Stream scenario. Intel submission* compared to the experiment reproduction on our system (see Reproduced Experiment 1, table 12.2) using different values of threads.*

To validate the parameters specified by Intel and fine-tune those for our systems, we add parameters like the number of requests, threads, and streams (*nireq/nthreads/nstreams*) to our PROVA! project. When running a *Single-Stream* scenario, changing the number of requests and streams is not affecting the results since MLPerf LoadGen sends the requests sequentially, waiting for the previous to be completed [44]. That is why we consider *nthreads* the only parameter to tune.

Figure 12.2 shows that, with two physical cores (4 virtual ones) on our system, the best performance is achieved when the number of threads requested is equal to the number of physical cores: our best result is still worse than the original experiment's one but now only by 5%.

To complete the experiment reproduction, we needed to run the benchmark in **accuracy mode** since MLPerf submission requires, for each benchmark task, a minimal accuracy to be met to consider the submission valid.

*MLPerf v0.5 Inference Closed SSD-MobileNets-v1 Single-Stream. Retrieved from `https://mlcommons.org/en/inference-edge-05` 18 April 2021, entry Inf-0.5-24. MLPerf name and logo are trademarks. See `https://mlcommons.org` for more information.



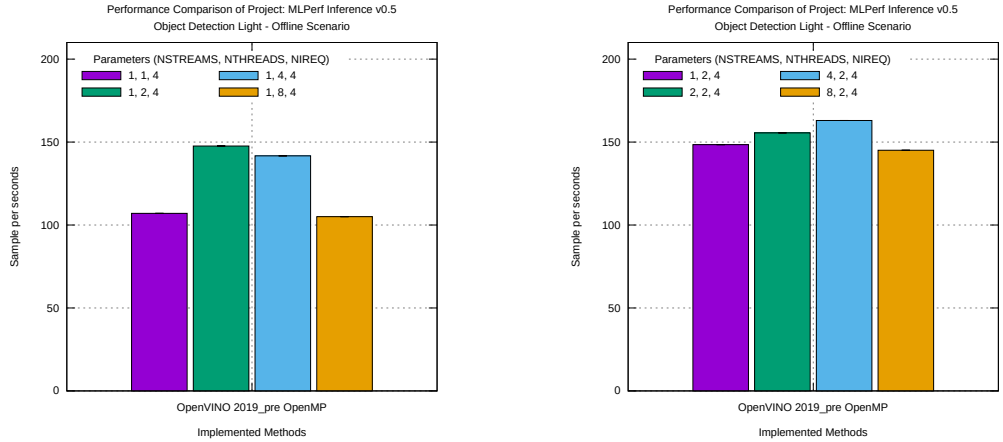

**(a)** *Accuracy comparison.* **(b)** *Performance comparison.*

**Figure 12.3:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Intel submission*[†] *compared to the experiment reproduction on our system (see Reproduced Exp. 1, table 12.2).*

When running the benchmark in accuracy mode, we noticed a big difference from the one reported in MLPerf. Figure 12.3a shows both the original accuracy (purple) and the reproduced one (green). This accuracy would be too low for the submission to be accepted (see the *accuracy threshold* dashed line). Based on a discussion on the official MLPerf GitHub repository [145], the low accuracy seems to be related to the "--reverse_input_channels" option used when generating the OpenVINO model as described by Intel in their MLPerf submission [146]. Still, this explanation has not been validated by the submitter. In our PROVA! project, we set the *mode* to **accuracy** and added a *MODEL_NAME* parameter to run the experiment with different models (w/wo "--reverse_input_channels" option). The blue bar in figure 12.3a shows that we can achieve the same accuracy as the official submission if we skip the option documented by Intel.

We did similar steps for the Offline scenario. We start comparing the results we get using the same original parameter configuration, and we then try to tune the parameters. In this case, the values for *nireq/nthreads/nstreams* are set to the number of logical cores available on the system. Figure 12.3b shows that our device performs worse than the original, even in the Offline scenario (-30%). To tune the parameters, we first decided to

[†]MLPerf v0.5 Inference Closed SSD-MobileNets-v1 Offline. Retrieved from `https://mlcommons.org/en/inference-edge-05` 18 April 2021, entry Inf-0.5-24. MLPerf name and logo are trademarks. See `https://mlcommons.org` for more information.




**(a)** *Threads tuning.* **(b)** *Streams tuning.*

**Figure 12.4:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Performance tuning for OpenVINO 2019 pre-version configured with OpenMP threading: running on our system (see Reproduced Exp. 1, table 12.2) using code submitted by Intel[148].*

try increasing only the number of threads, keeping the number of inference requests fixed to the number of logical cores and the number of streams to 1. As for the Single-Stream scenario, we notice the best performance achieved using physical cores as the number of threads (Figure 12.4a).

In case of experiments with a short execution time, Intel suggests using a throughput-oriented approach for the parallelization, especially if running on CPUs: increasing the number of streams, OpenVINO will assign/pin them to the execution resources to increase the throughput [147]. Figure 12.4b shows that the performance grows with the number of streams, which is best when this number equals the logical cores: the performance reached is close to the one achieved using the official parameters, which, in this case, we can confirm to be effectively optimal.

After this first experiment mainly aimed at reproducing the original submission, in a second reproduced experiment, we tried to perform the same benchmark task (Object Detection lightweight) using our system without adding any resource limitation. We ran the Single-Stream scenario using two different versions of the OpenVINO framework: one uses the OpenMP library (same configuration suggested by Intel) and the other the Threading Building Blocks (TBB) library (the default in OpenVINO starting from version 2019). As for the previous experiment, using the physical cores available as the number of threads provides the best performance (Figure 12.5).

Furthermore, even if the OpenMP version of OpenVINO performs better than the TBB one, this difference is always less than 4%.

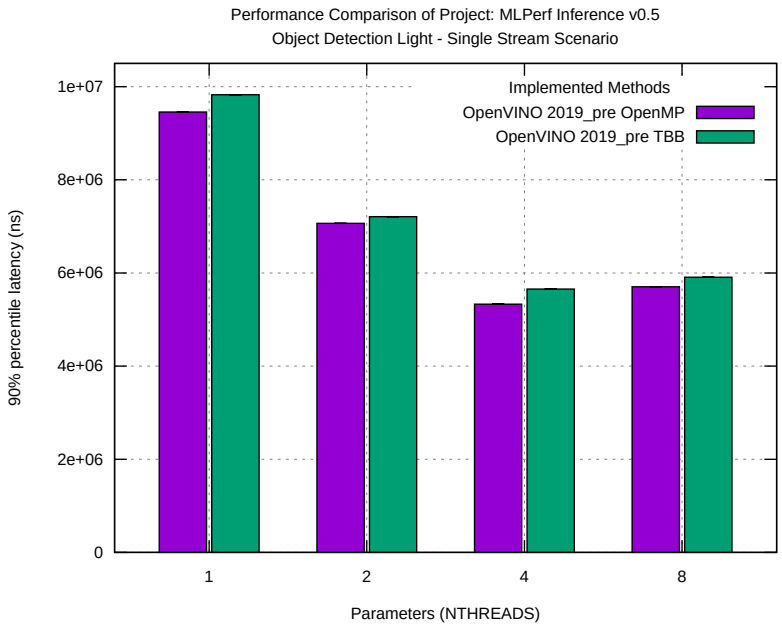



**Figure 12.5:** *MLPerf Inference v0.5: Object Detection lightweight, Single-Stream scenario. Performance comparison of OpenVINO 2019 pre-version configured with OpenMP and TBB threading: running on our system (see Reproduced Exp. 2, table 12.2) using code submitted by Intel[148] and different values of threads.*

When running the benchmark in the Offline scenario, we were able to investigate the behaviour of OpenVINO for both versions (OpenMP/TBB) by keeping the number of threads and inference requests constant (optimal values) and varying both the number of streams and the batch size. Figure 12.6 shows that increasing the batch size value affects (albeit minimally) the results but only when not using multiple streams (throughput-based approach discussed previously). In fact, if we use the suggested number of streams (number of logical cores available), the batch size does not play a role, and the results obtained are the optimal ones. Again, the two versions behave similarly with the TBB one, which seems slightly better when using one stream and bigger batch size.

Finally, keeping out the batch size set to 1 (as suggested), we experiment with how the parallelization works in the OpenMP version, analyzing its behaviour with different combinations of threads and inference requests for an increasing value of streams.

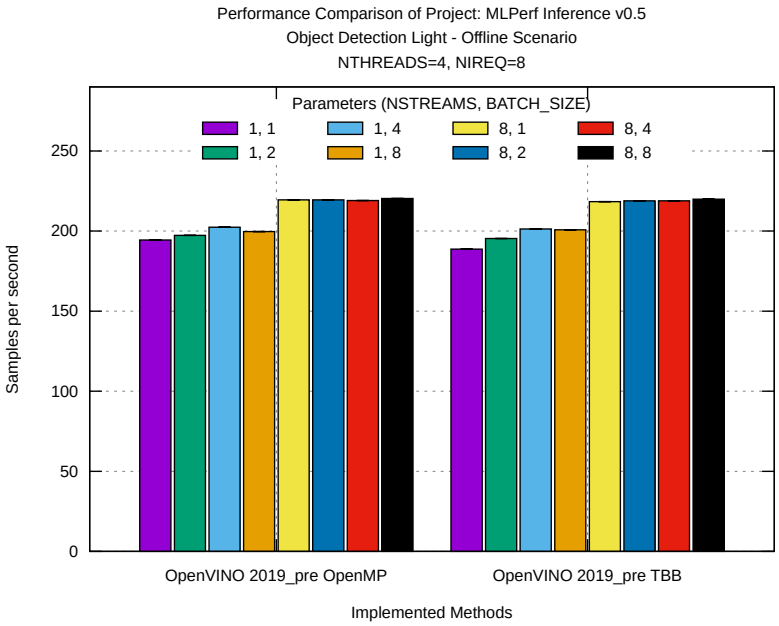



**Figure 12.6:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Comparison of performance behavior of OpenVINO 2019 pre-version configured with OpenMP and TBB threading when changing the number of streams and batch sizes: running on our system (see Reproduced Exp. 2, table 12.2) using code submitted by Intel[148].*

As discussed previously, when the number of streams is set to 1, the parallelization happens in a traditional way using multiple threads for solving each request coming to the stream. In this case, the number of inference requests doesn't matter for the performance, which is based on the number of threads. Increasing the streams, we start parallelizing the "outer loop" of the requests: the available resources are split/assigned to different streams which are working in parallel. In the case of 2 streams, using 1 or 2 threads provides the same performance since the total amount of cores used will always be 2 (bounded either by the number of streams or threads). When increasing the number of threads, the performance is not bounded anymore since the system has four physical cores. We can already notice that the overall performance is increasing when moving from 1-stream to 2-streams configuration, showing how, in a benchmark like this with no heavy requests, the throughput-approach parallelization works better. Moving to 4 streams, we can see that all four cores are used in each thread-request configuration with always similar performance. Finally, in the 8-streams run, the results are affected by the number of inference requests: creating

fewer requests than the number of available streams brings performance down. In this case, the performance gets even worse than the run using four streams because the 8-streams configuration will threaten all the cores independently, which results in more requests to possibly go to the same physical core, causing a high load imbalance (Figure 12.7).

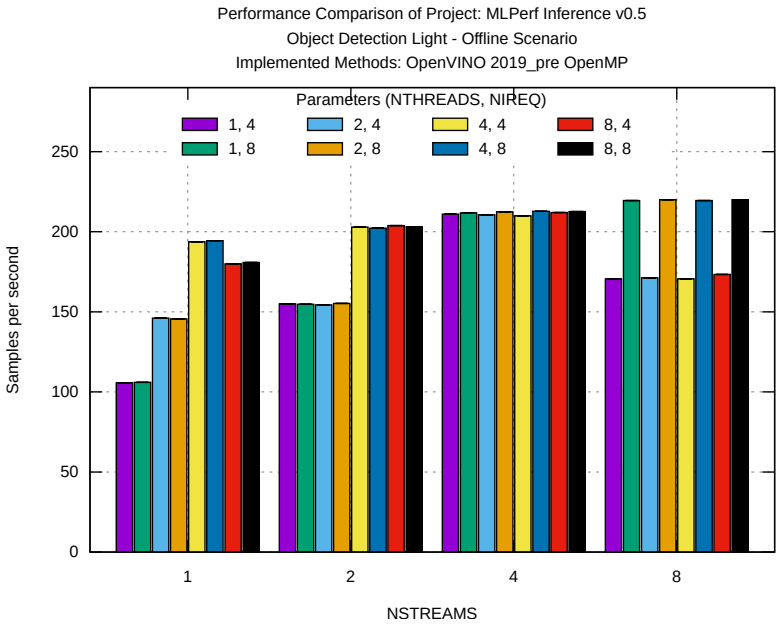



**Figure 12.7:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Comparison of performance behavior of OpenVINO 2019 pre-version configured with OpenMP threading when using a variable number of streams and a combination of a different number of threads and inference requests: running on our system (see Reproduced Experiment 2, table 12.2) using code submitted by Intel[148].*

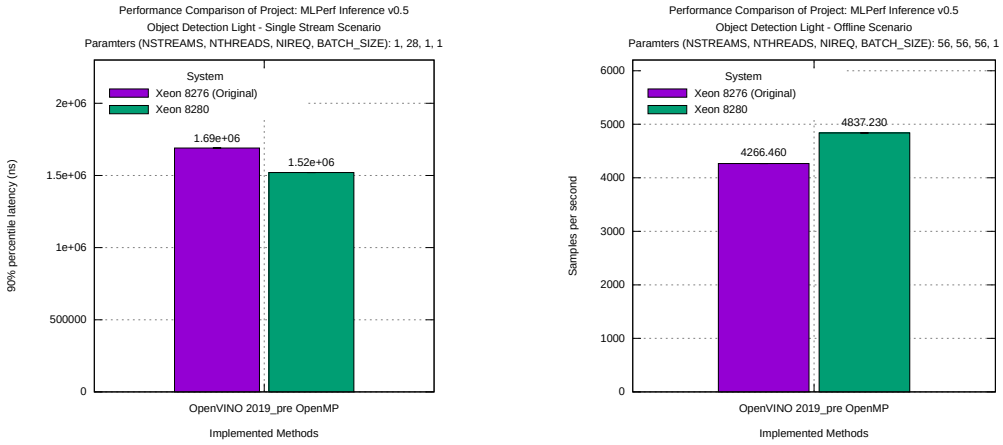
For the same Benchmark task, i.e. Object Detection lightweight, we also tried to reproduce the Dell submission in the datacenter category using a machine hosted in the sciCORE data center at the University of Basel. The processor we used is from the same generation as the one used in the original submission (Cascade Lake) but has a higher base CPU frequency (2.70 vs 2.20 GHz). Table 12.3 shows the experiment in terms of <Problem, Method, System> 3-tuple.

The OpenVINO configuration of the Dell submission was the same as the Intel one. For this reason, we reuse the ones we built for the previous experiment. Moreover, such submission does not provide the scripts used to run the experiments, and thus no experiment parameters are available.

| | Original Exp. | Reproduced Exp. 1 | Reproduced Exp. 2 |
|---|---|---|---|
| **Problem** | Object detection (Single-Stream/Offline scenario) [44]<br>Dataset: COCO [144] (300x300) | | |
| **Method** | Single Shot MultiBox Detector (SSD) [133]<br>Model: SSD-MobileNet-v1 | | |
| **System** | • Device: Xeon 8276<br>• Framework: OpenVINO<br>(Configuration 1) | • Device: Xeon 8280 (sciCORE)<br>• Framework: OpenVINO<br>(Configuration 1-3) | • Device: Xeon E5-2630<br>• Framework: OpenVINO<br>(Configuration 1-3) |

**Table 12.3:** *Reproduction of OpenVINO experiments submitted by Dell as part of the MLPerf Inference Benchmark v0.5: object detection (lightweight) task for Single-Stream and Offline scenarios. A detailed description of the systems is available in section 11.2.*

Since Intel submitted benchmark results using a *datacenter* machine with the same socket/cores configuration, we used the parameters specified by them. Our system performs around 11% better in both Single-Stream and Offline scenarios (figure 12.8), which is somehow expected since, even though the max frequency is the same, we have a 20% difference in base CPU frequency.



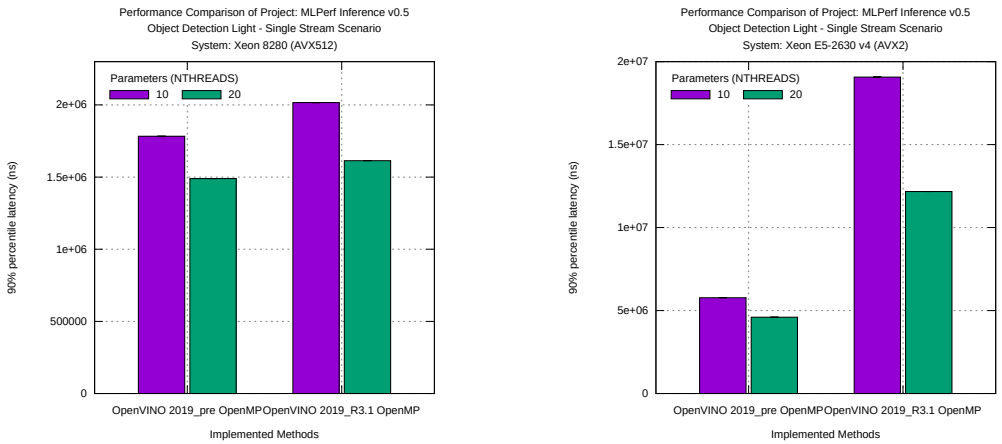

**(a)** *Single-Stream scenario.* **(b)** *Offline scenario.*

**Figure 12.8:** *MLPerf Inference v0.5: Object Detection lightweight. Dell submission[‡] compared to the experiment reproduction on our system (see Reproduced Exp. 1, table 12.3) using code submitted by Intel[148].*

After reproducing the original configuration, we decided to change some experiment elements like the software version, as well as varying

[‡]MLPerf v0.5 Inference Closed SSD-MobileNets-v1 Offline. Retrieved from https://mlcommons.org/en/inference-edge-05 18 April 2021, entry Inf-0.5-4. MLPerf name and logo are trademarks. See https://mlcommons.org for more information.

*NSTREAMS*, *NTHREADS* and *NIREQ* parameters. We set up an experiment using the final release of the OpenVINO framework version, i.e., 2019 R3.1, instead of the "pre-release" used by the submitters. For the Single-Stream scenario (figure 12.9a), the performance gets worse when using all the available physical cores, confirming the Intel parameters configuration as the best possible. The Single-Stream scenario is not worth running using a *datacenter* machine: in fact, starting with the sequent benchmark version, the submission for this category includes only Server and Offline scenarios as mandatory. From the same experiment, we can see also how the pre-release version performs better than the later version, even though the difference is not much relevant. Different is the situation shown in figure 12.9b: we executed the same experiment on an older Intel process architecture (Broadwell) available in the same sciCORE data center (Reproduced Exp. 2 of table 12.3), finding a huge difference in performance when using the two OpenVINO versions. The performance of the final release of OpenVINO 2019 is more than 2.5 times slower compared to the pre-release in the case of using 20 threads: such a difference goes up to more more than three times when using 10 threads.




**(a)** *Performance on Cascade Lake architecture.*

**(b)** *Performance on Broadwell architecture.*

**Figure 12.9:** *MLPerf Inference v0.5: Object Detection lightweight, Single-Stream scenario. Comparison of different versions of OpenVINO framework running on different processor architectures (see Reproduced Exp. 1 (a) and Reproduced Exp. 2 (b) columns of Table 12.3) using code submitted by Intel[148].*

The same comparison, based on versions, has been performed with the Offline scenario as well. Using the Cascade Lake processor (figure 12.10), we

observe the same behaviour shown by the Single-Stream case: (1) a slightly different performance (still in favour of the pre-release) and (2) the best performance are achieved using the parameters configuration used by Intel.

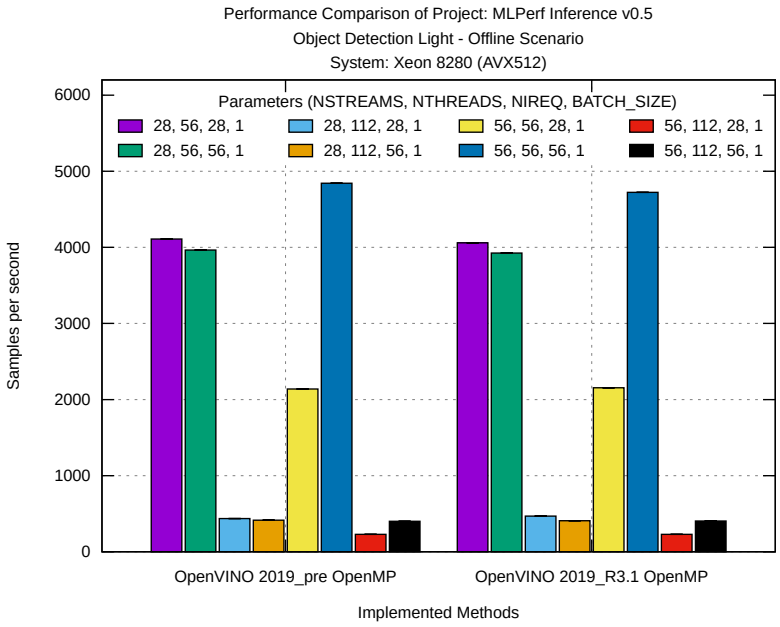



**Figure 12.10:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Comparison of different versions of OpenVINO framework when using a different number of streams, threads and inference requests: running on our system (see Reproduced Experiment 1, table 12.3) using code submitted by Intel[148].*

Moving to the Broadwell processor, we experience a relevant performance loss again for the newer OpenVINO version, especially for the parameters combinations that generate the best performance: in this case, the newer software version is almost four times slower (figure 12.11).

Besides the performance, we can check if the processor version affects the task accuracy as well. The accuracy reported in the official submission is 22.627 mAP which we could exactly reproduce on our Cascade Lake processor (figure 12.12) while we obtain a lower accuracy when changing the processor and the software version (even below the acceptable threshold, if using the old processor in combination with the OpenVINO 2019 R3.1 version).

In our last experiment related to this task, within the datacenter category, we compared two versions of the OpenVINO 2019 pre-release compiled with different threading libraries, i.e., OpenMP and TBB. Figure 12.13

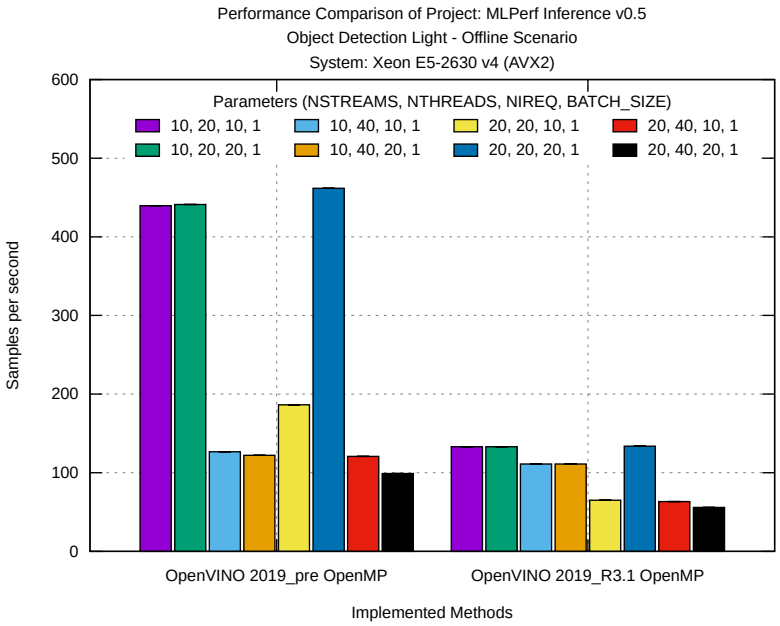



**Figure 12.11:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Comparison of different versions of OpenVINO framework when using a different number of streams, threads and inference requests: running on our system (see Reproduced Experiment 2, table 12.3) using code submitted by Intel[148].*

shows how the best performance is still achieved by the OpenMP, but the TBB version is not much worse and seems to better manage cases that use more threads.

### 12.1.2 Image Classification Heavyweight task

As shown in figure 12.1, for the same benchmark submission round, Intel provided results for other tasks using the same edge system. The experiment details for the Image Classification (heavyweight) task are presented in Table 12.4.

In this case, the system used also includes an Intel GPU device which was used in the original experiment to accelerate the Offline scenario. As documented in the Intel submission[146], we calibrated the official FP32 model to use INT8 precision after converting it from TensorFlow to OpenVINO format. Following the Intel suggestion, we kept FP32 data type as a base for calibrating INT8 to be executed on CPU (Single-Stream scenario) while we used FP16 data type for the model used as the base for the GPU

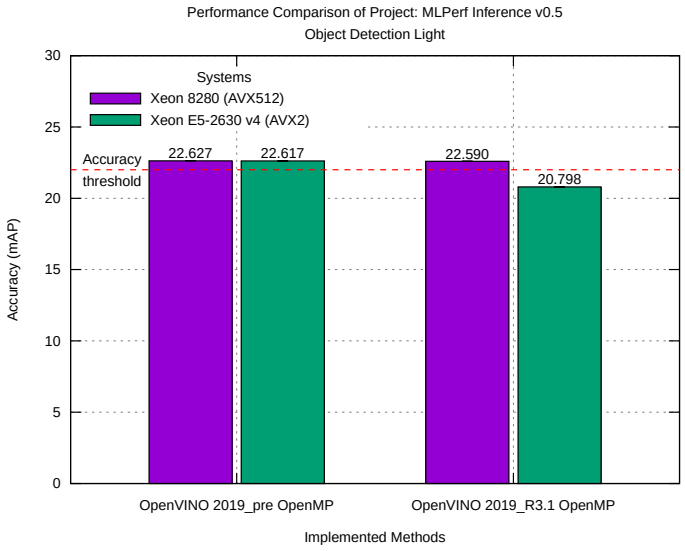



**Figure 12.12:** *MLPerf Inference v0.5: Object Detection lightweight. Comparison of model accuracy for different versions of OpenVINO framework running on different processor architectures (see Reproduced Exp. 1 (a) and Reproduced Exp. 2 (b) columns of Table 12.3) using code submitted by Intel[148].*

| | Original Exp. | Reproduced Exp. 1 | Reproduced Exp. 2 |
|---|---|---|---|
| **Problem** | Image Classification (Single-Stream/Offline scenario) [44]<br>Dataset: ImageNet [149] (224x224) | | |
| **Method** | Deep Residual Learning [150]<br>Model: ResNet-50 v1.5 | | |
| **System** | • Device: ICL i3 1005G1<br>• Framework: OpenVINO (Configuration 1) | • Device: Flex i5 1035G1*<br>• Framework: *Unchanged* | • Device: Flex i5 1035G1<br>• Framework: OpenVINO (Configuration 1-4) |

* Used in a 2-core/4-thread configuration

**Table 12.4:** *Reproduction of OpenVINO experiments submitted by Intel as part of the MLPerf Inference Benchmark v0.5: image classification (heavyweight) task for Single-Stream and Offline scenarios. A detailed description of the systems is available in section 11.1.*

calibration (Offline scenario). The different base for the calibration slightly affects the accuracy, which changes from one scenario to the other: we repeated all the steps described by Intel and could get the same behaviour but not the identical results (figure 12.14).

[§]MLPerf v0.5 Inference Closed ResNet-50 v1.5 Offline. Retrieved from https://mlcommons.org/en/inference-edge-05 18 April 2021, entry Inf-0.5-24. MLPerf name and logo are trademarks. See https://mlcommons.org for more information.

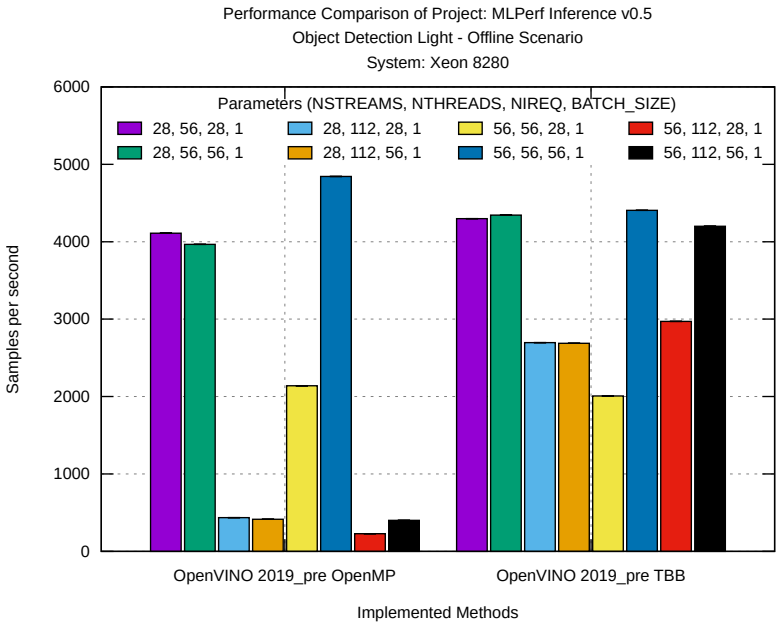



**Figure 12.13:** *MLPerf Inference v0.5: Object Detection lightweight, Offline scenario. Comparison of different versions of OpenVINO framework when using a different number of streams, threads and inference requests: running on our system (see Reproduced Experiment 2, table 12.3) using code submitted by Intel[148].*

In our tries, we saw a tiny variance in the accuracy calculated using the Intel GPU, but that was never the case for the experiments run on the CPU. That is why we were expecting to exactly reproduce the model accuracy (at least for CPU) like in the object detection experiment.

To verify how the model data precision is affecting the performance, we not only run the suggested configuration but also try using the different models on both devices (CPU/GPU). The results from the Single-Stream scenario are shown in figure 12.15: neither using GPU acceleration nor a model built starting from FP16 data precision is bringing a performance improvement, but, also for this task, we could not reproduce the original results (∼25% worse performance).

The same happens when running the Offline scenario. As for the original experiment, we reached the best performance using the Intel GPU hardware and the model based on FP16 data precision, but we still have a performance circa 25% worse than the Intel benchmark (figure 12.16).

¶See footnote for Figure 12.14
‖See footnote for Figure 12.14

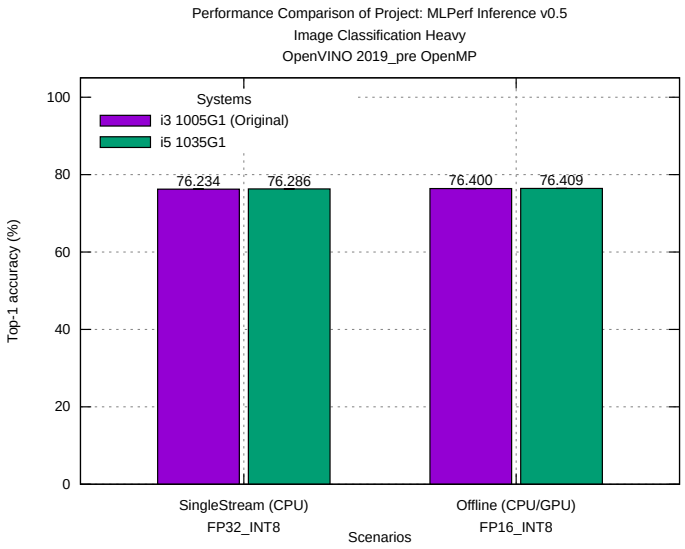



**Figure 12.14:** *MLPerf Inference v0.5: Image Classification heavyweight. Model accuracy of Intel submission*[§] *compared to the experiment reproduction on our system (see Reproduced Exp. 1, table 12.4).*

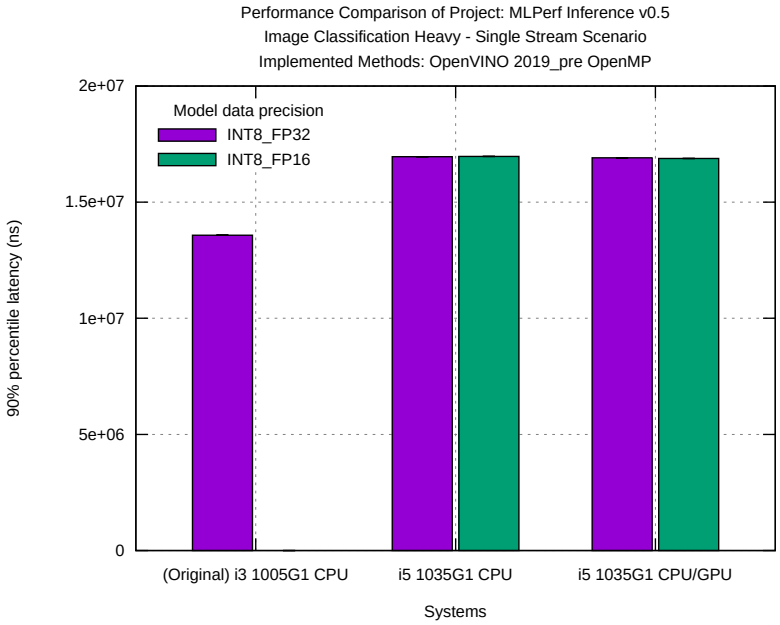



**Figure 12.15:** *MLPerf Inference v0.5: Image Classification heavyweight, Single-Stream scenario. Intel submission*[¶] *compared to the experiment reproduction on our system using both CPU-only and GPU systems (see Reproduced Exp. 1, table 12.4) with different model precision.*

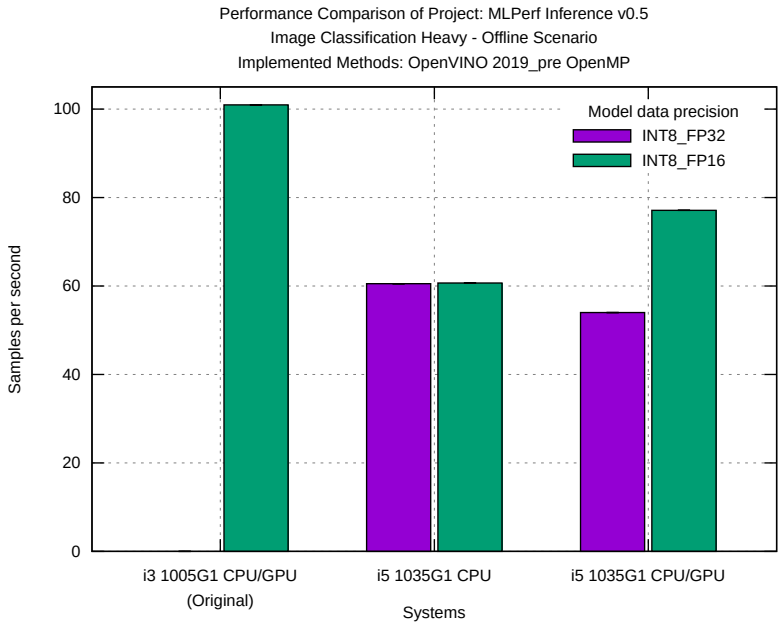



**Figure 12.16:** *MLPerf Inference v0.5: Image Classification heavyweight, Offline scenario. Intel submission*[‖] *compared to the experiment reproduction on our system using both CPU-only and GPU systems (see Reproduced Exp. 1, table 12.4) with different model precision.*

To perform a parallel analysis with the previous benchmark experiment, we compare performance using the TBB threaded implementation of both pre and final versions of OpenVINO 2019. Figure 12.17 shows the GPU executions perform better only when using INT8 precision in combination with FP16 for all of the framework versions. Even in this experiment, the difference between OpenMP and TBB is not impressive, but it is in favour of the TBB variant, differently from before. The configuration applied is the one used in the benchmark result submission, i.e., NSTREAMS=4, NTHREADS=8, NIREQ=8, BATCH_SIZE=1. As for the object detection task, we found the best configuration to be using NSTREAMS=1, NTHREADS=2, NIREQ=4, BATCH_SIZE=1. Nevertheless, the improvement falls within a few percentage points.

### 12.1.3 Image Classification Lightweight task

The last benchmark experiment for MLPerf inference v0.5 is the Image Classification (lightweight) task (table 12.5).

This task follows the same approach as the other image classification

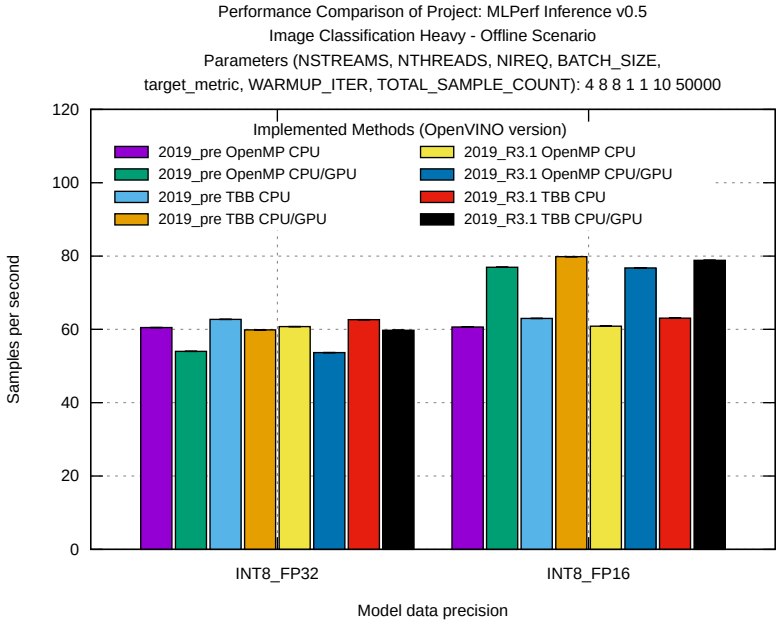



**Figure 12.17:** *MLPerf Inference v0.5: Image Classification heavyweight, Offline scenario. Performance comparison of OpenVINO 2019 pre-version configured with OpenMP and TBB threading: running on both CPU-only and GPU systems (see Reproduced Exp. 1, table 12.4) using code submitted by Intel[148] and different model precision.*

| | Original Exp. | Reproduced Exp. 1 | Reproduced Exp. 2 |
|---|---|---|---|
| **Problem** | Image Classification (Single-Stream/Offline scenario) [44]<br>Dataset: ImageNet [149] (224x224) | | |
| **Method** | MobileNet [134]<br>Model: MobileNet-v1 224 | | |
| **System** | • Device: ICL i3 1005G1<br>• Framework: OpenVINO (Configuration 1) | • Device: Flex i5 1035G1*<br>• Framework: *Unchanged* | • Device: Flex i5 1035G1<br>• Framework: OpenVINO (Configuration 1-2) |

* Used in a 2-core/4-thread configuration

**Table 12.5:** *Reproduction of OpenVINO experiments submitted by Intel as part of the MLPerf Inference Benchmark v0.5: image classification (lightweight) task for Single-Stream and Offline scenarios. A detailed description of the systems is available in section 11.1.*

experiment with CPU used for the Single-Stream and GPU for the Offline scenario, as well as the calibration of the model to use INT8 data precision. The accuracy of both CPU and GPU model versions could not be exactly replicated even though quite similar (figure 12.18).

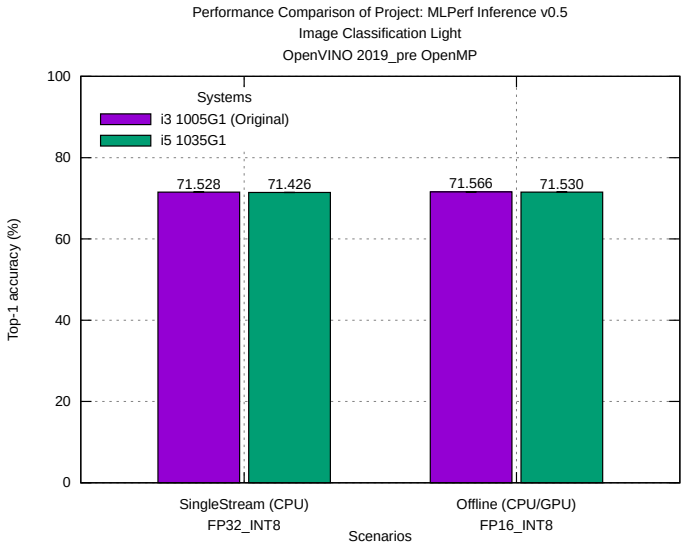



**Figure 12.18:** *MLPerf Inference v0.5: Image Classification lightweight. Model accuracy of Intel submission[**] compared to the experiment reproduction on our system (see Reproduced Exp. 1, table 12.5).*

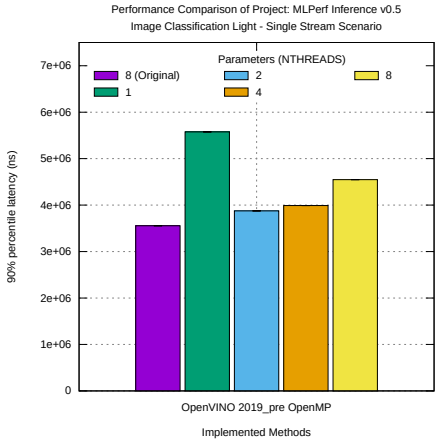
The first test ran the Single-Stream scenario using only the CPUs of the device like in the original configuration (Reproduces Exp.1 in table 12.5). Despite the fact that Intel uses eight threads for the benchmark submission, we found the best amount of threads to be two (equal to the number of physical cores), confirming the behaviour we already observed with the object detection experiment (figure 12.19a). With our optimal configuration, the results are not far from the original (a bit less than 10%).

The difference with original results increases when running the Offline scenario: our performance stops at a level 40% lower than the Intel benchmark (figure 12.19b).

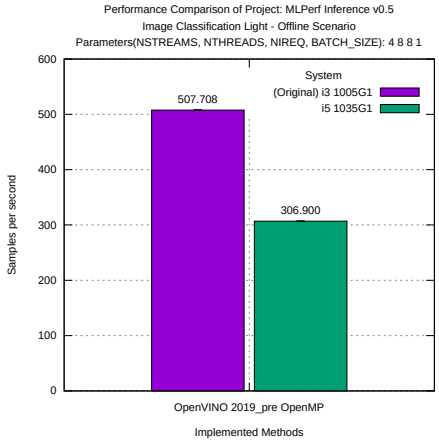
To understand the best parameter configuration, we run various combinations testing as well the TBB version of OpenVINO. For all we could

**MLPerf v0.5 Inference MobileNet-v1 Offline. Retrieved from https://mlcommons.org/en/inference-edge-05 18 April 2021, entry Inf-0.5-24. MLPerf name and logo are trademarks. See https://mlcommons.org for more information.

††See footnote for Figure 12.18




**(a)** *Single-Stream scenario using variable number of threads.*




**(b)** *Offline scenario.*

**Figure 12.19:** *MLPerf Inference v0.5: Image Classification lightweight. Intel submission*[††] *compared to the experiment reproduction on our system (see Reproduced Exp. 1, table 12.5).*

identify a better parameters combination, the performance results are still far from the benchmark, as one can see in figure 12.20.

### 12.1.4 Reproducibility considerations

During the reproduction of the Intel submission, we faced several issues with the code and instructions provided, which forced us to make some changes.

Sample size Using the Windows OS version, the code requires to pass the *total_sample_count* value as a parameter. In the script available as part of the submission, the value used is 50000, which causes the failure of the benchmark execution. Based on the Linux OS version of the code and the logs for the Windows OS version, we figured out the correct value for Object detection tasks, which uses the COCO dataset, is 5000. Instead, a value of 50000 is correct in the case of the Image processing tasks since using the Imagenet dataset. We adapted the sample value accordingly.

Code discrepancies The common part of Windows and Linux code versions is identical except for one function, namely the order of two input parameters. As a result, none of the code fails when running in **Performance** mode, but the situation changes when running the code using the **accuracy** mode: the code version developed for Windows OS fails. Since we used the

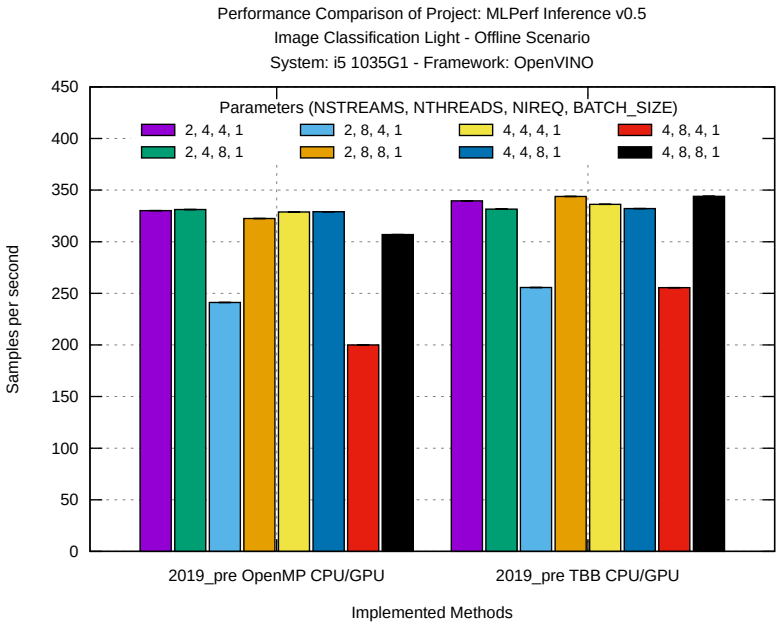



**Figure 12.20:** *MLPerf Inference v0.5: Image Classification lightweight, Offline scenario. Performance of OpenVINO 2019 pre-version built with OpenMP and TBB threading using a different parameters configurations running on CPU and GPU systems (see Reproduced Exp. 2, table 12.5) using code submitted by Intel[148].*

Windows OS version of the code, we had to use the arguments the same way they are used in the Linux version. Analysing the code used in the Dell submission, one can spot the same "wrong" function call, but we didn't test its behaviour since using the same Intel code for all our experiments.

Software version The version of OpenVINO to use is not clearly mentioned in the documentation attached to the submission. We run our initial tests using the final realease from 2019, i.e., R3.1. Only after figuring out the unexpected behaviour this version have on an older architecture (as discussed earlier in this chapter), we decided to use the pre-release of OpenVINO 2019, which was available at the time of submission and re-run all the experiments.

System priviledges The Intel submission for Linux systems, in the case of Single-Stream and Server scenarios, requires the execution of some OS command to clean shared memory, caches and configure intel_pstate. Those commands require root privileges which we don't have on any of the tested

devices in the datacenter category. Instead, we could run the commands to clean the memory when using our edge device while the pstate configuration cannot be applied because it results in an error. Anyway, the original experiments run on the Intel edge device was using a Windows OS and, thus, not considering the execution of those commands.

Number of executions While a valid submission to the MLPerf Inference Benchmark does not require multiple runs of a task, we re-executed at least 5 time each of our experiments. Since we found the results to be consistent from an execution to the other, we decided to include a single result for each experiment configuration for our study.

### 12.1.5 Performance considerations

Container overhead All our experiments run through containers which may produce an overhead. Based on past studies [151], we think this can't explain the performance discrepancies we have encountered during our experimentation.

Process pinning In its submission, Intel documented the usage of *numactl* util for binding the processes to the cores. We did not experience any improvement when using it. Setting *KMP_AFFINITY* did not produce any improvement either, with the best performance being obtained using the default configuration. Moreover, pinning is only used in the Single-Stream scenario while the performance constantly differs from the original results, including the Offline scenario.

CPU frequency governor The process governor for the datacenter machines was set on performance mode, and we could not test any different settings. While on our edge device tested, we could test both the performance and the default *ondemand* modes. The performance drops when forcing the performance mode, and, consequently, we always kept the settings to the ondemand mode. Since the edge device used for the original experiment was running a Windows OS, none of those settings was discussed or even mentioned.

## 12.2 MLPerf v0.7

There were no substantial changes in the newer versions of the MLPerf Inference benchmark, and setting up the correspondent PROVA! driver did not require much effort but updating the driver **descriptor**. The required changes are the creation of the methodTypes (software and scripts) by following the documentation provided by the submitters of which we want to reproduce the results.

The tasks included in the Intel and Dell submissions are the heavyweight version of image classification and object detection problems (table 12.6), but we will only cover the image classification one.

| System | Framework | Workloads | | | |
|---|---|---|---|---|---|
| | | Image Classification | | Object Detection | |
| | | ICL* | ICH | ODL | ODH |
| 1-node-2S-CLX | OpenVINO | — | S, O | — | — |
| 1-node-2S-CLX | MXNet | — | S, O | — | — |
| 1-node-2S-CLX | Tensorflow | — | S, O | — | — |
| R740xd8280 | OpenVINO | — | S, O | — | S, O |

* Deprecated since v0.7 of MLPerf Inference benchmark

**Table 12.6:** *MLPerf Inference Benchmark v0.7 scenarios submitted for the Intel and DELL EMC systems selected (section 11.1 and 11.2): Single-Stream (**SS**), Server (**S**) and Offline (**O**).*

Both the systems we consider are from the *datacenter* category and use either a single ML framework, like for the Dell submission with OpenVINO, or multiple ones, like for the Intel submission adopting OpenVINO, MXNet and TensorFlow. The detailed experiment stack is presented in figure 12.21.

### 12.2.1 Image Classification Heavyweight task

To reproduce the results of the image classification task submitted by Intel, we used a machine that happens to be the same as the original one, so even that is physically a different system, we can consider our experiment a **repetition** from which we expect to have full reproducibility of the performance results (table 12.7).

We configured the three ML frameworks used following the steps documented in the Intel submission for both the software configuration and installation[152] as well as the model calibration[153]. In terms of accuracy, our system configuration effort did not help to have the exact matching

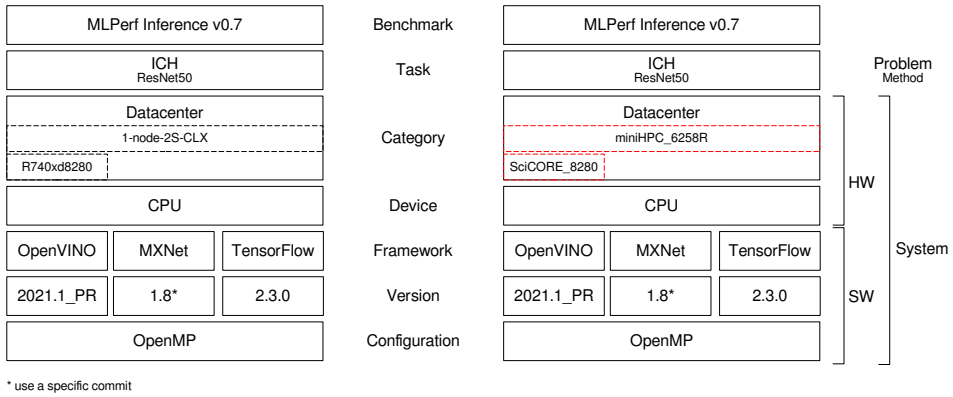



**Figure 12.21:** *MLPerf Inference v0.7 reproduction: High level view of the experiment elements we changed (red) for our reproduction. The method is defined by the benchmark itself and bound to the task.*

| | Original Experiment | Reproduced Experiment |
|---|---|---|
| **Problem** | Image Classification (Server/Offline scenario) [44]<br>Dataset: ImageNet [149] (224x224) | |
| **Method** | Deep Residual Learning [150]<br>Model: ResNet-50 v1.5 | |
| **System** | • Device: Xeon 6258R<br>• Framework: OpenVINO/MXNet/TensorFlow (Configuration 1-3) | • Device: Xeon 6258R (miniHPC)<br>• Framework: *Unchanged* |

**Table 12.7:** *Reproduction of OpenVINO experiments submitted by Intel as part of the MLPerf Inference Benchmark v0.7: image classification (heavyweight) task for Server and Offline scenarios. A detailed description of the systems is available in section 11.2.*

of the results, although extremely close (figure 12.22). The tiny differences should not affect the final performance results: still, we cannot be fully confident about the precision of the steps followed for the software and model configuration.

In the case of the Server scenario, whose results are shown in figure 12.23, our execution of the three frameworks precisely reproduced the original values. As for the original submission, OpenVINO overperforms the other two frameworks.

[‡‡]MLPerf v0.7 Inference ResNet-50 v1.5 Offline. Retrieved from https://mlcommons.org/en/inference-datacenter-07 19 March 2022, entries 0.7-100, 0.7-101 and 0.7-102. MLPerf name and logo are trademarks. See https://mlcommons.org for more information.

[§§]See footnote for Figure 12.22

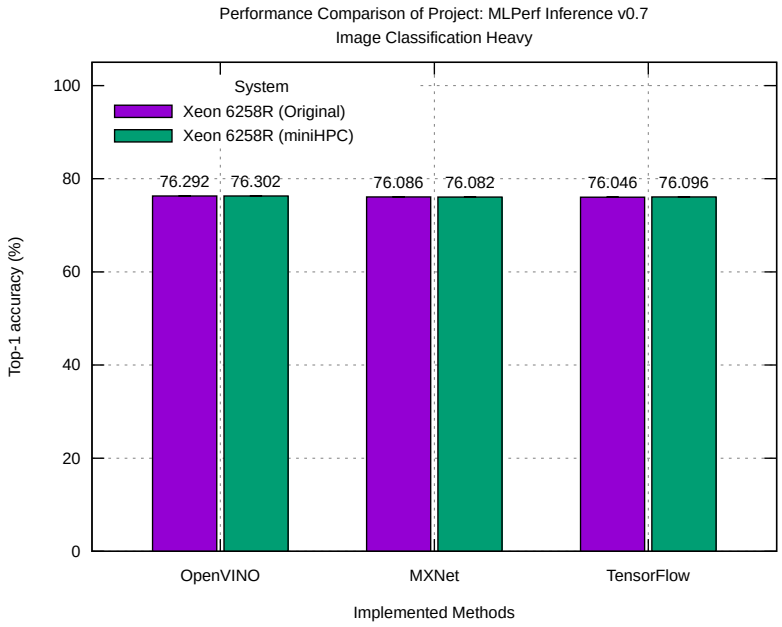



**Figure 12.22:** *MLPerf Inference v0.7: Image Classification heavyweight. Model accuracy of Intel submission[‡‡] compared to the experiment reproduction on our system (see Reproduced Experiment, table 12.7).*

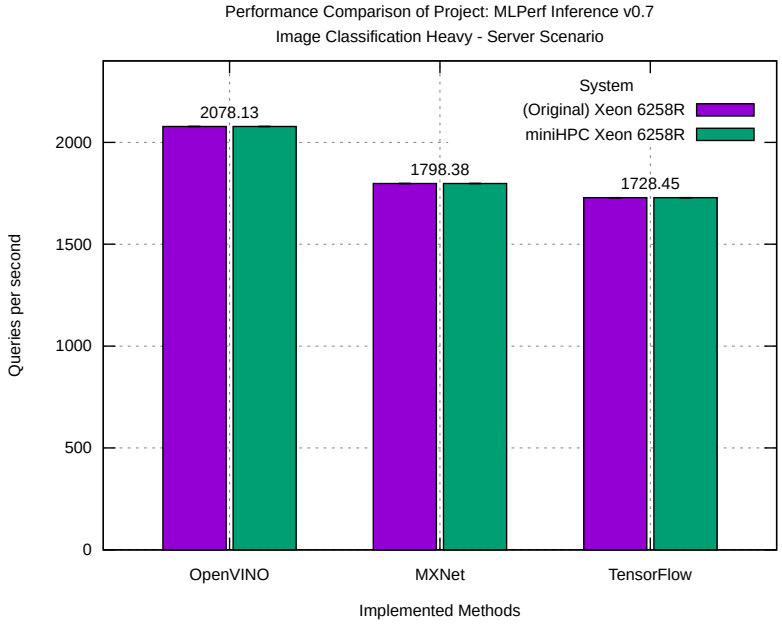



**Figure 12.23:** *MLPerf Inference v0.7: Image Classification heavyweight, Server scenario. Intel submission[§§] compared to the experiment reproduction on our system (see Reproduced Experiment, table 12.7).*

We could not obtain similar reproducible results for the Offline scenario (see figure 12.24): in such case, the performance is 7% to 10% lower.

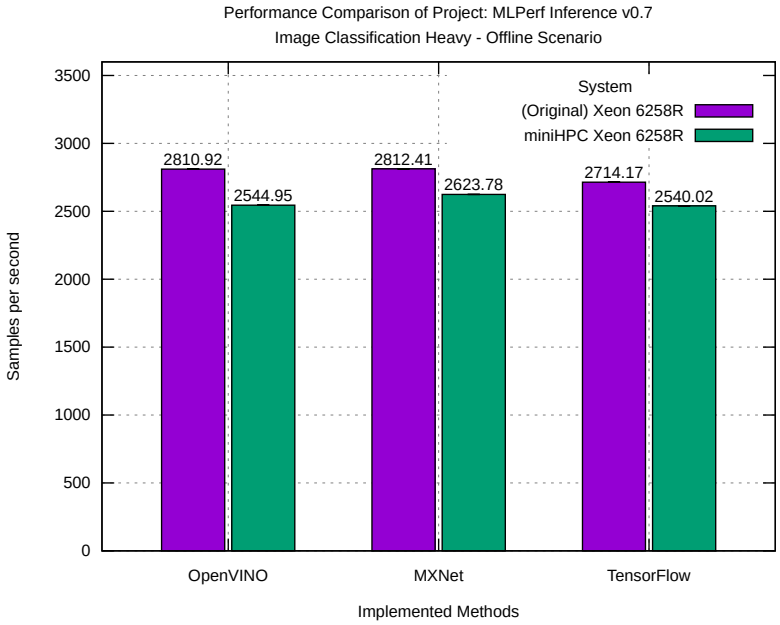



**Figure 12.24:** *MLPerf Inference v0.7: Image Classification heavyweight, Offline scenario. Intel submission*[¶¶] *compared to the experiment reproduction on our system (see Reproduced Experiment, table 12.7).*

If we did a ranking with the results from our experiments, we would notice the same positions for the three frameworks with MXNet in the first place and TensorFlow in the last like for the original results. However, the performance obtained by OpenVINO in the original experiment was very close to the MXNet ones, while in our execution, they are closer to the performance of TensorFlow.

Dell used the same Intel's OpenVINO code in its submission on a different machine which, like the previous case, we could use for an exact **repetition** (table 12.7).

Similarly to the Intel example, the accuracy we obtained is marginally different from the original, but it precisely confirms the previous value (figure 12.25c). It is worth specifying that we did not copy the previous model to test over to the new system but reproduced all the same steps from scratch on the second system.

As already happened with the Intel reproduction, the Server scenario experiment produced an exact match with the original performance (fig-

[¶¶]See footnote for Figure 12.22

| | Original Experiment | Reproduced Exp. 1 | Reproduced Exp. 2 |
|---|---|---|---|
| **Problem** | Image Classification (Server/Offline scenario) [44]<br>Dataset: ImageNet [149] (224x224) | | |
| **Method** | Deep Residual Learning [150]<br>Model: ResNet-50 v1.5 | | |
| **System** | • Device: Xeon 8280<br>• Framework: OpenVINO (Configuration 1) | • Device: Xeon 8280 (sciCORE)<br>• Framework: *Unchanged* | • Device: Xeon 8280 (sciCORE)<br>• Framework: OpenVINO (Configuration 5-6) |

**Table 12.8:** *Reproduction of OpenVINO experiments submitted by Dell as part of the MLPerf Inference Benchmark v0.7: image classification (heavyweight) task for Server and Offline scenarios. A detailed description of the systems is available in section 11.2.*

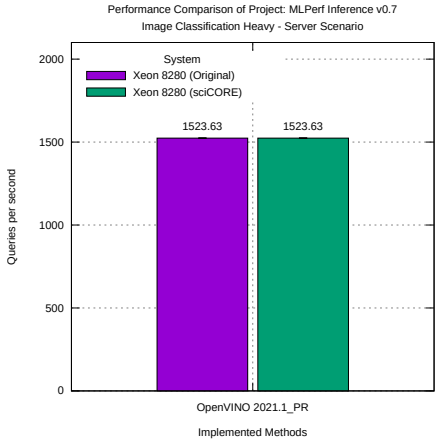
ure 12.25a). It is interesting how the Dell and Intel systems may have such a substantial variation in performance running this scenario (almost a 30%) while being similar in terms of the number and frequency of processors as well as cache and memory specifications.

Finally, we run the Offline scenario, achieving a performance that differs less than 1% from the original result (see figure 12.25b), meaning we can reproduce all the Dell experiments (accuracy, Server and Offline scenario).

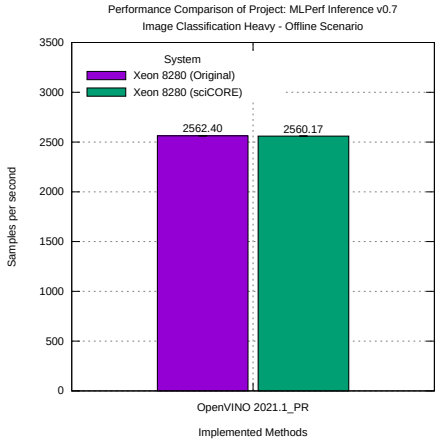
Analyzing the results, we notice that, unlike the Server scenario, the performance difference of the Dell and Intel systems is not so relevant. The Intel performance results show that the first system we benchmarked (Intel Xeon 6258R) produces results around 10% better than the other system (Intel Xeon 8280), and this difference becomes lower than 1% based on our experiments.

Another difference between the OpenVINO submissions are the experiment parameters: although in the Server scenario, the command used to run the experiment is the same for both systems, for the Offline scenario, there is a different workload "parallelization" approach. In the Dell submission, the number of streams, threads and requests is set to the amount of physical cores available, i.e. 56, and the batch size is kept to 1. Intel, instead, sets only the number of threads equal to the number of the physical cores while increasing the batch size to 4 and creating only half streams and requests compared to the Dell example, i.e. 28. Applying the same Intel strategy to the second system (Intel Xeon 8280), we fixed number of streams, threads and requests varying only the batch size to look at the performance be-

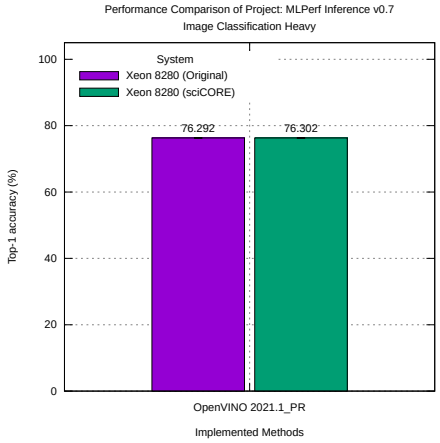
***MLPerf v0.7 Inference ResNet-50 v1.5 Offline. Retrieved from https://mlcommons.org/en/inference-datacenter-07 19 March 2022, entry 0.7-94. MLPerf name and logo are trademarks. See https://mlcommons.org for more information.




**(a)** *Performance comparison for Server scenario*




**(b)** *Performance comparison for Offline scenario*




**(c)** *Accuracy comparison*

**Figure 12.25:** *MLPerf Inference v0.7: Image Classification heavyweight. Dell submission*** *compared to the experiment reproduction on our system (see Reproduced Experiment, table 12.8).*

haviour. Furthermore, we tried not only the OpenMP implementation but also the TBB one.

Figure 12.26 shows how it is still the parameters' combination suggested by Intel that generates the best results, but in our case, it is the TBB variant of the OpenVINO software reaching the best overall performance. This last parameters combination reaches a performance of 2635 samples per second which, even though not a huge improvement, is better than the original Dell result.

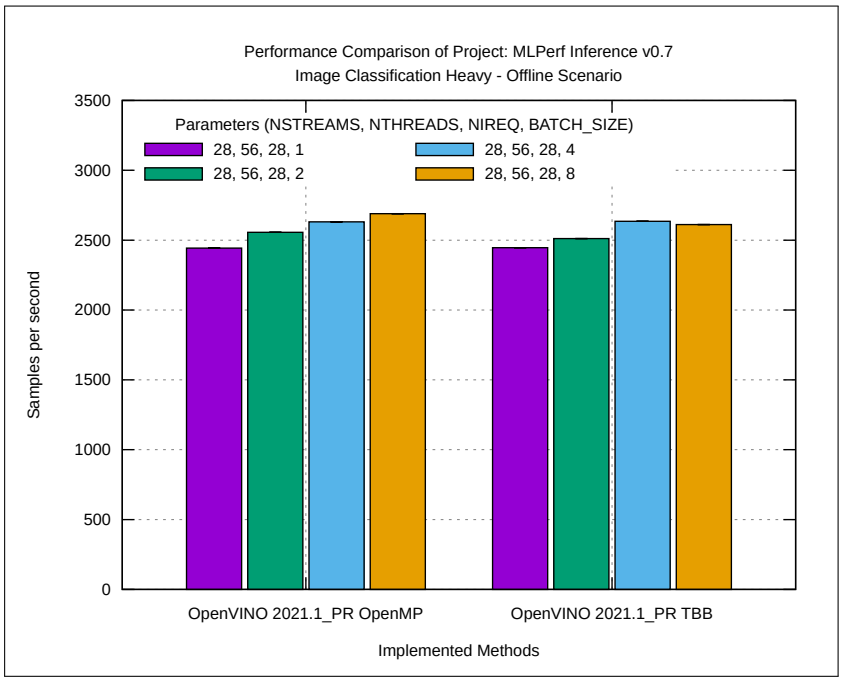



**Figure 12.26:** *MLPerf Inference v0.7: Image Classification heavyweight, Offline scenario. Performance comparison of OpenVINO 2021.1 pre-version configured with OpenMP and TBB threading using a variable batch size: running on our system (see Reproduced Exp. 2, table 12.8) using code submitted by Intel[152].*

## 12.2.2 Reproducibility considerations

Configuration effort Although the information for re-run the code submitted was more complete, the effort to prepare and execute the experiments was remarkable. Because of the simplicity and flexibility of the PROVA! approach, we could integrate all the software configurations in some methodTypes definition. Still, the lack of a common way of setting the experiments seems to be one of the obstacles to reproducibility.

Imprecise configuration Since we could not get the same results as in the original experiment, we think we may have some misconfiguration in our experiment. Except for these, during the configuration of the TensorFlow code, we found an error affecting the compilation, which is still present in the newer MLPerf Inference v1.0 submission [154]. After solving this problem, we could continue but found another issue with a missing configuration file[155]. We had this issue using the scripts provided in the submission version v0.7 (the one we wanted to reproduce). We only later figured out

that Intel updated the file in the sequent version v1.0, which we eventually used to execute the experiment.

Failing executions The scripts provided for the execution of the TensorFlow experiment create a "framework within the framework". Based on the parameters defined by the user, the execution script virtually splits the host in different instances, which are used either to send or elaborate requests. From our experience with it, we found this whole experiment architecture to be as nice as unstable. It was not easy to find all the valid parameters combinations, and, even for the valid ones, the executions were randomly failing (mainly for the Offline scenario).

### 12.2.3 Performance considerations

Performance vs configuration effort Configuring the three ML frameworks, i.e. OpenVINO, TensorFlow, MXNet, required a very different amount of work. Compiling a CPU-optimized **TensorFlow** is, on its own, time-consuming and difficult to automate in a CI/CD pipeline since a pipeline executor usually gets a couple of cores and a limited amount of memory as resources. On top of this, integrating into PROVA! the aforementioned complex scripts built by Intel was not quick, and the misconfigurations made it even more tedious. On the other hand, **MXNet** worked almost out of the box thanks to the precise instructions provided. Still, the steps included a few patches, and the calibration process was not extremely easy since it needed an extra step for the dataset preprocessing. From our tests, the easiest software to set up is **OpenVINO**. Having simple tools for conversion and calibration made all the steps trivial, and switching from one version to another was as easy as changing a line in the container recipe. It is essential to consider the configuration efforts when looking at the results. Not only did we get better accuracy and much better performance for the Server scenario, but even the slightly worse performance in the Offline scenario may not justify using one of the other frameworks.

## 12.3 MLPerf v1.1

At the time of writing, the last completed MLPerf Inference benchmark round is version v1.1. In our experiments, we are not reproducing any submission from it but use the new benchmark version to run the OpenVINO submission from MLPerf v0.7 and execute it with both the previous and the

most recent version of the framework. This experiment aims to show how we can port the old experiment on a newer PROVA! driver version and compare the evolution of the performance in terms of software enhancements.

### 12.3.1 Image Classification Heavyweight task

As for the other PROVA! driver version, updating it to the latest benchmark version took almost no effort and as well on the methodType side, we directly reuse the one prepared for the previous driver version with a mere copy/paste. Finally, having all the previous versions of OpenVINO built as a container, also building the newer OpenVINO version we want to compare against was truly straightforward. Table 12.9 presents the details of the system used for experimenting the version v1.1 of the MLPerf Inference benchmark.

| | **Experiment 1** | **Experiment 2** |
|---|---|---|
| **Problem** | Image Classification (Server/Offline scenario) [44]<br>Dataset: ImageNet [149] (224x224) | |
| **Method** | Deep Residual Learning [150]<br>Model: ResNet-50 v1.5 | |
| **System** | • Device: Xeon 6258R (miniHPC)<br>• Framework: OpenVINO (Configuration 3) | • Device: *Unchanged*<br>• Framework: OpenVINO (Configuration 4) |

**Table 12.9:** *OpenVINO experiments using MLPerf Inference Benchmark v1.1* PROVA! *driver: image classification (heavyweight) task for Offline scenario. A detailed description of the systems is available in section 11.2.*

The first check is for any change in accuracy using different versions of OpenVINO with the old and new models. Figure 12.27 shows a slight change that is not surprising, but it is curious to see that the model generated using the new OpenVINO version reaches the exact accuracy shown in the previous MLPerf benchmark submission (see figure 12.22).

Lastly, in the Offline scenario results, we can see how, on the one hand, the performance behaviour with different combinations of experiment parameters did not change and, on the other, the new version of OpenVINO brings a 2 to 6% improvement regardless of whether using the previous or current model.

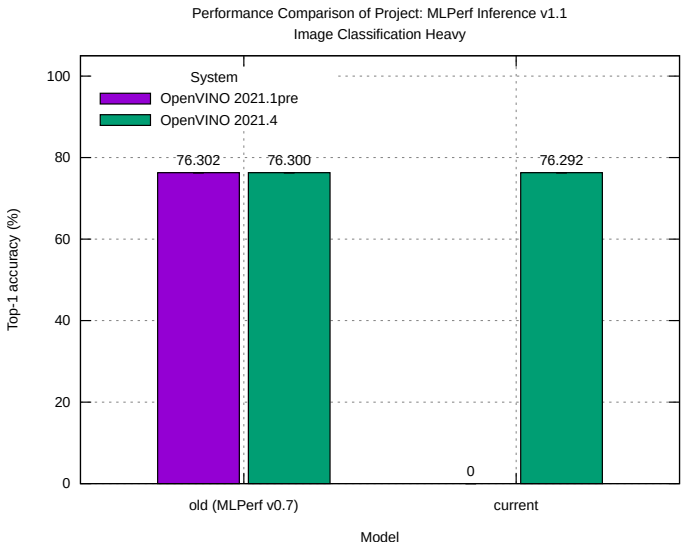



**Figure 12.27:** *MLPerf Inference v1.1: Image Classification heavyweight. Model accuracy of different OpenVINO versions running on our system (see Experiment 1, table 12.9) using code submitted by Intel[152].*

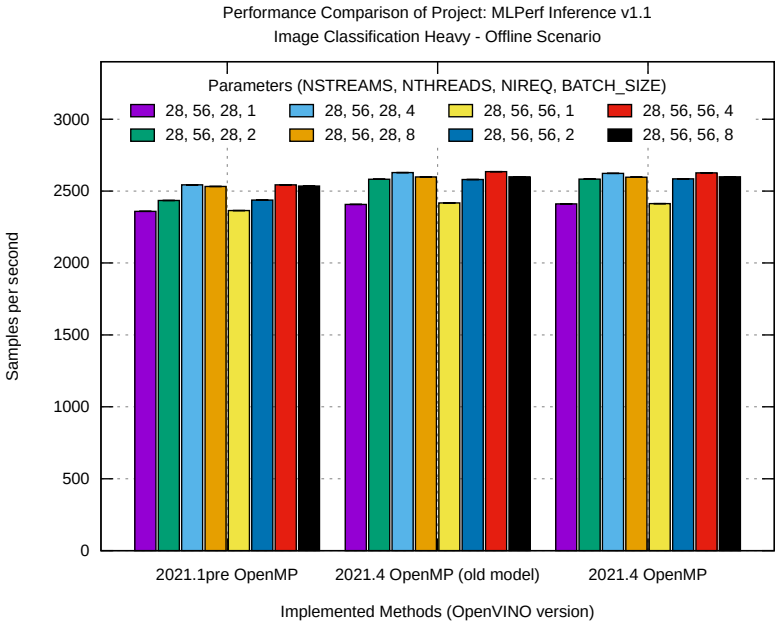



**Figure 12.28:** *MLPerf Inference v1.1: Image Classification heavyweight. Performance of different OpenVINO versions using a variable inference requests number and batch size: running on our system (see Experiment 2, table 12.9) using code submitted by Intel[152]. With "old model" we indicate the model built using the version 2021.1pre of OpenVINO.*

# Part V

# Conclusions and Future Work

# Chapter 13

# Conclusion and Future Work

There is a long debate around the end of Moore's law and whether it may be really at its end or just facing a slowdown. Nevertheless, the increasing request for computation for a fastly evolving field like machine learning is undebatable, same as the fact that the traditional CPU architecture struggles to provide such computational power. As machine learning devices and frameworks are improving to keep up with the increasing request for computation, likewise, all its experimentation needs to happen in a reproducible manner. Bearing in mind the fundamental importance of reproducibility of results, using these highly optimized ML-specific hardware architectures to carry on experiments whose performances are not reproducible is a waste of time, money and, above all, opportunities.

We discussed the importance of having a benchmark competition in the machine learning field to push for improvements like it happens for Linpack benchmark and high-performance computing. For this reason, we decided to perform a reproducibility study of the most prominent machine learning benchmark, i.e., MLPerf, taking into account both accuracy and performance. Our first try was to manually reproduce code written and documented by a domain expert from a "naive" user point of view. The main challenges came from manually recreating both the environment and the setup needed by the experiments. In fact, the documented instructions may look easy to the expert who wrote them but, obviously, more difficult to the user who sees them for the first time. If, on the one hand, it is clear that reproducing an experiment from two years before can always be challenging, on the other, it would be beneficial to build experiment's artifacts (code,

scripts, documentation) with reproducibility in mind to avoid specifying, for example, a link to the "latest" version of the documentation, which may create issues even after a few months. This is just an example of an easily avoidable bad practice that prevented us from having a smoothly running experiment and, instead, requested us additional effort to acquire extra knowledge about the tool we wanted to reproduce, even just to re-run an existing experiment.

Those challenges relate to our research question about performance reproducibility in machine learning. To overcome them and answer our question, we propose an experiment workflow tool: PROVA!. With the support of our experiment taxonomy, we could precisely describe the original experiments and eventual variations as *<Problem, Method, System>* 3-tuple, which maps directly into PROVA!. Using a tool like PROVA! indeed helps follow best practices regarding reproducible research. Still, the most challenging obstacle to overcome on the road to reproducible science is creating habits in scientists, emphasizing the importance of reproducibility and adding it to the curricula. There are already efforts in this direction like the one from the Center for Reproducible Science*(University of Zurich), whose mission is precisely to train the next generation of researchers in good research practices. A further example is given by The Turing Way handbook[156]: a collection of guides on reproducible research, project design, communication, collaboration and ethical research with the aim to "provide all the information that researchers and data scientists in academia, industry and the public sector need at the start of their projects to ensure that they are easy to reproduce at the end".

From the technical point of view, needing enough expertise to deal with novel devices and get the best performance out of them is unexceptional. In effect, having a solid base for an experiment does not mean the user will bypass acquiring the right expertise, but only that he can avoid wasting time trying to solve naive errors. Regardless, the manual configuration of software and experiments is an error-prone process even for the expert. We also had errors during our study, but by automating our workflow, we could reduce these errors in the first instance and sped up the "re-execution" of wrong experiments when necessary. Having specific software frameworks to manage DSAs is undoubted of great help, but this is usually not enough. In addition, users typically have different interests, which means they may be highly interested in using optimized code to solve their problems without

*https://www.crs.uzh.ch/en.html

necessarily caring about the detailed explanation of how the optimization is performed.

These thoughts motivated us to dig into the challenges one faces during the experimentation on HPC systems, showing the need for software, connection and workflow management. We implemented our solutions in PROVA! by extending the first version of the tool, adding the support for containerized software, which supplies an environment "as-Code", significantly sustaining reproducibility and representing implicit documentation. We also improved aspects like the interaction with the job scheduler and the experiment workflow management. Those updates allowed us to demonstrate how a scientist can configure an experiment to run on different systems and accelerators seamlessly. This ability to re-run an experiment built on a different system by a different person dramatically lowers the learning curve, boosts collaboration, and, thus, answers our research question.

Despite being created to support typical HPC applications, PROVA! can be used for running various applications thanks to its simple and flexible design. In the case of the MLPerf benchmark, we went a step further by adding the "driver" concept to manage the execution of predefined applications. With the PROVA! driver for the MLPerf Inference benchmark, we empowered an easy configuration of the benchmark applications with the possibility of adding ready-to-run experiments based on the benchmark submissions. Through PROVA!, we could compare the original accuracy and performance benchmark results against both the results obtained using the same experiment configuration and execution systems as well as against custom ones.

At the end of our experiments, we can say that we could re-run the benchmark experiment using the code and the instructions provided by the original submitter; this worked. However, many other aspects did not work, like reproducing the configuration steps, which required acquiring information from other submissions and examples not part of the benchmark or providing scripts with values that do not seem correct. As discussed in our reproducibility considerations of section 12.1.4, we also needed to produce some fixes that we assume are right, while we got no official confirmation. Those issues we found for the first benchmark version are absent in the following versions, but in section 12.2.2, we discussed other issues found in the newer versions. Even though we most likely found the right solution for those last issues, we cannot be sure we correctly reproduced all the experiments' configurations. We think that the effort required by a scientist who wants to reproduce an existing experiment should not be comparable with

the ones needed to produce the original one: this is not reflecting the situation we faced during our work. Instead, we could show how we automated the experiment configuration and execution using PROVA! as a driver to provide the basic blocks to configure and run the MLPerf benchmark. At the same time, the code developer builds his experiment defining a *methodType* in PROVA! (environment and scripts needed), which will be an easy task for the code expert. This way, a different user can reproduce the experiment by pressing a button in an interface and, most importantly, even build new experiments to get more insights and not just threaten someone else's code as a black box, which is the answer we provide to our last research question.

Besides the reproducibility of an experiment, we also need to consider the importance of the results generated and how researchers can use them. Recently, the Swissuniversities organization developed a national strategy and an action plan for Open Research Data (ORD) activities [157], funding initiatives for ORD practices by researchers across disciplines and higher education institutions in Switzerland [158]. They aim to support reproducible research findings by open access and reuse of research data. Nevertheless, it is fair to point out the importance of not only making data available but also defining the steps needed to regenerate them. If, on the one hand, reproducibility increases the credibility of the research, on the other, an independent research group that reproduces this data make it sounder or even richer when it performs variations of the initial experiment and contributes the new data generated back.

Finally, from a software engineering perspective, we demonstrated some PROVA! proprerties like the extensibility with the enhancements presented in section 9.1, the adaptability to the MLPerf benchmark support using the driver mode (see section 9.3), the portability, principally to Linux/Unix systems, using the container support (see section 9.2) and, in general, reusability.

The work accomplished in this thesis can be viewed as support to raise awareness of the importance of reproducibility for scientific research and, particularly, the machine learning field. Still, there are many possible extensions to it. We showed how the MLPerf PROVA! driver could be used to reproduce some of the results submitted. To experiment with all the submissions, a user will need access to several different systems, which is not practicable. Nevertheless, there are still numerous submissions that can be configured into the driver. Some of the latest submissions are leveraging other frameworks like Triton Inference Server [159] from NVidia, which will need to be tested in PROVA! to understand how it would be possible to in-

tegrate it. Moreover, the Inference benchmark has added other metrics like power/energy consumption which would be extremely important to test to cover ML inference on tiny devices.

Among the MLPerf benchmark suites, the Inference one is the less expensive in terms of computation and, thus, usually, execute on one node only. Instead, benchmarks like the MLPerf Training and the MLPerf HPC use a much larger amount of resources. Albeit PROVA! was already used in the past to run multi nodes HPC applications, it would be interesting to test how those new benchmarks integrate with PROVA!.

As discussed in section 6.2, the MLPerf community also started an official project to support the benchmark experiment reproducibility, i.e., MLCube™, while MLCommons took one of the other tools, i.e., Collective Knowledge under its umbrella. Those are, clearly, steps in the right direction and an opportunity for direct comparison and, possibly, integration with PROVA!

Extensively trying all the combinations of an experiment is not something a single user can do. Instead, it would be helpful to create a place where, as a community effort, people can store the results of their experiments. Other research groups can analyse those raw results and possibly get new/different insights.

In [160], we presented a possible architecture to interface PROVA! with different frontends based on the type of user accessing the tool. We created a prototype implementation of a Jupyter Notebook extension, but since the Jupyter community moves its default interface from the Notebook to the Lab, we will need to rework our PROVA! extension for Jupyter.

Ultimately, we want to reinforce that, to help machine learning research move forward, it is fundamental to foster collaboration, aiding reproducibility.

# Appendix A

# MLPerf containers

## A.1 MLPerf Inference v0.5: OpenVINO example

To build the environment needed to execute the OpenVINO experiments on the Intel GPU, we used several container as building blocks. We started from containers including the CPU dependencies (see Listing A.1) for OpenVINO and on top of this we add the Intel GPU support (see Listing A.2). Those first 2 containers are used to speed up the build of the actual OpenVINO container (see Listing A.3). Finally, we add the MLPerf Inference benchmark files (see Listing A.4), showing also how to run the OpenVINO model conversion as part of the container build process.

```
FROM ubuntu:18.04

ENV DEBIAN_FRONTEND="noninteractive" \
    LC_ALL="C.UTF-8" \
    LANG="C.UTF-8" \
    LANGUAGE="C.UTF-8" \
    OPENCV_VERSION="4.1.2" \
    PYTHON_VERSION="3.6"

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        git \
        cmake \
        build-essential \
        curl \
        unzip \
        ca-certificates \
        sudo \
        python${PYTHON_VERSION}-dev \
```

```
        python${PYTHON_VERSION}-distutils && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

RUN cd /usr/bin/ && \
    ln -s python3 python && \
    cd / && \
    curl https://bootstrap.pypa.io/get-pip.py -o /get-pip.py && \
    python${PYTHON_VERSION} /get-pip.py && \
    rm /get-pip.py && \
    python${PYTHON_VERSION} -m pip install \
        numpy \
        cython && \
    curl -LO https://github.com/opencv/opencv/archive/${OPENCV_VERSION}.zip
         && \
    unzip -q ${OPENCV_VERSION}.zip && \
    rm ${OPENCV_VERSION}.zip && \
    cd /opencv-${OPENCV_VERSION} && \
    mkdir build && cd build && \
    cmake \
        -DPYTHON_EXECUTABLE="/usr/bin/python${PYTHON_VERSION}" \
        -DPYTHON3_LIBRARY="/usr/lib/x86_64-linux-gnu/libpython${
            PYTHON_VERSION}m.so" \
        -DPYTHON3_INCLUDE_DIR="/usr/include/python${PYTHON_VERSION}" \
        -DCMAKE_INSTALL_PREFIX="/opt/opencv" \
        .. && \
    cmake --build . && make install

ENV OpenCV_DIR="/opt/opencv/lib/cmake/opencv4"
```

**Listing A.1:** *Dockerfile with OpenVINO CPU dependencies*

```
FROM provarepro/openvino:2019_c_deps-ubuntu18

ARG BASE_URL="https://github.com/intel/compute-runtime/releases/download" \
    VER="19.41.14441"

#Install Intel Graphics Compute Runtime for OpenCL Driver package
    19.04.12237.
RUN apt-get update && \
    apt-get install -y --no-install-recommends ocl-icd-libopencl1 && \
    rm -rf /var/lib/apt/lists/* && \
    mkdir /neo && cd /neo && \
    curl -LO /intel-gmmlib_19.3.2_amd64.deb && \
    curl -LO ${BASE_URL}/${VER}/intel-igc-core_1.0.2597_amd64.deb && \
    curl -LO ${BASE_URL}/${VER}/intel-igc-opencl_1.0.2597_amd64.deb && \
    curl -LO ${BASE_URL}/${VER}/intel-opencl_19.41.14441_amd64.deb && \
    curl -LO ${BASE_URL}/${VER}/intel-ocloc_19.41.14441_amd64.deb && \
    sudo dpkg -i *.deb && \
    ldconfig && \
    cd / && rm -rf /neo
```

**Listing A.2:** *Dockerfile with Intel GPU support for OpenVINO*

```
FROM openvino/ubuntu18_runtime:2019_R3.1 as runtime
FROM provarepro/openvino:2019_cg_deps-ubuntu18
```

```
ARG OV_REL="releases/2019/pre-release"

RUN git clone \
        --depth 1 \
        --single-branch \
        -b ${OV_REL} \
        https://github.com/openvinotoolkit/openvino.git && \
    cd /openvino && \
    git submodule update --init --recursive && \
    cd inference-engine && \
    ./install_dependencies.sh && \
    cd /usr/bin/ && rm python && \
    ln -s python3 python && \
    cd /openvino/inference-engine && \
    mkdir build && cd build && \
    cmake \
        -DCMAKE_BUILD_TYPE=Release \
        -DTHREADING=OMP \
        -DENABLE_DLIA=OFF \
        -DENABLE_VPU=OFF \
        -DENABPP=OFF \
        -DENABLE_PROFILING_ITT=OFF \
        -DENABLE_VALIDATION_SET=OFF \
        -DENABLE_TESTS=OFF \
        -DENABLE_GNA=OFF \
        -DENABLE_CLDNN=ON \
        -DENABLE_MKL_DNN=ON \
        -DENABLE_OPENCV=OFF \
        .. && \
    make --jobs=$(nproc --all) && \
    sed -i '/<plugins>/a \        <plugin name="MULTI" location="
        libMultiDevicePlugin.so">\n        </plugin>' \
           /openvino/inference-engine/bin/intel64/Release/lib/plugins.xml

#Copy MULTI plugin from official release
COPY --from=runtime \
         /opt/intel/openvino/deployment_tools/inference_engine/lib/intel64/
             libMultiDevicePlugin.so \
         /openvino/inference-engine/bin/intel64/Release/lib/
             libMultiDevicePlugin.so

ENV LD_LIBRARY_PATH="/openvino/inference-engine/temp/omp/lib/:/opt/opencv/
    lib:/openvino/inference-engine/bin/intel64/Release/lib" \
    InferenceEngine_DIR="/openvino/inference-engine/build"

# Creating user openvino
RUN useradd -ms /bin/bash -G users openvino && \
    chown openvino -R /home/openvino

USER openvino

CMD ["/bin/bash"]
```

**Listing A.3:** *Dockerfile for the final OpenVINO version 2019_pre container with GPU support built using OpenMP threading*

```
FROM provarepro/openvino:2019_pre-release_cg_omp-py36-gcc75-ubuntu18 as
    base
```

```
FROM openvino/ubuntu18_dev:2019_R3.1 as builder
WORKDIR /tmp

ARG BASE_URL="https://zenodo.org/record"
RUN curl -O ${BASE_URL}/3401714/files/
    ssd_mobilenet_v1_quant_ft_no_zero_point_frozen_inference_graph.pb && \
    curl -O ${BASE_URL}/3252084/files/mobilenet_v1_ssd_8bit_finetuned.tar.
        gz && \
    tar xf mobilenet_v1_ssd_8bit_finetuned.tar.gz && \
    rm mobilenet_v1_ssd_8bit_finetuned.tar.gz && \
    cp mobilenet_v1_ssd_finetuned/pipeline.config . && \
    rm -rf mobilenet_v1_ssd_finetuned && \
    python3 /opt/intel/openvino/deployment_tools/model_optimizer/mo.py \
        --input_model /tmp/
            ssd_mobilenet_v1_quant_ft_no_zero_point_frozen_inference_graph.
            pb \
        --input_shape [1,300,300,3] \
        --tensorflow_use_custom_operations_config /opt/intel/openvino/
            deployment_tools/model_optimizer/extensions/front/tf/
            ssd_v2_support.json \
        --tensorflow_object_detection_api_pipeline_config /tmp/pipeline.
            config

FROM base
USER root
WORKDIR /

# Install Boost
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        cmake \
        build-essential \
        git \
        wget \
        libicu-dev \
        libbz2-dev \
        liblzma-dev && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

ENV BOOST_VERSION="1.71.0" \
    _BOOST_VERSION="1_71_0"
ARG BASE_URL="https://boostorg.jfrog.io/artifactory/main/release"
RUN wget -q ${BASE_URL}/${BOOST_VERSION}/source/boost_${_BOOST_VERSION}.tar
    .gz && \
    tar xf boost_${_BOOST_VERSION}.tar.gz && \
    cd boost_${_BOOST_VERSION} && \
    ./bootstrap.sh --with-libraries=system && \
    ./b2 --with-system install && \
    cd / && rm -rf boost_*

RUN python${PYTHON_VERSION} -m pip install --ignore-installed --no-cache-
    dir \
        absl-py \
        pybind11 && \
    git clone \
        --recurse-submodules \
        --single-branch \
```

```
        -b r0.5 \
        https://github.com/mlcommons/inference.git /mlperf_inference && \
    cd /mlperf_inference && \
    git config --global user.email "antonio.maffia@gmail.com" && \
    git config --global user.name "fenz" && \
    git pull --no-commit --force origin pull/502/head && \
    git pull --no-commit origin pull/482/head && \
    git commit -m "merge PRs" && \
    mkdir loadgen/build && cd loadgen/build && \
    cmake .. && cmake --build . && \
    cp libmlperf_loadgen.a .. && \
    rm -r /mlperf_inference/loadgen/build && \
    cp -r /mlperf_inference/loadgen /mlperf_loadgen && \
    rm -rf /mlperf_inference

RUN CODE_PATH="closed/Intel/code/ssd-small/openvino-linux" && \
    git clone \
        --depth 1 \
        -b code \
        --single-branch \
        https://github.com/fenz-org/mlperf_inference_results_v0.5.git
            inference_results_v0.5 && \
    mv inference_results_v0.5/${CODE_PATH} /mlperf_inference && \
    rm -rf inference_results_v0.5

WORKDIR /mlperf_inference
RUN mkdir build && cd build && \
    cmake \
        -DLOADGEN_DIR=/mlperf_loadgen \
        -DIE_SRC_DIR=${InferenceEngine_DIR}/../src \
        -DBOOST_SYSTEM_LIB=/usr/local/lib/libboost_system.so \
        -DCMAKE_BUILD_TYPE=Release \
        .. && \
    cmake --build . --config Release

COPY --from=builder \
         /tmp/
             ssd_mobilenet_v1_quant_ft_no_zero_point_frozen_inference_graph
             .xml \
         /mlperf_inference/model/ssd-mobilenet.xml

COPY --from=builder \
         /tmp/
             ssd_mobilenet_v1_quant_ft_no_zero_point_frozen_inference_graph
             .bin \
         /mlperf_inference/model/ssd-mobilenet.bin

COPY --from=builder \
         /tmp/
             ssd_mobilenet_v1_quant_ft_no_zero_point_frozen_inference_graph
             .mapping \
         /mlperf_inference/model/ssd-mobilenet.mapping

USER openvino
```

**Listing A.4:** *Dockerfile OpenVINO 2019_pre with MLPerf Inference benchmark v0.5*

## A.2 MLPerf Inference v0.7: OpenVINO example

In the version v0.7 of MLPerf Inference benchmark we use the OpenVINO version 2021.1_pre . We starting building the container with CPU dependencies (see Listing A.5) and we compile OpenVINO on top of it (see Listing A.6) using TBB threading configuration. This can be done changing the line with the *-DTHREADING* cmake option. The final MLPerf Inference benchmark version v0.7 with OpenVINO is built using the Dockerfile in Listing A.7. We added the *Gflags* build as documented by Intel.

```
FROM ubuntu:18.04

SHELL ["/bin/bash", "-xo", "pipefail", "-c"]

ENV DEBIAN_FRONTEND="noninteractive" \
    LC_ALL="C.UTF-8" \
    LANG="C.UTF-8" \
    LANGUAGE="C.UTF-8" \
    OPENCV_VERSION="4.1.2" \
    CMAKE_VERSION="3.17.2" \
    PYTHON_VERSION="3.6"

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        git \
        build-essential \
        curl \
        unzip \
        ca-certificates \
        sudo \
        python${PYTHON_VERSION}-dev \
        python${PYTHON_VERSION}-distutils && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

ARG OV_VER="releases/2021/1.pre"
RUN git clone \
        --depth 1 \
        --single-branch \
        -b ${OV_VER} \
        https://github.com/openvinotoolkit/openvino.git && \
    cd /openvino && \
    git submodule update --init --recursive && \
    ./install_dependencies.sh

RUN apt-get purge -y cmake && \
    rm -rf /var/lib/apt/lists/*

RUN cd /usr/bin/ && rm python && \
    ln -s python3 python && \
    cd / && \
    curl https://bootstrap.pypa.io/get-pip.py -o /get-pip.py && \
    python${PYTHON_VERSION} /get-pip.py && \
```

```
    rm /get-pip.py && \
    python${PYTHON_VERSION} -m pip install \
        numpy \
        cython \
        cmake==${CMAKE_VERSION} && \
    curl -LO https://github.com/opencv/opencv/archive/${OPENCV_VERSION}.zip
         && \
    unzip -q ${OPENCV_VERSION}.zip && \
    rm ${OPENCV_VERSION}.zip && \
    cd /opencv-${OPENCV_VERSION} && \
    mkdir build && cd build && \
    cmake \
        -DPYTHON_EXECUTABLE=/usr/bin/python${PYTHON_VERSION} \
        -DPYTHON3_LIBRARY=/usr/lib/x86_64-linux-gnu/libpython${
            PYTHON_VERSION}m.so \
        -DPYTHON3_INCLUDE_DIR=/usr/include/python${PYTHON_VERSION} \
        -DCMAKE_INSTALL_PREFIX=/opt/opencv \
        .. && \
    cmake --build . && make install && \
    rm -rf /opencv-${OPENCV_VERSION}

ENV OpenCV_DIR="/opt/opencv/lib/cmake/opencv4"
```

**Listing A.5:** *Dockerfile OpenVINO version 2021.1_pre CPU dependencies*

```
FROM provarepro/openvino:2021.1pre_c_deps-ubuntu18

RUN cd /openvino && \
    mkdir build && cd build && \
    cmake \
        -DCMAKE_BUILD_TYPE=Release \
        -DENABLE_PYTHON=ON \
        -DPYTHON_EXECUTABLE=/usr/bin/python${PYTHON_VERSION} \
        -DPYTHON_LIBRARY=/usr/lib/x86_64-linux-gnu/libpython${
            PYTHON_VERSION}m.so \
        -DPYTHON_INCLUDE_DIR=/usr/include/python${PYTHON_VERSION} \
        -DENABLE_OPENCV=OFF \
        -DENABLE_VPU=OFF \
        -DENABLE_CLDNN=OFF \
        -DENABLE_GNA=OFF \
        -DENABLE_TESTS=OFF \
        -DTHREADING=TBB \
        **-DNGRAPH_ONNX_IMPORT_ENABLE=OFF \
        -DNGRAPH_DEPRECATED_ENABLE=FALSE** \
        .. && \
    make --jobs=$(nproc --all)

ENV LD_LIBRARY_PATH="/opt/opencv/lib:/openvino/bin/intel64/Release/lib" \
    InferenceEngine_DIR="/openvino/build"

# Creating user openvino
RUN useradd -ms /bin/bash -G users openvino && \
    chown openvino -R /home/openvino

USER openvino
CMD ["/bin/bash"]
```

**Listing A.6:** *Dockerfile for the final OpenVINO version 2019_pre container with GPU support built using TBB threading*

```
ARG OV_VER="2021.1pre"
ARG HW_VER="c"
ARG THREAD_VER="tbb"
ARG PY_VER="py36"
ARG GCC_VER="gcc75"
ARG BASEOS_VER="ubuntu18"

FROM provarepro/openvino:${OV_VER}_${HW_VER}_${THREAD_VER}-${PY_VER}-${
    GCC_VER}-${BASEOS_VER}

USER root
WORKDIR /

# Build Gflags
RUN git clone https://github.com/gflags/gflags.git && \
    mkdir gflags/build && cd gflags/build && \
    cmake .. && make

ENV gflags_DIR="/gflags/build"

# Install Boost
# Build Boost-Filesystem
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        cmake \
        build-essential \
        git \
        wget \
        libicu-dev \
        libbz2-dev \
        liblzma-dev && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

ENV BOOST_VERSION="1.72.0" \
    _BOOST_VERSION="1_72_0"
ARG BASE_URL="https://boostorg.jfrog.io/artifactory/main/release"
RUN wget -q ${BASE_URL}/${BOOST_VERSION}/source/boost_${_BOOST_VERSION}.tar
    .gz && \
    tar xf boost_${_BOOST_VERSION}.tar.gz && \
    cd boost_${_BOOST_VERSION} && \
    ./bootstrap.sh --with-libraries=system && \
    ./b2 --with-filesystem

ENV BOOST_DIR="/boost_${_BOOST_VERSION}"

# Build MLPerf LoadGen
ARG MLPERF_LOADGEN_VER="r0.7"
RUN python${PYTHON_VERSION} -m pip install --ignore-installed --no-cache-
    dir \
        absl-py \
        pybind11
```

```
RUN git clone \
        --recurse-submodules \
        --single-branch \
        -b ${MLPERF_LOADGEN_VER} \
        https://github.com/mlcommons/inference.git /mlperf_inference && \
    cd /mlperf_inference && \
    git checkout cf15214 && \
    mkdir loadgen/build && cd loadgen/build && \
    cmake .. && cmake --build . && \
    cp libmlperf_loadgen.a .. && \
    rm -r /mlperf_inference/loadgen/build && \
    cp -r /mlperf_inference/loadgen /mlperf_loadgen && \
    rm -rf /mlperf_inference

USER openvino
```

**Listing A.7:** *Dockerfile OpenVINO 2021.1_pre with MLPerf Inference benchmark v0.7*

# Bibliography


[1] OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute. [Online; accessed 07-March-2021]. [cited at p. 1, 17]

[2] Antonio Maffia, Helmar Burkhart, and Danilo Guerrera. Reproducibility in Practice: Lessons Learned from Research and Teaching Experiments. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 592–603, Cham, 2015. Springer International Publishing. [cited at p. 2, 4, 55]

[3] D. Guerrera, A. Maffia, and H. Burkhart. Reproducible stencil compiler benchmarks using PROVA! *Future Generation Computer Systems*, 92:933–946, 2019. [cited at p. 2, 4, 57]

[4] Antonio Maffia. Reproducing ML benchmarks: What works what doesn't work, 2023. Under Review. [cited at p. 4]

[5] D. Guerrera, H. Burkhart, and A. Maffia. Reproducible Stencil Compiler Benchmarks Using PROVA! In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 108–115. IEEE, 2016. [cited at p. 4, 33]

[6] H. Burkhart, D. Guerrera, and A. Maffia. No more Believe Me : Make Your Informatics Experiments Reproducible. http://www.informatics-europe.org/images/ECSS/ECSS2015/ECCS2015-Burkhart.pdf, 2015. Presented as a poster in the 11th European Computer Science Summit (ECSS 2015). [cited at p. 4]

[7] Helmar Burkhart, Danilo Guerrera, and Antonio Maffia. Trusted High-Performance Computing in the Classroom. In *Proceedings of the Workshop*

*on Education for High-Performance Computing*, EduHPC '14, page 27–33. IEEE Press, 2014. [cited at p. 5]

[8] Danilo Guerrera, Helmar Burkhart, and Antonio Maffia. Reproducible Experiments in Parallel Computing: Concepts and Stencil Compiler Benchmark Study. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesus Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 464–474, Cham, 2014. Springer International Publishing. [cited at p. 5, 32, 33, 46]

[9] J. L. Hennessy and D. A. Patterson. A New Golden Age for Computer Architecture. *Commun. ACM*, 62(2):48–60, January 2019. [cited at p. 9, 14]

[10] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. [cited at p. 9]

[11] Wikipedia contributors. Transistor count — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Transistor%20count&oldid=1005210655, 2021. [Online; accessed 07-February-2021]. [cited at p. 10]

[12] F. Peper. The End of Moore's Law: Opportunities for Natural Computing? *New Generation Computing*, 35(3):253–269, Jul 2017. [cited at p. 10]

[13] P. Ye, T. Ernst, and M. V. Khare. The last silicon transistor: Nanosheet devices could be the final evolutionary step for Moore's Law. *IEEE Spectrum*, 56(8):30–35, 2019. [cited at p. 10, 11]

[14] R. M. Koduri. No Transistor Left Behind. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–87, 2020. [Conference keynote]. [cited at p. 11, 16]

[15] The International Roadmap for Devices and Systems. 2020 Update: More Moore. Available at https://irds.ieee.org/images/files/pdf/2020/2020IRDS_MM.pdf, 2020. [Online; accessed 04-January-2021]. [cited at p. 11]

[16] M. Lapedus. Big Trouble At 3nm. https://semiengineering.com/big-trouble-at-3nm/, Jun 2018. [Online; accessed 06-February-2021]. [cited at p. 11]

[17] G. E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. [cited at p. 11]

[18] C. Mack. The Multiple Lives of Moore's Law. *IEEE Spectrum*, 52(4):31–31, 2015. [cited at p. 11]

[19] P. Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980. [cited at p. 11]

[20] J. Preskill. Quantum computing and the entanglement frontier, 2012. [cited at p. 11]

[21] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, Oct 2019. [cited at p. 11]

[22] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, and R. Wisnieff. Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits, 2019. [cited at p. 11]

[23] J. Shalf. The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2166):20190061, 2020. [cited at p. 11]

[24] General Electric. *GE-205/215/225 Auxiliary Arithmetic Unit*, Jan 1964. Available at http://www.bitsavers.org/www.computer.museum.uq.edu.au/pdf/CPB-325A%20GE225%20Auxiluary%20Arithmetic%20Unit.pdf, Rev. March 1965, [Online; accessed 04-January-2021]. [cited at p. 12]

[25] N. Thompson and S. Spanuth. The Decline of Computers As a General Purpose Technology: Why Deep Learning and the End of Moore's Law are Fragmenting Computing. Available at SSRN: https://ssrn.com/abstract=3287769, [Online; accessed 23-January-2021], Nov 2018. [cited at p. 12, 16]

[26] R. M. Russell. The CRAY-1 Computer System. *Commun. ACM*, 21(1):63–72, January 1978. [cited at p. 13]

[27] Jen-Hsun Huang. 2009: The GPU computing tipping point. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–29, 2009. [cited at p. 13]

[28] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. [cited at p. 13]

[29] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), 2020. [cited at p. 14]

[30] TOP500, The List. https://www.top500.org, Nov 2020. [Online; accessed 18-February-2021]. [cited at p. 15]

[31] G. Chrysos. Intel® Xeon Phi coprocessor (codename Knights Corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31, 2012. [cited at p. 16]

[32] N. Ali, D. Bradford, S. Chinthamani, J. Corbal, A. Hassan, and K. Janik. Knights Mill: Intel Xeon Phi Processor for Machine Learning. In *2017 IEEE Hot Chips 29 Symposium (HCS)*, August 2017. [cited at p. 16]

[33] Intel. Intel Acquires Artificial Intelligence Chipmaker Habana Labs. https://newsroom.intel.com/news-releases/intel-ai-acquisition, Dec 2019. [Online; accessed 06-March-2021]. [cited at p. 16]

[34] D. E. Shaw, J. P. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, A. Forte, J. Gagliardo, G. Gill, B. Greskamp, C. R. Ho, D. J. Ierardi, L. Iserovich, J. S. Kuskin, R. H. Larson, T. Layman, L. Lee, A. K. Lerer, C. Li, D. Killebrew, K. M. Mackenzie, S. Y. Mok, M. A. Moraes, R. Mueller, L. J. Nociolo, J. L. Peticolas, T. Quan, D. Ramot, J. K. Salmon, D. P. Scarpazza, U. B. Schafer, N. Siddique, C. W. Snyder, J. Spengler, P. T. P. Tang, M. Theobald, H. Toma, B. Towles, B. Vitale, S. C. Wang, and C. Young. Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53, 2014. [cited at p. 16]

[35] John McCarthy. What is AI? Available at http://jmc.stanford.edu/articles/whatisai/whatisai.pdf, 2007. [Online; accessed 04-February-2022]. [cited at p. 19]

[36] McKenna Fitzgerald, Aaron Boddy, and Seth D. Baum. 2020 survey of artificial general intelligence projects for ethics, risk, and policy. Technical Report 20-1, Global Catastrophic Risk Institute, 2020. [cited at p. 19]

[37] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. [cited at p. 20]

[38] Frank Rosenblatt. The perceptron - a perceiving and recognizing automaton. Technical Report 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, January 1957. [cited at p. 21]

[39] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982. [cited at p. 21]

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986. [cited at p. 21]

[41] Carver Mead and Mohammed Ismail, editors. *Analog VLSI Implementation of Neural Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer / Springer US, 1989. [cited at p. 21]

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017. [cited at p. 22]

[43] Simone Bianco, Remi Cadene, Luigi Celona, and Paolo Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018. [cited at p. 22]

[44] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark, 2020. [cited at p. 22, 37, 38, 85, 111, 113, 114, 120, 124, 128, 134, 137, 141]

[45] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training, 2019. [cited at p. 24]

[46] Intel. 3rd Gen Intel Xeon Scalable processors. Product Brief, 2021. [cited at p. 24]

[47] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 The A100 Datacenter GPU and Ampere Architecture. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 48–50, 2021. [cited at p. 24]

[48] Paresh Kharya. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format, May 2020. [Online; accessed 21-March-2022]. [cited at p. 25]

[49] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the Graphcore IPU architecture via microbenchmarking, 2019. [cited at p. 25]

[50] Cerebras. Cerebras systems: Achieving industry best AI performance through a systems approach. Technical report, Cerebras, April 2021. [cited at p. 25]

[51] Tensorflow. API documentation. https://www.tensorflow.org/versions/r2.8/api_docs. [Online; accessed 11-February-2022]. [cited at p. 28]

[52] PyTorch. PyTorch documentation. https://pytorch.org/docs/1.10/index.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[53] ONNX Runtime. ORT API docs. https://onnxruntime.ai/docs/api/. [Online; accessed 11-February-2022]. [cited at p. 28]

[54] ONNX Runtime. ONNX runtime execution providers. https://onnxruntime.ai/docs/execution-providers/. [Online; accessed 11-February-2022]. [cited at p. 28]

[55] ONNX Runtime. ONNX runtime for training. https://onnxruntime.ai/docs/#onnx-runtime-for-training. [Online; accessed 11-February-2022]. [cited at p. 28]

[56] OpenVINO. API reference. https://docs.openvino.ai/2021.4/api/api_reference.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[57] Nvidia. TensorRT documentation. https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-823/api/index.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[58] Theano. API documentation. https://theano-pymc.readthedocs.io/en/latest/library/index.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[59] Aesara. https://aesara.readthedocs.io/en/latest/, 2022. [Online; accessed 11-February-2022]. [cited at p. 28]

[60] Facebook. Caffe2 C++ and Python APIs. https://caffe2.ai/docs/api-intro.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[61] Preferred Networks. Chainer Docs - API Reference. https://docs.chainer.org/en/v7.8.1/reference/index.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[62] Apache. MXNet Docs. https://mxnet.apache.org/versions/1.9.0/api. [Online; accessed 11-February-2022]. [cited at p. 28]

[63] Microsoft. CNTK Library API. https://docs.microsoft.com/en-us/cognitive-toolkit/CNTK-Library-API. [Online; accessed 11-February-2022]. [cited at p. 28]

[64] MindSpore. MindSpore API. https://www.mindspore.cn/api/en/0.1.0-alpha/index.html. [Online; accessed 11-February-2022]. [cited at p. 28]

[65] ONNX. ONNX about. https://onnx.ai/about.html. [Online; accessed 17-February-2022]. [cited at p. 27]

[66] Alexander Aarts, Joanna Anderson, Christopher Anderson, Peter Attridge, Angela Attwood, Jordan Axt, Molly Babel, Štěpán Bahník, Erica Baranski, Michael Barnett-Cowan, Elizabeth Bartmess, Jennifer Beer, Raoul Bell, Heather Bentley, Leah Beyan, Grace Binion, Denny Borsboom, Annick Bosch, Frank Bosco, and Mike Penuliar. Estimating the reproducibility of psychological science. *Science*, 349, 08 2015. [cited at p. 31]

[67] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018. [cited at p. 31]

[68] Saeed S. Alahmari, Dmitry B. Goldgof, Peter R. Mouton, and Lawrence O. Hall. Challenges for the repeatability of deep learning models. *IEEE Access*, 8:211860–211868, 2020. [cited at p. 31]

[69] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr 2018. [cited at p. 31, 33, 34]

[70] Edward Raff. A step toward quantifying independently reproducible machine learning research, 2019. [cited at p. 31]

[71] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program), 2020. [cited at p. 31]

[72] HPL - A portable implementation of the high-performance Linpack Benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/, 2008. [cited at p. 32, 36]

[73] Association for Computing Machinery. Artifact Review Badging. https://www.acm.org/publications/policies/artifact-review-badging, 2018. [cited at p. 33]

[74] Michael A. Heroux, Lorena Barba, Manish Parashar, Victoria Stodden, and Michela Taufer. Toward a Compatible Reproducibility Taxonomy for Computational and Computing Sciences. 10 2018. [cited at p. 33]

[75] National Information Standards Organization. *NISO RP-31-2021, Reproducibility Badging and Definitions*. National Information Standards Organization (NISO), 3600 Clipper Mill Road, Suite 302Baltimore, MD 21211, 01 2021. [cited at p. 33]

[76] Karl Popper. *The Logic of Scientific Discovery*. Routledge, 1934/1959. [cited at p. 34]

[77] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michèle Sebag, and Olivier Temam. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pages 36–45, 2012. [cited at p. 35]

[78] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery. [cited at p. 35]

[79] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: reference workloads for modern deep learning methods. *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep 2016. [cited at p. 35, 36]

[80] Benchmarking deep learning operations on different hardware, 2021. [Online; accessed 26-December-2021]. [cited at p. 35]

[81] Wanling Gao, Fei Tang, Jianfeng Zhan, Xu Wen, Lei Wang, Zheng Cao, Chuanxin Lan, Chunjie Luo, Xiaoli Liu, and Zihan Jiang. AIBench scenario: Scenario-distilling AI benchmarking, 2021. [cited at p. 36]

[82] A benchmark framework for Tensorflow, 2021. [Online; accessed 08-January-2022]. [cited at p. 36]

[83] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of DAWNBench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, jul 2019. [cited at p. 36]

[84] Cody A. Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei A. Zaharia. DAWNBench: An end-to-end deep learning benchmark and competition. 2017. [cited at p. 36]

[85] Jack Dongarra and Piotr Luszczek. *LINPACK Benchmark*, pages 1033–1036. Springer US, Boston, MA, 2011. [cited at p. 36]

[86] MLCommons. MLCommons. https://mlcommons.org. [cited at p. 36, 37]

[87] Wanling Gao, Fei Tang, Lei Wang, Jianfeng Zhan, Chunxin Lan, Chunjie Luo, Yunyou Huang, Chen Zheng, Jiahui Dai, Zheng Cao, Daoyi Zheng, Haoning Tang, Kunlin Zhan, Biao Wang, Defei Kong, Tong Wu, Minghe Yu, Chongkang Tan, Huan Li, Xinhui Tian, Yatao Li, Junchao Shao, Zhenyu Wang, Xiaoyu Wang, and Hainan Ye. AIBench: An industry standard internet service AI benchmark suite, 2019. [cited at p. 37]

[88] Nina Ihde, Paula Marten, Ahmed Eleliemy, Gabrielle Poerwawinata, Pedro Silva, Ilin Tolovski, Florina M. Ciorba, and Tilmann Rabl. A survey of big data, high performance computing, and machine learning benchmarks. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, pages 98–118, Cham, 2022. Springer International Publishing. [cited at p. 37]

[89] MLCommons. MLPerf™ Training Launched. https://mlcommons.org/en/news/mlperf-training-launched, May 2019. [Online; accessed 17-July-2021]. [cited at p. 37]

[90] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2020. [cited at p. 37]

[91] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey C. Fox, David Kanter, Thorsten Kurth, Peter Mattson, Dawei Mu, Amit Ruhela, Kento Sato, Koichi Shirahata, Tsuguchika Tabaru, Aristeidis Tsaris, Jan Balewski, Ben Cumming, Takumi Danjo, Jens Domke, Takaaki Fukai, Naoto Fukumoto, Tatsuya Fukushi, Balazs Gerofi, Takumi Honda, Toshiyuki Imamura, Akihiko Kasagi, Kentaro Kawakami, Shuhei Kudo, Akiyoshi Kuroda, Maxime Martinasso, Satoshi Matsuoka, Henrique Mendonça, Kazuki Minami, Prabhat Ram, Takashi Sawada, Mallikarjun Shankar, Tom St. John, Akihiro Tabuchi, Venkatram Vishwanath, Mohamed Wahib, Masafumi Yamazaki, and Junqi Yin.

MLPerf HPC: A holistic benchmark suite for scientific machine learning on HPC systems. *CoRR*, abs/2110.11466, 2021. [cited at p. 37]

[92] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, Urmish Thakker, Antonio Torrini, Peter Warden, Jay Cordaro, Giuseppe Di Guglielmo, Javier Duarte, Stephen Gibellini, Videet Parekh, Honson Tran, Nhan Tran, Niu Wenxu, and Xu Xuesong. MLPerf Tiny Benchmark, 2021. [cited at p. 37]

[93] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Kenneth Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, Mostafa El-Khamy, Jungwook Hong, Michael Buch, Cindy Trinh, Thomas Atta-fosu, Fatih Cakir, Masoud Charkhabi, Xiaodong Chen, Jimmy Chiang, Dave Dexter, Woncheol Heo, Guenther Schmuelling, Maryam Shabani, and Dylan Zika. MLPerf Mobile Inference Benchmark, 2021. [cited at p. 37]

[94] MLCommons™. General policies for MLPerf™. https://github.com/mlcommons/policies. [Online; accessed 18-February-2022]. [cited at p. 39, 43]

[95] Peter Mattson. MLPerf™ Training & Inference Benchmarks. https://hc33.hotchips.org/assets/program/tutorials/HC2021.Google.PeterMattson.pdf, Aug 2021. Presented at 2021 IEEE Hot Chips 33 Symposium (HCS) - Tutorials, [Online; accessed 20-February-2022]. [cited at p. 39]

[96] Intel. Intel® Distribution of OpenVINO™ Toolkit. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html. [Online; accessed 18-July-2021]. [cited at p. 44]

[97] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570. IEEE, 2017. [cited at p. 45]

[98] Grigori Fursin. Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2197), Mar 2021. [cited at p. 46]

[99] MLCommons. MLCube. https://mlcommons.github.io/mlcube. [cited at p. 46]

[100] Gregg Barrett. Introducing MLCube. https://towardsdatascience.com/introducing-mlcube-83b94a811a69. [cited at p. 46]

[101] Antonio Maffia, Helmar Burkhart, and Gang Mu. Accelerating life science notebook applications: Architectural issues and use cases. Poster at 2018 Platform for Advanced Scientific Computing Conference (PASC18), July 2018. [cited at p. 46, 58]

[102] Virtuozzo. Open source container-based virtualization for Linux. https://openvz.org/. [Online; accessed 13-February-2022]. [cited at p. 50]

[103] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, page 241–254, USA, 2004. USENIX Association. [cited at p. 50]

[104] Canonical Ltd. What's LXC? https://linuxcontainers.org/lxc/introduction/. [Online; accessed 13-February-2022]. [cited at p. 50]

[105] Docker Inc. Docker overview. https://docs.docker.com/get-started/overview/. [Online; accessed 12-February-2022]. [cited at p. 50]

[106] IBM. Docker. https://www.ibm.com/cloud/learn/docker. [Online; accessed 12-February-2022]. [cited at p. 51]

[107] Peini Liu and Jordi Guitart. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. *The Journal of Supercomputing*, 77(6):6273–6312, Jun 2021. [cited at p. 51]

[108] Lucas Benedicic, Felipe A. Cruz, Alberto Madonna, and Kean Mariotti. Sarus: Highly scalable docker containers for HPC systems. In Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, pages 46–60, Cham, 2019. Springer International Publishing. [cited at p. 51, 53]

[109] Matt et. al Heon, Dan Walsh, Brent Baude, Urvashi Mohnani, Ashley Cui, Tom Sweeney, Giuseppe Scrivano, Chris Evich, Valentin Rothberg, Miloslav Trmač, Jhon Honce, Qi Wang, Lokesh Mandvekar, Adrian Reber, Eduardo Santiago, Sascha Grunert, Nalin Dahyabhai, Anders Bjorklund, Kunal Kushwaha, Sujil Ashwin Sha, Yiqiao Pu, Zhangguanzhang, Matej Vasek, and Podman Communit. Podman: A tool for managing oci containers and pods, 1 2018. [cited at p. 51]

[110] The Linux Foundation. Open Container Initiative. https://opencontainers.org/. [Online; accessed 13-February-2022]. [cited at p. 51]

[111] Podman. What is Podman? https://podman.io/whatis.html. [Online; accessed 13-February-2022]. [cited at p. 51]

[112] Inc. Red Hat. Chapter 15. using the container-tools API. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/building_running_and_managing_containers/assembly_using-the-container-tools-api_building-running-and-managing-containers, 2022. [Online; accessed 21-March-2022]. [cited at p. 51]

[113] Holger Gantikow, Steffen Walter, and Christoph Reich. Rootless containers with Podman for HPC. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 343–354, Cham, 2020. Springer International Publishing. [cited at p. 51]

[114] Douglas M Jacobsen and Richard Shane Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, pages 33–49, 2015. [cited at p. 52]

[115] Reid Priedhorsky and Tim Randles. Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. [cited at p. 52]

[116] Buildah. A tool that facilitates building Open Container Initiative (OCI) container images. https://buildah.io/. [Online; accessed 12-February-2022]. [cited at p. 52]

[117] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. [cited at p. 52]

[118] Chapter 5 - the essential resource management. In Thomas Sterling, Matthew Anderson, and Maciej Brodowicz, editors, *High Performance Computing*. [cited at p. 54]

[119] Naweiluo Zhou, Yiannis Georgiou, Marcin Pospieszny, Li Zhong, Huan Zhou, Christoph Niethammer, Branislav Pejak, Oskar Marko, and Dennis Hoppe. Container orchestration on HPC systems through Kubernetes. *Journal of Cloud Computing*, 10(1):16, Feb 2021. [cited at p. 54]

[120] Angel M. Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E. Grant. Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms. In *2019*

*IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20, 2019. [cited at p. 54]

[121] Sergio López-Huguet, J. Damià Segrelles, Marek Kasztelnik, Marian Bubak, and Ignacio Blanquer. Seamlessly managing HPC workloads through Kubernetes. In Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 310–320, Cham, 2020. Springer International Publishing. [cited at p. 54]

[122] Guohua Li, Joon Woo, and Sang Boem Lim. HPC cloud architecture to reduce HPC workflow complexity in containerized environments. *Applied Sciences*, 11(3), 2021. [cited at p. 54]

[123] Enis Afgan, Dannon Baker, Bérénice Batut, Marius van den Beek, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Björn A. Grüning, Aysam Guerler, Jennifer Hillman-Jackson, Saskia Hiltemann, Vahid Jalili, Helena Rasche, Nicola Soranzo, Jeremy Goecks, James Taylor, Anton Nekrutenko, and Daniel Blankenberg. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Res.*, 46(W1):W537–W544, 2018. [cited at p. 55]

[124] V. Petkov, M. Gerndt, and M. Firbach. PAThWay: Performance Analysis and Tuning Using Workflows. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 792–799, Nov 2013. [cited at p. 55]

[125] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 137–148, New York, NY, USA, 2013. ACM. [cited at p. 55]

[126] Apache Software Foundation. Apache Airflow. https://airflow.apache.org. [Online; accessed 21-March-2022]. [cited at p. 55]

[127] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018. [cited at p. 55]

[128] Danilo Guerrera. *Towards a discipline of performance engineering: lessons learned from stencil kernel benchmarks*. PhD thesis, University of Basel, 2018. [cited at p. 57, 69]

[129] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, Netherlands, 2016. IOS Press. [cited at p. 58]

[130] OpenJS Foundation. Node.js, 4 2021. version v14.16.1. [cited at p. 66]

[131] OpenJS Foundation. Express.js, 5 2019. version 4.17.1. [cited at p. 66]

[132] Socket.IO. Socket.IO, 9 2020. version 2.3.0. [cited at p. 67]

[133] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016. [cited at p. 92, 113, 120]

[134] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications, 2017. [cited at p. 92, 128]

[135] MLCommons. MLPerf v0.5 Inference Results - Intel Closed - GitHub repository: OpenVINO SSD-MobileNets-v1 Single-Stream execution scripts (Windows OS version). https://github.com/mlcommons/inference_results_v0.5/blob/master/closed/Intel/code/ssd-small/openvino-windows/scripts/int8_cpu_ssdmobilenet_single.bat, 2019. [Online; accessed 21-March-2022]. [cited at p. 92]

[136] GitHub. GitHub actions. https://docs.github.com/en/actions. [Online; accessed 13-February-2022]. [cited at p. 101]

[137] Docker Inc. Docker Hub. https://docs.docker.com/docker-hub/. [Online; accessed 13-February-2022]. [cited at p. 101]

[138] GitHub. Storing workflow data as artifacts. https://docs.github.com/en/actions/advanced-guides/storing-workflow-data-as-artifacts. [Online; accessed 13-February-2022]. [cited at p. 101]

[139] MLCommons. MLPerf v0.7 Results - Inference: Datacenter - ID: Inf-0.7-102. https://mlcommons.org/en/inference-datacenter-07/, October 2020. [Online; accessed 13-February-2022]. [cited at p. 105]

[140] MLCommons. MLPerf v0.7 Results - Inference: Datacenter - ID: Inf-0.7-101. https://mlcommons.org/en/inference-datacenter-07/, October 2020. [Online; accessed 13-February-2022]. [cited at p. 105]

[141] MLCommons. MLPerf v0.7 Results - Inference: Datacenter - ID: Inf-0.7-100. https://mlcommons.org/en/inference-datacenter-07/, October 2020. [Online; accessed 13-February-2022]. [cited at p. 106]

[142] sciCORE. About sciCORE. https://scicore.unibas.ch/about-scicore/. [Online; accessed 13-February-2022]. [cited at p. 107]

[143] HPC group. miniHPC: SMALL BUT MODERN HPC. https://hpc.dmi.unibas.ch/en/research/minihpc/. [Online; accessed 13-February-2022]. [cited at p. 109]

[144] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing. [cited at p. 113, 120]

[145] MLCommons. MLPerf v0.5 Results - Inference, GitHub issue: Intel SSD-MobileNet-v1 accuracy. https://github.com/mlcommons/inference_results_v0.5/issues/27, April 2020. [Online; accessed 16-July-2021]. [cited at p. 115]

[146] MLCommons. MLPerf v0.5 Inference Results - Intel Closed - GitHub repository: OpenVINO SSD-MobileNets-v1 / ResNet-50 v1.5 / MobileNet-v1 model calibration (Windows OS). https://github.com/mlcommons/inference_results_v0.5/tree/master/closed/Intel/calibration, 2019. [Online; accessed 16-July-2021]. [cited at p. 115, 123]

[147] Intel. Intel® Distribution of OpenVINO™ Toolkit 2019_R3.1: Optimization guide. https://docs.openvinotoolkit.org/2019_R3.1/_docs_optimization_guide_dldt_optimization_guide.html. [Online; accessed 18-July-2021]. [cited at p. 116]

[148] MLCommons. MLPerf v0.5 Inference Results - Intel Closed - GitHub repository: OpenVINO SSD-MobileNets-v1 / ResNet-50 v1.5 / MobileNet-v1 (Windows OS version). https://github.com/mlcommons/inference_results_v0.5/tree/4191ca07994d4e3f48ccf567c2e27d56914b2b88/closed/Intel/code/ssd-small/openvino-windows, 2019. [Online; accessed 16-July-2021]. [cited at p. 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 128, 131]

[149] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. [cited at p. 124, 128, 134, 137, 141]

[150] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. [cited at p. 124, 134, 137, 141]

[151] Alfred Torrez, Timothy Randles, and Reid Priedhorsky. HPC container runtimes have minimal or no performance impact. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 37–42, 2019. [cited at p. 132]

[152] MLCommons. MLPerf v0.7 Inference Results - Intel Closed - GitHub repository: ResNet-50 v1.5. https://github.com/mlcommons/inference_results_v0.7/tree/master/closed/Intel/code/resnet, 2020. [Online; accessed 20-March-2022]. [cited at p. 133, 139, 142]

[153] MLCommons. MLPerf v0.7 Inference Results - Intel Closed - GitHub repository: ResNet-50 v1.5 model calibration. https://github.com/mlcommons/inference_results_v0.7/tree/master/closed/Intel/calibration, 2020. [Online; accessed 20-March-2022]. [cited at p. 133]

[154] MLCommons. MLPerf v1.0 Results - Inference, Intel Closed - GitHub issue: TensorFlow ResNet-50 v1.5 compilation. https://github.com/mlcommons/inference_results_v1.0/issues/10, September 2021. [Online; accessed 20-March-2022]. [cited at p. 139]

[155] MLCommons. MLPerf v1.0 Results - Inference, Intel Closed - GitHub issue: TensorFlow ResNet-50 v1.5 execution. https://github.com/mlcommons/inference_results_v1.0/issues/14, February 2022. [Online; accessed 20-March-2022]. [cited at p. 139]

[156] The Turing Way Community. The Turing Way: A handbook for reproducible, ethical and collaborative research, July 2022. [cited at p. 146]

[157] Swissuniversities. National strategy and action plan. [Online; accessed 25-June-2022]. [cited at p. 148]

[158] Swissuniversities. Swiss Open Research Data Grants. [Online; accessed 25-June-2022]. [cited at p. 148]

[159] NVIDIA. The Triton Inference Server provides an optimized cloud and edge inferencing solution, 2022. [Online; accessed 28-March-2022]. [cited at p. 148]

[160] Antonio Maffia. Accelerators in a hybrid HPC world: How can applications benefit? Poster at PhD Forum 2018 ISC High Performance (ISC18), June 2018. [cited at p. 149]