Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Bachelor Degree in Informatics Engineering

Computer Engineering Specialization

Degree Final Project

Computational Performance Analysis of Deep Learning using High Resolution Images with Variable Shapes

Author
Pedro Megias Montesinos
Director
Marta Garcia Gasulla
Co-director
Ferran Parés Pont
Tutor
Julián David Morillo Pozo
GEP Tutor
Paola Lorenza Pinto

Contents

List of Figures

List of Tables

1 Project Management

1.1 Context and scope of the project

1.1.1 Context

Introduction

Nowadays, Artificial Intelligence (AI), is becoming a huge study field in computer science, particularly through Deep Neural Networks (DNN), and it is being applied in different daily life spheres, such as medical diagnosis [4, 5], autonomous driving [6, 7] or satellite data analysis [8].

Some of these fields face similar problems, like object recognition and classification, which could be considered a critical task when looking into, for example, autonomous driving.

In fact, image classification is a task commonly performed by Convolutional Neural Networks (CNN), by training a model to understand what we are looking for. Usually, training is done with squared and low resolution images to speed up the process, but this practice entails deformation of visual patterns and generalizes information loss.

In this project we want to study a specific CNN called ResNet [2], executing in a High Performance Computing (HPC) environment, using the MAMe [9, 10] dataset, which contains images of high resolution and variable shape, to analyse the needs and outcomes of using this kind of data.

This project is under an Educational Cooperation Agreement between The Facultat d'Informàtica de Barcelona and the Barcelona Supercomputing Center (BSC) and has been developed in collaboration with the Best Practices for Performance and Programmability (BePPP) group.

Terms and concepts

High Performance Computing

High Performance Computing (HPC) refers to the aggregation of compute power to perform massive and complex computations. This term is commonly associated with supercomputers.

Supercomputer

A supercomputer is a machine assembled with many computers interconnected to allow performing operations that, otherwise, could not be feasible or may take too much time.

MareNostrum4

MareNostrum 4 [11], or MN4, is a supercomputer managed by the Barcelona Supercomputing Center and it is divided in two blocks: general purpose block, which represents the bigger portion of computing power; and the emergent technologies block, which aims to test new technologies. The general purpose block contains 3456 nodes, which contain 2 Intel Xeon 8160 CPUs per node. These CPUs are equipped with 24 cores that run at 2.1 GHz. Also, nodes are interconnected with an Omni-Path full-tree at 100 Gbps.

Machine Learning

Machine learning represents a branch of the artificial intelligence focused in the usage of data and algorithms to emulate the way we learn as humans. This concept is a subset of all the AI applications and gives birth to the concept of Neural Networks.

Neural Network

Neural Networks are inspired in human brain cells by trying to simulate their behaviour with mathematical approaches. Basically, a neural network is comprised of four main components: inputs, weights, bias and output. These programs are structured with interconnected layers that can adapt themselves based on the information that flows through the network. Large networks with more layers are also called Deep Neural Networks (DNN) due to their complexity. In this project we are using a kind of DNN which relies on using the convolution operation, Convolutional Neural Networks (CNN), which excel at object recognition in image data.

MAMe dataset

MAMe [9, 10] is an image classification dataset that contains thousands of artworks from three different museums, becoming a huge data pool with high resolution images. This dataset has been created to provide a tool for studying the impact of high resolution and variable shape properties in the image classification field, and also to train CNN models in material recognition.

PyTorch

PyTorch is an open source machine learning framework specialized in tensor modeling networks, created by Facebook's AI research lab. It is based on the Torch library and many Deep Learning softwares are built on top of it, like Tesla Autopilot or Uber's Pyro [12].

Hyperparameter

This term is used by AI developers just to highlight that a particular parameter of the execution is used to control the learning process. With this naming scheme, we can differentiate between parameters that have a direct impact on the model's behaviour from an AI perspective.

OpenMP

Open Multi-Processing [13], or OpenMP, is an application programming interface that allows the user to create shared-memory multiprocessing software. Consists on a set of compiler directives, library routines and environment variables that affect run-time behaviour.

Performance Analysis

Performance Analysis refers to the process of gathering data from the application executions, study this data and identify possible bottlenecks that may impact its performance. As this project is a collaboration with the Best Practices for Performance and Programmability group (BePPP) and the Performance Optimization and Productivity (POP) center of excellence, we will follow their methodology.

BSC tools

In order to do a performance analysis of an execution, we need some data extraction, processing and visualization tools [14]. The main BSC tools we are going to use are:

- **Extrae:** This tool is responsible of gathering all the needed data from the execution. Extrae allows us to trace events as OpenMP calls, gather hardware counter values and create custom events at code level.
- **Basic Analysis:** Is an aggregation of scripts which use existing tools to automatically compute the POP methodology metrics.
- **Paraver:** This tool is a desktop application that allows us to load traces generated by Extrae and to visually study them. The most common views of the traces are histograms and timelines.

Figure 1 shows an example timeline representing the Instructions Per Cycle (IPC) of threads during the execution, by representing time in the X axis and assigning a row to each thread. With this view we can spot where the IPC of any thread is high or low by its code color and allows us to analyse the behaviour of the code in certain regions. Also, when no color is displayed, means that this thread was not created yet or it does not have any computation to do in the application scope.

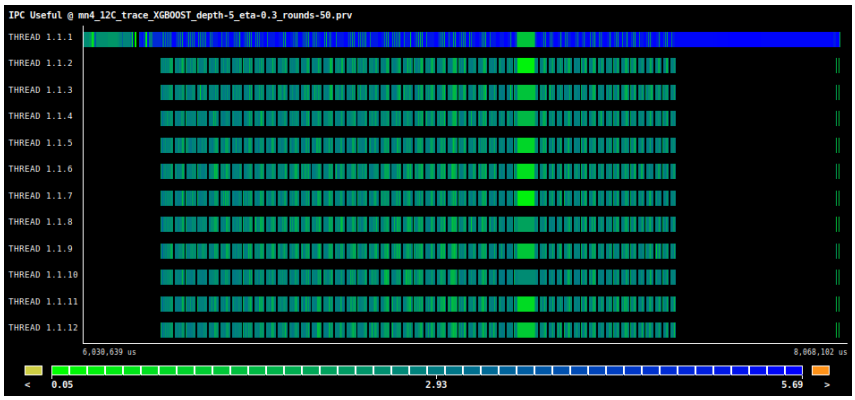

Figure 1: Sample Paraver IPC Timeline. Own creation.

When looking at Figure 2 we can see a histogram representing the same metric, IPC. The first row shows a gradient color that represents the IPC value from 0 to the maximum value found. Each other row represents a thread and colors in these rows represent the amount of time this thread has been running with this specific IPC value.

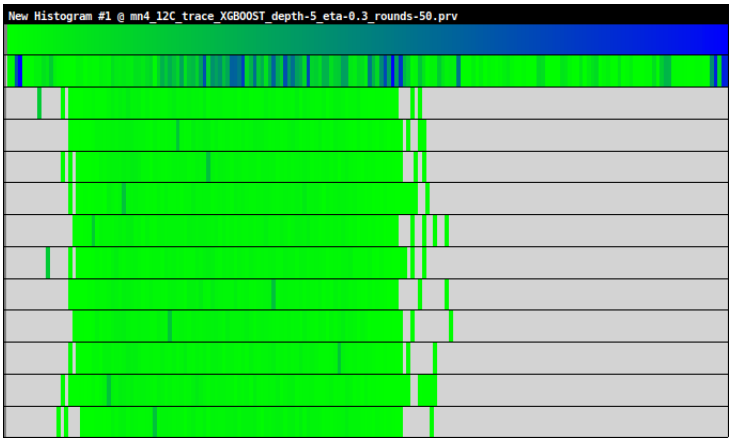

Figure 2: Sample Paraver IPC Histogram. Own creation.

1.1.2 Motivation

A common practice when training DNN models for image-related tasks is to down-sample images before computations, which is the process of reducing their size to a set amount of pixels and changing their aspect ratio to a squared one. This is typically done to reduce the memory requirements and training time of the model, but comes with a huge drawback, deformation of visual patterns. Many applications are not particularly affected by these deformations because they do not rely on details of the input, as for example telling cats from dogs, but in other domains images are naturally of high-resolution and variable shape and down-sampling could entail significant prediction performance reductions.

High-resolution data entails large memory requirements, which limits the amount of images that can be processed together, or in the same batch, and batch size limitations have a significant effect on the efficiency of computation. Currently used accelerators, as Graphics Processing Units (GPU), are rather limited in terms of memory capacity, although workarounds to load larger memories than the one offered by the device have already been proposed, such as model parallelism [15, 16], activations re-computation [17] and offloading [18], enabling greater memory loads at the cost of computation efficiency.

In this high memory load context we want to avoid accelerators and rather use CPU devices as an alternative. The two main reasons in favour of CPUs are the lack of added components and functionalities for their execution and the access to large memory devices, enabling larger, and therefore more efficient, batch sizes.

Variable shape is challenging to deal with in the context of batch training. All images in a batch need to have the exact same shape to be computed, and when our data is of variable shape the easiest way to achieve a uniform one without deformation or loss of information is through padding. Padding is a technique used to extend the size of an image by adding fixed-valued pixels, in our case zeros, to fill the gaps between images of different shape found in the same batch. However, padding generates three main problems [19]. First, it introduces noise in form of non-informative pixels, which can affect the learning process. Second, it increases the computational cost of the task, as padding pixels are also computed by the CNN. And third, padding increases the memory requirements of the task, as these values still need to be kept in memory. Additionally, applying random batching in a variable shape context may result in landscape and portrait images being batched together, entailing huge amounts of padding, to the point where more padding than informative pixels may be present in the batch.

Although AI practitioners tend to measure model performance taking into account the elapsed execution time and the model prediction accuracy, we want to study how these different inputs affect computational performance in terms of instruction count, frequency and parallel efficiency among others. With this, we may be able to find run-time configurations that achieve a good AI perspective performance while improving program efficiency in the hardware level.

1.1.3 Scope

In this section you will find a description of the objectives of the project and the identification of the possible risks.

Objectives

This project has three main objectives:

- Highlight differences between low resolution and high resolution datasets in terms of computational performance and behaviour.
- Identify computational challenges and bottlenecks derived from high resolution data processing.

Other sub-objectives for this project:

- Gaining experience in the performance analysis, in particular POP methodology and optimization.
- Learning about Neural Networks constraints, needs and applications.
- Apply performance analysis methodologies to Deep Learning field.

Obstacles and risks

The following risks have a non zero probability of causing some trouble during the elaboration of this project:

- **MareNostrum 4 unavailable:** As MN4 is a shared resource with lots of other users and services, this machine could be stopped by maintenance, overheating or other errors. If this happens, moving the study to another similar machine is possible.
- **System noise:** Supercomputers operate over complex environments that could lead to system noise [20], which refers to abnormal behaviour that can disrupt the application execution. With a correct statistical analysis, this risk can be avoided.
- **Misinterpretation of metrics:** This is always possible, leading to incorrect conclusions. By keeping synchronized with the director we can minimize this risk.
- **COVID-19:** As we are currently living in a pandemic scene, we must ensure our security by following all the official guidelines regarding this matter. Keeping our meetings in online format will further reduce the risk of getting sick.
- **Incorrect executions:** It is a common flaw when executing different tests to forget to change a parameter, leading to an incorrect execution. This risk can be avoided by naming the results after the parameters that produced them or during the feedback received while development is in course.
- **Russian invasion:** During the last few weeks this conflict started in north-east Europe. Although probabilities of Spain being involved in an armed conflict are few, we cannot oversee the economic side effects, as the electricity prices increase. This risk will be considered as of high impact for our project.

1.1.4 Methodology

As this project merges AI concepts with computational performance analysis, the working methodology will relay in constant communication between Marta Garcia (the project director and performance analysis specialist), Ferran Parés (the project co-director and AI specialist), Julián Morillo (the project tutor) and me. All communications are done via email and group meetings, preferably online. In fact, this project follows the agile [21] methodology for dynamic work flow and communication.

Regarding code management, a git repository is being used to keep track of all changes done. An issue page in the internal GitLab has also been created to share my updates to keep BePPP team members informed. Run logs and generated data are stored in the MN4 file system with a local backup in my laptop and another synchronized copy in an external disk.

All project documentation is written in LaTeX using the OverLeaf [22] online tool and all graphics and tables are built with Google spreadsheet for ease of use.

Performance analysis methodology is defined by the POP project and will be explained in detail in following chapters.

1.2 Time planning

In order to maintain a constant workflow during this project development, the time planning presented in this section is designed using 8 hours a day except for the weekends, summing 40 hours a week.

1.2.1 Task description

To ensure this project follows his tracks we must define all tasks to be performed in order to achieve our objectives.

1. **Project Management**
   - 1.1. **Context and Scope:** definition of context and scope, includes writing all sections required for the first GEP deliverable.
   - 1.2. **Meetings:** regular meetings will occur each week to keep track of the progress and make decisions if needed.
   - 1.3. **Time Planning:** definition of tasks and detail of dependencies between them, including requirements and resources. Gantt diagram building and risk management with their corresponding effects over the tasks.
   - 1.4. **Budget and Sustainability:** report about monetary costs of the project and sustainability.
2. **Environment Setup**
   - 2.1. **Sources compilation:** even when the application can run with a precompiled version, we must ensure all our HPC capabilities are being profited, so compiling the framework inside our machine is not avoidable.
   - 2.2. **Application test:** to ensure all modules are working, some test executions have to be made using portions of the dataset. Required to test tools integration.
   - 2.3. **BSC tools test:** when the application is working correctly, then we can integrate the BSC tools to test if we can obtain our desired measures and check tools compatibility with the application. This task includes a trace generation and applying a modelfactors analysis to ensure all events are correctly gathered.
3. **Performance Analysis**
   - 3.1. **Trace generation:** data extraction can produce a bit of overhead during the execution extending the elapsed times. Also, generating the trace from this data needs a lot of time. If Extrae generates the trace correctly and can be visualized with Paraver, this task is completed.
   - 3.2. **Trace analysis:** analysis can be done when any trace is generated, we do not need all traces to start analysing. During this task Paraver will be used to inspect the whole trace in order to find and differentiate the regions or steps of the execution.

3.3. **Structure Analysis:** identifying the program structure is key to understand its behaviour. It is also needed to select a focus of our analysis and to better understand the metrics. The main goal is to find an iterative structure that describes the execution behaviour.

3.4. **ModelFactors:** executing these scripts we will extract all relevant metrics referring each execution. This task is mandatory to understand the following steps and may take some time to perform, depending on the traces size.

3.5. **Flaw detection:** identify which regions of the execution are not taking profit of the machine resources. In this task we want to state which reasons may produce this flaws and try to reason about a possible solution.

3.6. **Parameter tuning:** tweak different execution parameters to improve execution efficiency, as could be execution cores or work distribution grain size. During this task we will perform scalability tests to find a sweet spot for our specific executions and datasets.

3.7. **Hyperparameter tuning:** tweak different AI execution parameters to improve prediction accuracy and/or performance, as learning rate or batch size among others. The goal here is to find a combination of hyperparameters that produces good accuracy predictions and check its effects on hardware performance.

3.8. **Evaluation:** compare new behaviour with initial executions and analyse impact of tuning decisions.

4. **Final Milestone**

4.1. **Memory redaction:** this task is being developed during all the project, but has the most effort at the end, when all the analysis has been completed.

4.2. **Presentation warm-up:** preparation of the presentation slides and speech.

1.2.2 Resources and requirements

We can distinguish different kinds of resources, human and material ones, but not all of them are needed at each step of the project. Here you can find a description of all possible needed resources and, in the following section, those resources will be linked with specific tasks.

Human resources

- **Team:** project director, co-director, tutor and me, the author. The whole team involved in this project belongs to its human resources.

Material resources

- **MN4:** MareNostrum 4 supercomputer at BSC facilities where all executions will run.
- **Laptop:** my main computer, used for remote working at BSC and connecting to MareNostrum4.
- **Comms:** the mailing system used to keep contact with all team members as well as any other service used for online meetings.
- **OL:** Overleaf is the text editor used for writing all documentation.
- **GP:** "Ganttproject" software to produce Gantt diagrams.
- **GS:** Google Spreadsheet used to handle, organize and create graphics using the collected data.

Tasks summary and dependencies

Table 1 summarises all tasks dependencies and their estimated required time.

Task	ID	Time (h)	Dependencies	Resources
Project Management	1	120	-	-
Context and Scope	1.2	40	-	Comms, Laptop, OL
Meetings	1.3	50	-	Comms, Laptop, OL
Time planning	1.4	15	-	Comms, Laptop, OL, GP
Budget and Sustainability	1.5	15	-	Comms, Laptop, OL
Environment Setup	2	65	1	-
Sources Compilation	2.1	20	-	Comms, Laptop, MN4
Application test	2.2	25	2.1	Comms, Laptop, MN4
BSC tools test	2.3	20	2.2	Comms, Laptop, MN4
Performance Analysis	3	265	2	-
Trace generation	3.1	35	-	Comms, Laptop, MN4
Trace analysis	3.2	20	-	Comms, Laptop, MN4
Structure analysis	3.3	25	3.2	Comms, Laptop, MN4
ModelFactors	3.4	25	3.3	Comms, Laptop, MN4, GS
Flaw detection	3.5	30	3.4	Comms, Laptop, MN4, GS
Parameter tuning	3.6	50	3.5	Comms, Laptop, MN4, GS
Hyperparameter tuning	3.7	50	3.5	Comms, Laptop, MN4, GS
Evaluation	3.8	30	3.6 + 3.7	Comms, Laptop, MN4, GS
Final Milestone	4	100	-	-
Memory redaction	4.1	60	3.8	Comms, Laptop, OL
Presentation warm-up	4.2	40	4.1	Comms, Laptop, OL

Table 1: Task summary and dependencies. Own creation.

1.2.3 Risk management

In this section, all above mentioned risks will be evaluated in order to approximate their individual impact on the project planning:

- **MareNostrum 4 unavailable:** If this machine is not available, all executions will be delayed as we are evaluating performance on this specific architecture. When MN4 becomes unavailable, the issue is usually resolved in less than 12 hours. In case a critical failure occurs and MN4 could not be reestablished, it is contemplated to move the study to a similar machine that BSC disposes of, implying a repetition of all executed tests, which could add up from 48 to 72 extra hours. Tasks affected by this failure belong to groups 2 and 3.
- **System noise:** Solution for this kind of problems is contemplated in the methodology of this project. As we are executing a good number of samples, this risk is being minimized from the starting point. This risk mainly affects all group 3 tasks.
- **Misinterpretation of metrics:** This error can lead to repetition of executions, which can add from 4 to 12 hours depending on the dataset used. Probability of this happening is low, as every analysis will be supervised by the directors. This risk affects 3.5, 3.6 and 3.7 tasks.
- **COVID-19:** In case any of the team members gets sick, the situation may not affect at all the project development, as all meetings are conducted mainly online. Only in case the author gets sick the time planing could be affected by adding up to 24 hours to any task that is being conducted, so this risk could affect all tasks.

- **Incorrect executions:** As with the misinterpretation risk, if any execution must be repeated this could add up to 12 hours depending of the dataset. As explained before, results are named by the parameters that produced them, so this failure can be detected fast. This risk affects every task in group 3.
- **Russian Invasion:** Nothing we can plan here could avoid war. If something terrible happens in Spain due to this matter, this project would be immediately stopped, but probabilities and hope tell us this will not occur. We cannot avoid electricity pricing, but if it increases hugely, some efforts will be destined into finishing development with minimum executions in order to reduce our economic footprint.

1.2.4 Gantt diagram

In Figure 3 you can find the Gantt diagram representing the time distribution of all the above described tasks, starting by the 21/02/2022, which is the start date of the GEP course. Notice that tasks have spare time at the end to cover possible deviations. If this occurs, the diagram will be updated.

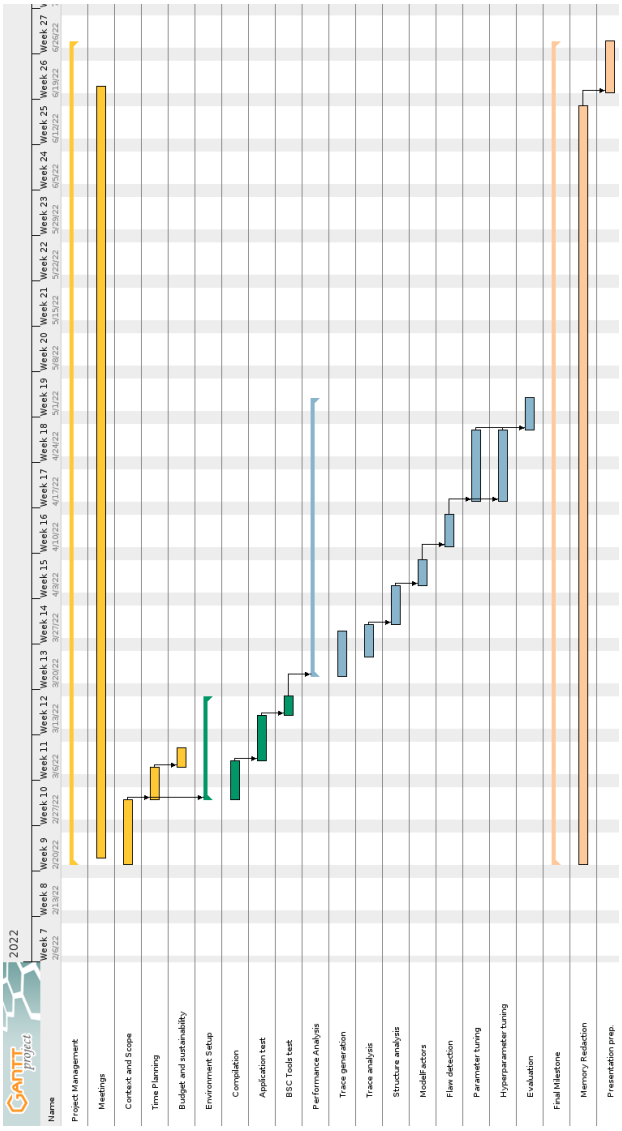

Figure 3: Project Gantt diagram. Own creation.

1.3 Budget and sustainability

1.3.1 Budget

When planning our budget we must take into account material and human costs. For the material costs we count on all the needed devices and tools for each task realization. As all software used during this project is free, we only account for hardware prices and usage costs.

Regarding the usage of Mare Nostrum 4 machine, we can only use one full node during our executions and one login node to interact with the machine. As the price of this machine must remain confidential, it has been estimated by simulating the creation of a node with similar components. Using Thinkmate [23] configuration website we have built Table 2 indicating prices for each component. These prices can vary at any time due to economic restrictions or component shortages. With this approximation we can estimate the costs of our cluster time usage.

MareNostrum 4 similar node			
Piece	**Price per unit**	**Units**	**Price**
Intel Xeon Platinum 8160	4.700,00 €	2	9.400,00 €
DDR4 RAM 32GB	165,00 €	12	1.980,00 €
Mellanox 100Gb/s HDR100 InfiniBand	979,00 €	1	979,00 €
Intel 10-Gigabit Ethernet	384,00 €	1	384,00 €
240GB Micron 5300 MAX series SATA	195,00 €	1	195,00 €
Total			**12.938,00 €**

Table 2: Node components list. Own creation.

Material costs are grouped in Table 3. This table includes all needed material for the main developer to properly work on the project, even if working from home or at the office. Costs derived from office services as internet or electricity are not included because half of the working time is used in remote working and those costs are included in the main developer salary in Table 4. Accessories refer to any electric device needed to work, as keyboard and mouse, battery charger and a mug heater. MN4 cluster time usage has been estimated from Environment Setup and Performance Analysis tasks, which in the initial estimation sum up to 330 hours.

Material costs					
Component	**Price**	**Lifespan (years)**	**Amortization (€/hour)**	**Total hours**	**Total price**
Laptop	1.500€	4	0,128€	550	70,634€
Monitor	150€	4	0,013€	550	7,063€
Accessories	80€	4	0,007€	550	3,767€
MN4 cluster	12.938,00 €	4	1,108€	330	365,543€
Total	-	-	-	-	**447,007€**

Table 3: Material costs summary. Own creation.

We have grouped the salaries of people related with this project in Table 4, as the director or the tutor, and their contribution has been estimated around 80 hours in total for the whole project. Their salary has been estimated on the basis of the main developer, but taking into account that their job position is higher.

A contingency budget will also be added to the expenses bag to try to cover any possible deviations that could come from any risks that may affect our initial budget. The amount to add represents a 20% of the total costs of the project to cover not only the extra working hours from human costs but also the extra usage time of the cluster. The total budget can be found in Table 5.

Human costs			
Type	**Price per hour**	**Total hours**	**Total price**
Main developer	9€	550	4.950€
Supervisor	33€	80	2.640€
Total	-	-	**7.590€**

Table 4: Human costs summary. Own creation.

Project budget	
Source	**Value**
Human costs	7.590,00€
Material costs	447,01€
Contingencies	1.607,40€
Total	9.644,41€

Table 5: Project budget summary. Own creation.

1.3.2 Sustainability report

Environmental dimension

- **Have you estimated the environmental impact of the project development?**
  We have not quantified the environmental impact of this project but it could be related to the amount of electricity consumed by the project development. About this matter, measuring some executions we have built Table 6 which considers a 5 hour execution and the power consumed during the computation approximately. An estimation of its cost has been included, but electricity price can vary at light speed levels these days.

Electric consumption in executions	
Power (W)	350
Execution time (h)	5
Consumtion (kwh)	**1,75**
Price (€/kwh)	0,6
Execution cost (€)	**1,05**

Table 6: Electric consumtion in executions. Own creation.

- **If you did this project again, could you do it with less resources?**
  I could probaly avoid some mistakes made when launching executions, which led to repeating some of them. Apart from that, all used resources were needed in order to produce enough data and avoid execution noise.
- **Which resources do you think will be used during this project's lifespan?**
  It depends on how many AI developers use our guidelines. If they use this project as a starting point, their resource usage could be lower during their own development.
- **Does this project allow to reduce other resources usage? Will your project improve or worsen the global environmental footprint?**

As said, if an AI developer uses this project as a guideline, their resource usage can be lowered, so their footprint will be smaller also.

- **Does exist any scenario where the environmental footprint of this project gets bigger?**
  This project's environmental footprint will only increase if more research is performed. As mentioned in different sections, main memory constraints have lowered the amount of tests we are able to perform. If this constraint is removed, more analysis can be performed and, therefore, increase our footprint.

Economic dimension

- **Have you estimated the project cost? Which decision have you made to reduce this cost?**
  This has been addressed in Budget 1.3.1 section above and the electricity costs can be found in Table 6. To further reduce costs, we have used free software during the whole development.

- **Is the final cost adjusted to the original budget?**
  The initial budget was calculated taking into account possible execution flaws or errors and no more than this amount has been spent.

- **Does exist any scenario where the project viability is affected?**
  Given the nature of CNNs and Deep Learning, our project's analysis only matter when using this specific ResNet model. Any scenario that uses a different model (even if based in our ResNet version) could produce different behaviour outcomes.

Social dimension

- **Has this project development implied meaningful thoughts at a personal, professional or ethical level?**
  The whole AI field is drowned in ethical discussions nowadays. Deep learning models are giving us the opportunity to create programs with basic but powerful features, like seeing and distinguishing different materials (as our ResNet does). This has brought some funny discussions at a personal level, as a lot of people that doesn't belong to the computer sciences field isn't aware of the things that neural networks are capable of doing, sometimes even thinking these are all lies.

- **Who can benefit of this project? Is there any group that can be harmed by your project? How much?**
  Lots of AI practitioneers could have some benefit by taking a look into this project, some hardware awareness can't be bad. Also, I don't really think that any harm can be done to any group with this study.

2 Background

2.1 Deep Learning and Convolutional Neural Networks

Deep Learning (DL) is a subgroup of Machine Learning, which is indeed a subgroup in AI field. DL takes a step further creating complex hierarchical models to simulate the way humans learn with new information. In this specific group, algorithms are inspired in our brain structure and are known as Neural Networks.

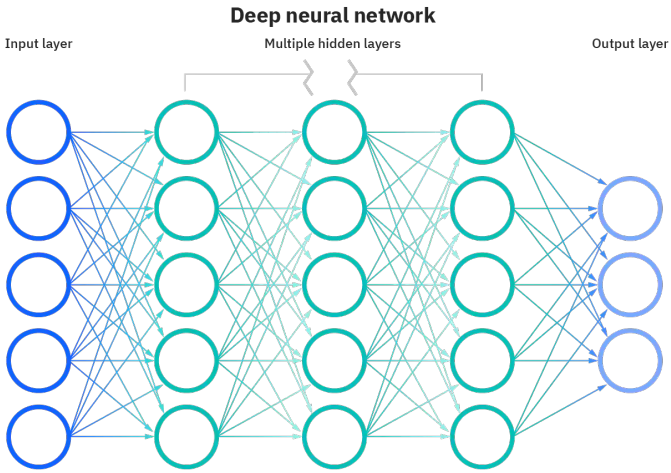

Figure 4: Deep Neural Network scheme from IBM [1].

Training and using these networks is time consuming and needs huge amounts of data to be able to test and polish the models. Although neural networks exist since the 50s, only a few years ago we have acquired the computational power and storage capacity needed to use this kind of software and design useful applications.

Convolutional Neural Networks, or CNNs, are a sub-type of neural networks based on deep learning modeling which excel in image processing tasks such as object recognition. The process of training a CNN begins with the input preparation. To feed the network we have to create a squared structure called batch, which will be filled with images from the dataset during the first phase of the execution. Also, the amount of images that compose a batch can be adjusted at the beggining of the process, modifying the batch size hyperparameter.

After creating a batch, forward phase starts. During forward, the batch is processed in different ways while traversing all the layers in the network. While passing through convolutional layers, batch dimensions are altered as a result of being processed by convolution operations. An example of this dimension change is shown in Figure 5. At the end of this phase, the model will output a set of predictions extracted from the images in the batch, which can be used to know the overall precision of the network, or accuracy.

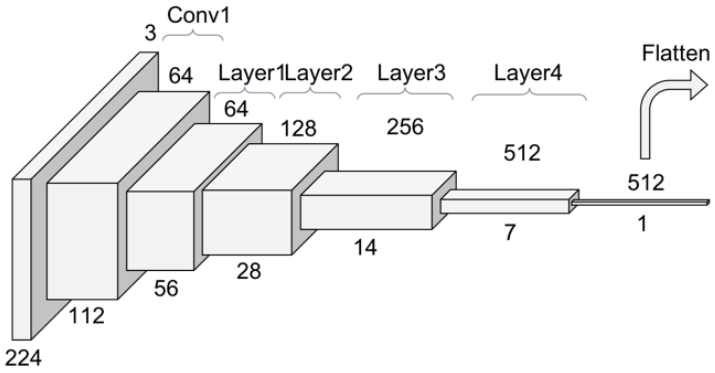

Figure 5: Batch dimension changes through forward phase. From [2].

When forward phase finishes, in order to improve the results of the predictions, backward phase begins. This phase will traverse again all the layers, but in reverse order, to propagate the learnt properties through the network and improve precision during following training steps. These properties are condensed in a variable called gradient, and this phase will register the according gradient value to each of the layers.

Once all gradients are calculated and registered in the different layers, the optimization phase begins, which consists on applying the gradients to the layers weight. When applied, this will modify the probabilities of being activated during the next forward phase, to effectively apply knowledge to the network and improve predictions.

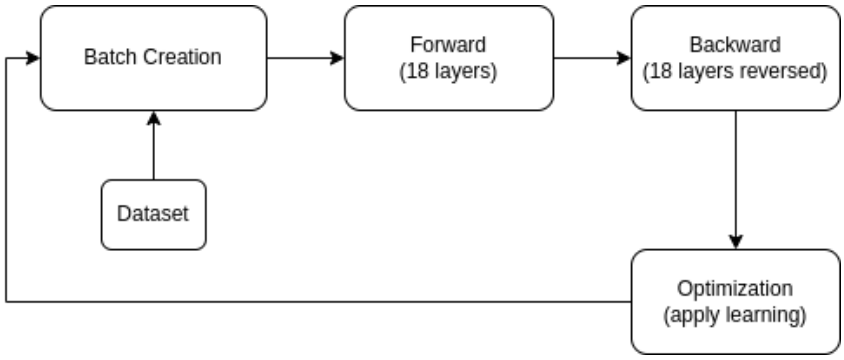

Figure 6: Training step box diagram. Own creation.

Completing the computation of the whole dataset is called an epoch, and could require more than one batch due to the amount of images in the dataset. Therefore this training must be split into several training steps, each one of them processing one batch at a time, until finishing all images in the dataset. A box diagram of the training process cicle can be seen in Figure 6. Also, results may not be accurate enough after a full training epoch, requiring to compute more epochs in sequence to allow the model to further improve its precision.

CNN models are based on the mentioned convolution operations, or convolutional layers, that transform two functions into a third that, in some sense, represents the magnitude by which the first and a translated and inverted version of the second overlap.

In essence, a convolution consists in using a matrix called filter or kernel. A representation can be seen in Figure 7. For each pixel in the image, we have to center the kernel on it and calculate its value using all surrounding pixels. This process is repeated for each pixel in the image and the kernel will move from left to right and top to bottom.

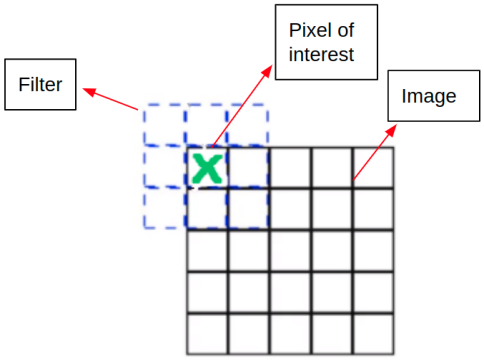

Figure 7: Convolution filter over image pixels. Own creation.

2.2 ResNet model

ResNet stands for Residual Neural Network [2]. It is a specific kind of neural network developed for image recognition. To solve complex problems, AI developers stack layers to improve accuracy and performance. The intuition behind adding more layers is that these layers can learn progressively more complex features. As an example, in case of image recognition, the first layer could learn to detect edges, the second layer may learn to recognize textures and so on.

In this project we are going to use a specific version of this neural network called ResNet-18, built with 18 of the mentioned layers:

- **Conv1:** this layers applies a first convolution operation over the input. The filter matrix applied during this layer is bigger than those used in following convolutional layers.
- **BN:** performs a batch normalization operation over each element of the input, therefore this layer doesn't change the output's size.
- **ReLu:** stands for the rectified linear unit function, which is an element-wise operation. This layer has gained popularity in the deep learning domain because it allows to decide efficiently when a neuron will be activated. This function is as simple as only applying a value if it is positive.
- **Maxpool:** the max pool operation is similar to the convolution, but it uses a submatrix to select the highest values instead of composing them.
- **Convolution layer:** although this layers also perform convolution operations, these differ from Conv1 ones. As kernel matrices are smaller, the output volume is not as reduced as with the first layer.
- **Averagepool:** as mentioned for the max pool operation, average pool uses a submatrix to traverse the input, but this layer obtains the average value for each submatrix processed.
- **Flatten:** this layer transforms multidimensional input data into one-dimensional and is a required step to feed the fully connected layer.
- **Fully connected:** this is also a single threaded layer that brings all inputs from one layer to every activation unit of the next layer. This is used to compile the data extracted by previous layers to form final output.

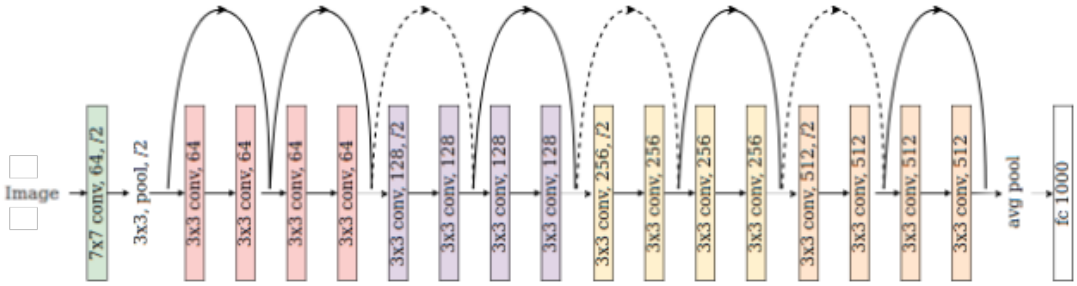

Figure 8: Resnet-18 layer connexion diagram. From [2]

In order to better understand how these layers are connected, you can have a look at Figure 8. This diagram shows how each layer serves its output as the input for the following one. There's also a chance that some layers are skipped during the execution, also shown in the diagram, but this is related to the "residual" part of the network's concept, which is not meaningful in this project's scope. Each one of the above described layers are used in the ResNet-18 model during the Forward and Backward propagation phases.

2.3 PyTorch Framework

PyTorch is an open source deep learning framework built to be flexible and modular for research, with the stability and support needed for production deployment. This framework provides a Python package for high-level features like tensor computation, with strong GPU acceleration modules. Prior to PyTorch, deep learning packages such as Caffe and Torch tended to be the most popular. Nowadays this position is being disputed between PyTorch and TensorFlow frameworks.

As deep learning started to revolutionize nearly all areas of computer science, developers and researchers wanted an efficient, easy to use library to construct, train and evaluate neural networks in the Python programming language. Python, along with R, are the two most popular programming languages for data scientist and machine learning, so it is natural that researchers wanted deep learning algorithms inside their Python ecosystems.

Although Python may not be the preferred programming language for application performance, it makes easier to test models even when not totally implemented. PyTorch also offers a C++ interface to be used, with C++ implemented operations, allowing to deploy with better performance.

2.4 From Low resolution to High resolution

AI developers tend to train their models using GPUs and, as these devices tend to be built with scarce memory, they have to face some questions: Should we train with small batch sizes to use the original images? Or should we reduce image resolutions to increase batch sizes?

In order to throw some light into these questions we will use the MAMe [9] dataset as a source to create 3 different datasets:

- **LRFS:** Low Resolution and Fixed Shape variant. All images have a squared shape with 256^2 pixels area. Each original image is down-sampled to fit the requirements.
- **MRVS:** Medium Resolution and Variable Shape variant. Images maintain their original aspect ratio or shape, but resolution is reduced. To fill a batch with these images, padding

pixels are added.

- **HRVS:** High Resolution and Variable Shape variant. In fact, this is not a variant but the original dataset, composed by extremely high resolution images with non squared aspect ratios. This input also requires the usage of padding pixels to fit in batches.

In Figure 9 you can see the differences between the batches. Creating a batch with just 1 image using LRFS images is straight forward, no gaps will generate, but looking at MRVS or HRVS you can see the added padding pixels needed to create a squared batch with non squared images.

Figure 9: Datasets image comparison. Own creation.

As mentioned in [19], this technique also increases batch computation time, as all pixels must be processed. We must take into account that random batching may lead to strange alignments between landscape and portrait images, generating a huge padding section.

2.5 POP methodology and Efficiency metrics

For the development of this project, POP [3] methodology will be followed. This is a performance analysis methodology that does not depend on the tools being used.

POP methodology can be summarised with Figure 10. The program structure must be studied in order to accomplish some objectives and to achieve a better understanding of how our program behaves, including:

- Understand general structure.
- Identify initialization/finalization phases.
- Detect iterative pattern.
- Work distribution granularity.

Selecting the Focus of Analysis is not always an easy task, because it depends on the context of the analysis. For the same trace we may select two different FoA to perform two different studies with two different objectives. When the region to be studied is clear, we can proceed to extract the performance metrics, you can find a summary of the metrics and their relation in Figure 11.

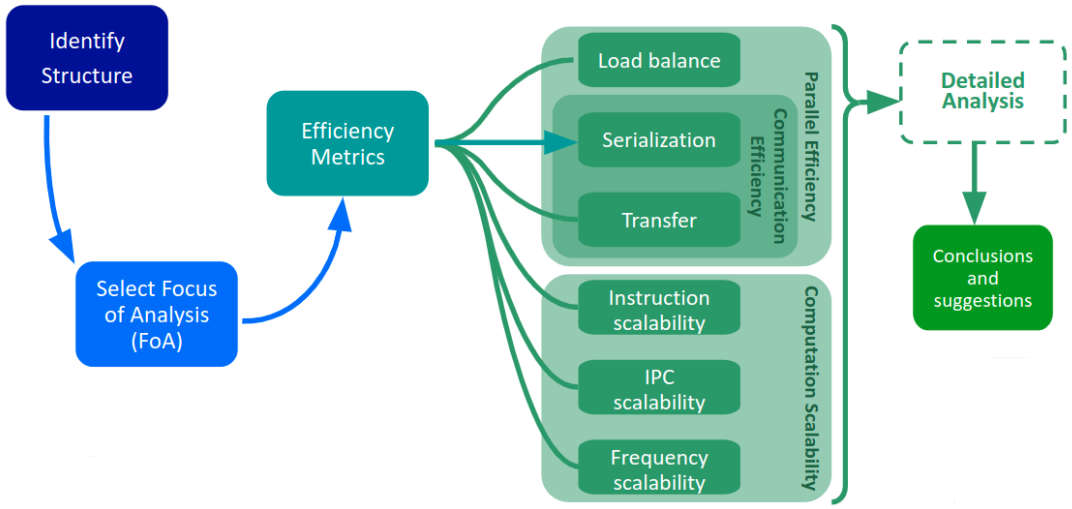

Figure 10: POP methodology scheme from [3]

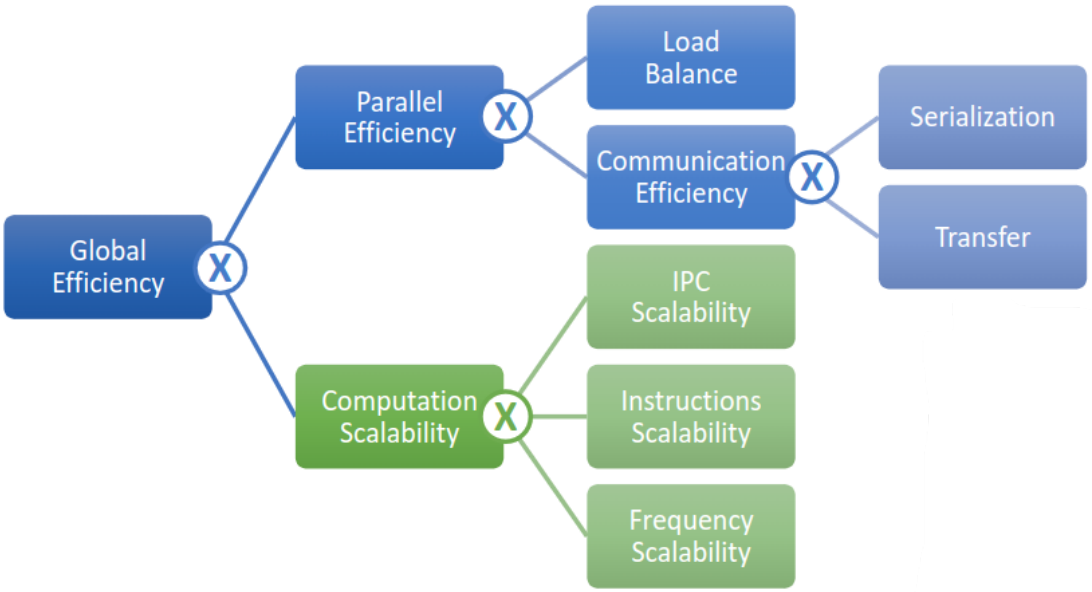

Figure 11: POP metrics relation from [3]

Metrics follow a hierarchical model and are multiplicative. All metrics are shown in percentages, but computation scalability ones are calculated with respect the base case, which is the first execution. Therefore, all first executions will show a 100% values in their IPC, Instructions and Frequency scalability, and the rest of the executions could show higher or lower values in comparison.

Computation scalability is calculated with instructions and cycles hardware counters, but parallel efficiency needs some clarifications:

- **Load Balance:** The efficiency loss due to the global distribution of work among processes or threads.
- **Communication efficiency:** The loss due to the communication of data, be it due to synchronizations between processes (or threads) or to the overhead introduced by the communication itself.

3 Implementation

3.1 Methodology

In order to understand this program's behaviour, we will perform several analysis, which will show the effects of altering the parameters in different ways.

First of all we have to perform a visual analysis of the execution trace to identify each one of the program phases during the training process and also offer a brief explanation of their functions inside the program execution.

When all phases are identified, we can proceed to their isolated analysis inside the traces. This is performed by zooming into each phase to analyse pricesely what happens in them. We will also compare behaviour and performance metrics of each phase for the different datasets used in order to highlight noticeable differences.

After the detailed analysis of the identified phases we will proceed with some parameter scaling tests. These tests will run several instances of the program altering only one of the execution parameters in order to see their impact on the overall execution performance and results.

At the end, the main goal of training a model is to improve the final accuracy of the network, therefore we will study how each dataset affects the overall accuracy and also we will apply what we learn about execution parameters to see if there's any improve in this matter.

3.2 Setup

In this section you can find a description of all necessary steps to configure our HPC system to run the application.

Container creation

It is a common practice to use containers in order to execute AI software, this way it becomes easier to solve runtime dependencies and using different execution configurations. Also, containerization allows us to better reproduce the executions, giving a fast way to migrate the experiment to another cluster if necessary.

In our case, we are using the Singularity [24] software. This container can be build by using the Docker [25] database with the Python basic tools to start creating our custom environment.

We also have a list of required Python modules and features, needed to execute our ResNet model in the PyTorch framework. This requirements can be found in Table 7.

We can build the container with the basic functionalities using two simple commands, found in Listing 1. It is also required to install our BSC data extraction tool, Extrae, from the local MN4 filesystem.

```
1 singularity build --sandbox torch_base  docker://python:latest
2 pip install -r requirements.txt
```

Listing 1: Building Singularity container

With the Singularity container already built, it is time to compile our main module: Torch.

Requirement	Version used
Cycler	0.10.0
Future	0.18.2
Kiwisolver	1.3.1
Matplotlib	3.3.3
Numpy	1.17.4
Pandas	0.25.3
Pillow	6.2.1
Pyparsing	2.4.7
Python-dateutil	2.8.1
Pytz	2019.3
Six	1.13.0
Torch	1.6.0
Torchvision	0.7.0

Table 7: List of Python required modules. Own creation.

PyTorch Compilation

Even though we could be using the precompiled version of Torch, downloaded during the setup, we want to achieve the best possible performance from the start, and to achieve that we must compile the module inside our HPC system. This way, our PyTorch framework instance will be able to use all our CPUs capabilities, such as AVX-512 instructions, and also will link against some libraries used in our system.

This is done not only to achieve the best initial performance, but to ensure that the system uses its own modified versions of some libraries, which can be optimized for this specific machine. First of all, we have to download the source code for the specific version we want to compile. These code lines can be found in Listing 2. This download will be performed in my local laptop and then copied into the system through scp in order to avoid connecting our HPC to the internet if it is not strictly necessary.

```
1 git clone --recursive https://github.com/pytorch/pytorch
2 cd pytorch
3 git checkout 1.6
4 git submodule sync
5 git submodule update --init --recursive
```

Listing 2: PyTorch v1.6 source download

The compilation process starts by setting up the compilation script with the appropriate flags. This script can be found in Listing 3, with all the necessary code lines to compile PyTorch framework.

We must pay attention to the library links, because we have to use a modified version of some of them, prepared to be used with Python. In fact, we have to execute the last 2 code lines in order to solve all Extrae's dependencies when interacting with Python.

```
export PYTORCH_DIR = <Path to output directory>
export PYTORCH_TAG = v1.6.0
export LIBOMPTRACE = <Path to libomptrace library>
export LIBGOMP = <Path to libgomp library>
export LIBS = /usr/local/lib
export PyLIBS = <Path to local Torch installation>
export USE_CUDA = 0
cd $PYTORCH_DIR/$PYTORCH_TAG/pytorch
python3.7 setup.py clean
python3.7 setup.py install

cp $LIBOMPTRACE $LIBS
cp $LIBGOMP $PyLIBS
```

Listing 3: PyTorch compilation script

With the compilation script prepared, we can create a batch script to be executed in MareNostrum 4. The idea is to compile the framework inside the Singularity container, without using any external source. This way, we can completely isolate the whole application from its compilation to its execution. In Listing 4 you can find the jobscript for the compilation of PyTorch inside the container environment.

```
#SBATCH --job-name="TorchComp"
#SBATCH -D <Path to working directory>
#SBATCH --output=torch_compilation_%J.out
#SBATCH --error=torch_compilation_%J.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=48
#SBATCH --exclusive
#SBATCH --time=3:00:00

module purge
module load singularity gcc/7.2.0 git cmake/3.15.4

SING_DIR = <Path to Singularity image>
singularity exec --writable $SING_DIR \
./pytorch_compilation_script.sh
```

Listing 4: PyTorch compilation script

Deploy and execute

Our application does not include any distributed computation module, therefore we can only use a full node allocation to run. All executions will be contained in one socket to avoid possible socket communication noise when trying to understand program's structure. This means that our maximum resources have been limited to:

- Maximum core count: 24 cores.
- Maximum main memory available: 384 GB.

In order to execute the application, we have to setup another batch script to define what is going to be executed inside the container and what outside of it. In fact, we created two different scripts.

The first one, described in Listing 5, sends a job to assign resources to the Singularity container and sets the path for the container to run the application inside. This means that our container will access another script when it finishes the deployment. In the example of Listing 5 we call the execution for the Low Resolution dataset. We use the "taskset" command to ensure our process will not jump between cores while in execution and also the "numactl" command to force the execution to be located in the socket we are using.

```
1  #SBATCH --job-name="MN_run"
2  #SBATCH -D <Path to working directory>
3  #SBATCH --output=torch_run_%J.out
4  #SBATCH --error=torch_run_%J.err
5  #SBATCH --ntasks=1
6  #SBATCH --cpus-per-task=48
7  #SBATCH --exclusive
8  #SBATCH --time=3:00:00
9
10 module purge
11 module load gcc/8.1.0 singularity
12
13 export LRU_CACHE_CAPACITY = 1
14 export OMP_NUM_THREADS = 24
15 export OMP_PROC_BIND = TRUE
16
17 JOB_DIR = <Path to Jobscripts directory>
18 SING_DIR = <Path to Singularity image>
19
20 numactl --cpunodebind=0 --membind=0 \
21 taskset -c 0-23 \
22 singularity exec $SING_DIR $JOB_DIR/run-downsample.sh
```

Listing 5: Singularity start script

The second one is accessed within the containerized environment and prepares for launching and tracing a specific dataset. This code can be found in Listing 6, showing an example of how to load the Low Resolution dataset, setting all Extrae variables including the trace naming based on parameters. Preloading the above mentioned libraries is mandatory to use the Extrae version compatible with Python tracing.

```
export BS=32
export gs=32k
export learning_Rate=0.0001
export dataset='lrfs'

BENCH_DIR= <Project path>

export PYTHONPATH=$BENCH_DIR:$PYTHONPATH

export EXTRAE_HOME = <Path to EXTRAE installation>
source $EXTRAE_HOME/etc/extrae.sh
export EXTRAE_CONFIG_FILE = <Path to XML configuration file>
export TRACE_FILE = mn4_${OMP_NUM_THREADS}C_
                    trace_${dataset}_bs${BS}_gs${gs}_lr${learning_rate}.prv

LD_PRELOAD = <LIBS path>/libomptrace.so:<PyLIBS path>/libgomp-7c85b1e2.so.1 \
python3 $BENCH_DIR/benchmarks/resnet18_MAMeBench.py \
lr_${BSC_MACHINE}_normal_train --datasetsize=$dataset --train --traceit --BS=$BS
```

Listing 6: Dataset launch script.

At this point, we are able to deploy the container, run the application with every dataset inside this environment and also trace the execution.

What to measure

As commented in previous sections, during this project we want to measure not just the POP defined metrics, but also those metrics monitored by AI developers, which tend to measure their application performance by reading the accuracy of the predictions and the elapsed time of the whole training process. The selected hyperparameters to be studied during this project are:

- **Batch Size:** amount of images to be computed at the same time per training step. The total steps to run a full epoch depends on this value. Higher values require more main memory.
- **Learning Rate:** value used as a multiplier for the accumulation of weight values. The weight value decides whether or not a neuron is more likely to be activated during the execution.

We will also test different **Grain Size** values, which stands for the minimum amount of work that can be assigned to a running thread, in bytes.

Thanks to Extrae custom events we are able to study the structure of the application by using them to trace when Forward, Backward, Optimization and Batch loading phases are being executed. This method will help us to filter the trace by those phases and check its individual behaviour regarding Load Balance or IPC among other metrics.

3.3 Instrumentation

In order to produce all the data we need to effectively analyse this program's structure and behaviour we had to modify some parts of the code.

The first modifications come from the training code, where we can use Extrae's API to enable registering when an events start and finish. In this case we have to modify the original code as shown in the example from Listing 7.

```
1 pyextrae.eventandcounters("phases", 1)
2 training.batchLoading()
3 pyextrae.eventandcounters("phases", 2)
4 training.forward()
5 pyextrae.eventandcounters("phases", 3)
6 training.backward()
7 pyextrae.eventandcounters("phases", 4)
8 training.optimization()
9 pyextrae.eventandcounters("phases", 0) # Last phase to be called
```

Listing 7: Creating events to trace different phases.

When focusing inside the different phases we wanted to know exactly which layers were being executed during the forward and backward phases. This pushed us to modify the code that defines how the resnet model is built in the Torch library. This modification was executed in the same way as seen in Listing 7 but adding the events between each layer construction call.

Our final instrumentation step is related to the operations executed during the training process. In order to know which operation has been called in the PyTorch context we had to modify the Torch API that handles events. In particular we modified the "Event" structure inside the framework to also register the operation's name that triggered the event. The portion of code needed to enable this tracing is shown in Listing 8.

```
[...] # structure initialization

extrae_type_t type = 9300001;
char* description = "Pytorch_Operaciones";
unsigned nvalues = 1;
auto name_str = name.str();

static std::map<std::string,unsigned int> extrae_funcs;

if(kind == EventKind::PopRange)
        Extrae_eventandcounters(type, (extrae_value_t)0);
else if(kind == EventKind::PushRange){
  if(extrae_funcs.find(name_str) == extrae_funcs.end()){
        extrae_funcs[name_str] = extrae_funcs.size()+1;
        extrae_value_t value = extrae_funcs.size()+1;
        char* value_descr[1];
        value_descr[0] = (char*)name_str;
        Extrae_define_event_type(&type, description,
                        &nvalues, &value, value_descr);
        }
        Extrae_eventandcounters(type,
                (extrae_value_t)extrae_funcs[name_str]);
}
```

Listing 8: Creating events to trace different operations.

With this codes we are now able to visualize operations, phases and layers to better understand the program structure.

3.4 Program structure

With the first traces, in Figure 12, we can identify how our model executes the whole computation process by distinguishing different program phases. All these phases are executed in the same order at each training step.

1. **Batch Loading:** During this phase the next images to compute are loaded in the batch structure. When using non squared images, padding pixels are added before starting the following phase.

2. **Forward propagation:** Images are fed into the input layer. All pixels are represented by numerical values that denote the intensity of each one. In fact, these values correspond to the RGB values, and layers apply mathematical operations on them. This phase generates a first output to be fed into the Backward phase.

3. **Backward:** Receives an output approximation and compares with the actual value. Based on the difference, or error, between the value and the approximation, the values of the parameters are updated. This process will force the network to produce a slightly different approximation when the following Forward phase finishes.

4. **Optimization:** In this phase an algorithm is executed to modify the weights of the neural network, in order to minimize the error in following steps. This phase depends on the network design, not on the input received by the model. Therefore we should not notice meaningful variations when comparing with other datasets.

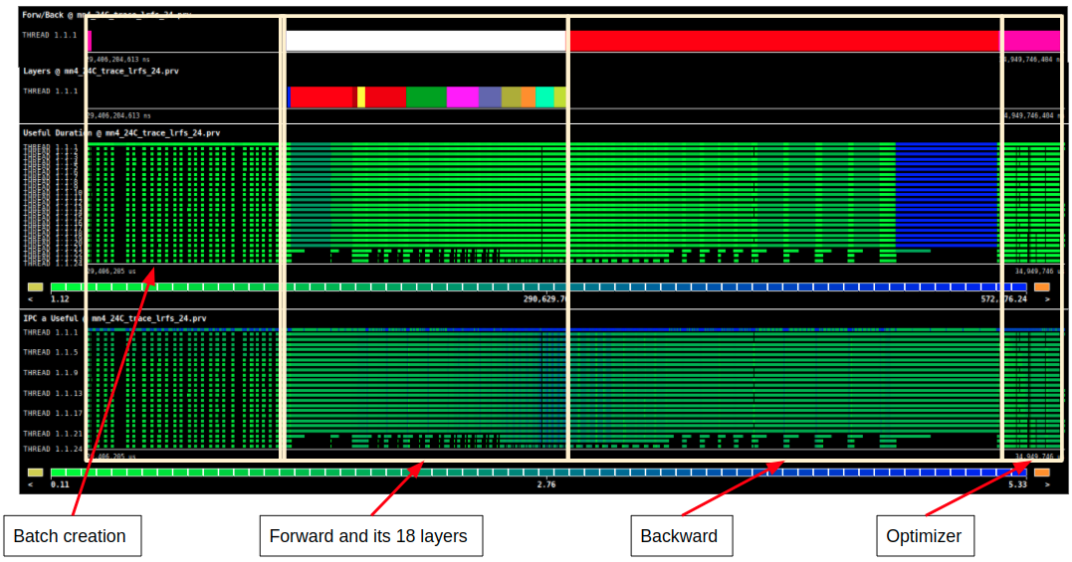

Figure 12: Full step phase dissemination. Own creation.

Figure 12 is a composition of views that show, from top to bottom:

- Which phase is being executed at each timestamp, distinguishing Batch loading, Forward, Backward and Optimization by colors black, white, red and purple respectively.
- Layers that are explicitly called during execution. These can only be detected during the Forward propagation phase.
- Useful duration view, showing how much time each thread has spent into effective computation of our program.
- IPC value for each thread when performing useful computation.

On following sections we will analyse each identified region to obtain a detailed view of their performance and behaviour, and the overall impact on the whole execution.

4 Analysis of size and shape

This part of the study is based on [26], where we tackled for the first time the complexities of using different datasets with this model.

4.1 Time study

First of all, we want to distinguish which is the time scale difference when comparing the three datasets. Given the characteristics of the images, when using LRFS data we can use more images per batch in comparison with MR and HR, therefore, those batch sizes have been modified for each dataset.

The goal is to use the maximum possible Batch Size value for each dataset given the amount of main memory we have that also allows us to compute the full dataset. With the addition of padding pixels into MRVS and HRVS datasets, the total amount of pixels may differ between steps and this scenario can generate a non computable step if the batch exceeds our available memory. The maximum batch size values for each dataset and their total amount of steps to perform are summarised in Table 8. These values have been found through trial and error executions until finding the maximum value that can fit into memory.

Dataset	Batch Size	Epoch Steps
LRFS	1024	20
MRVS	32	632
HRVS	2	10.111

Table 8: Batch Sizes and steps summary. Own creation.

In Figure 13 we can see the elapsed time per step to compute informative megapixels (those which are part of an image) per step. LRFS dataset has an extremely low variation in execution time from one step from another because no padding pixels are added in the batches, therefore each batch will contain exactly the same pixels except for the last one, where only the remaining images are present.

When looking into MRVS dataset we can see huge differences between steps. These variations come from the differences in the batched images, as each batch will contain distinct amounts of padding and informative pixels.

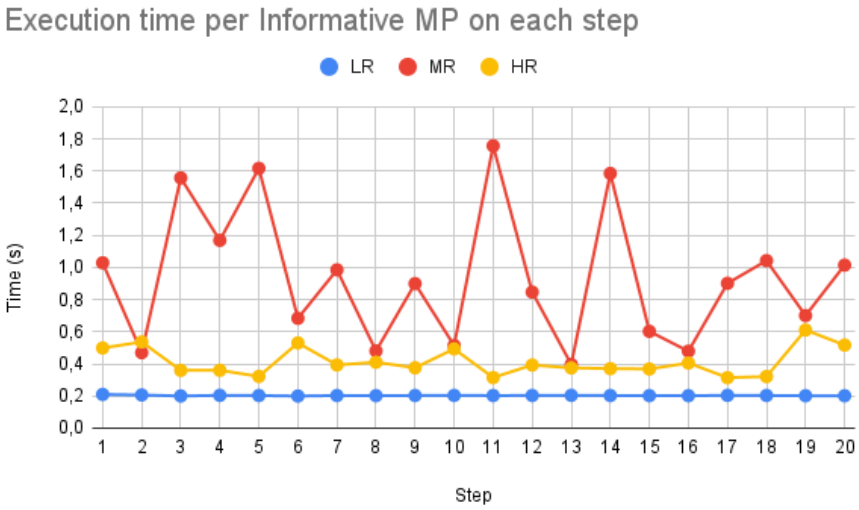

Figure 13: Informative megapixel execution times. Own creation.

HRVS dataset results show less variation if compared with MRVS. As these images are processed with their original resolution, the relative amount of padding pixels compared with the informative ones is lesser than in MRVS. If less padding is added proportionally, more computation is used in the informative pixels.

Figure 14 represents different steps in the X axis and the elapsed time per megapixel in the Y axis. We can see that MR is the one that needs less time to compute a megapixel, as we discussed before MR is the data set that contains more padding pixels per batch. Our hypothesis is that padding pixels are cheaper to compute than real pixels.

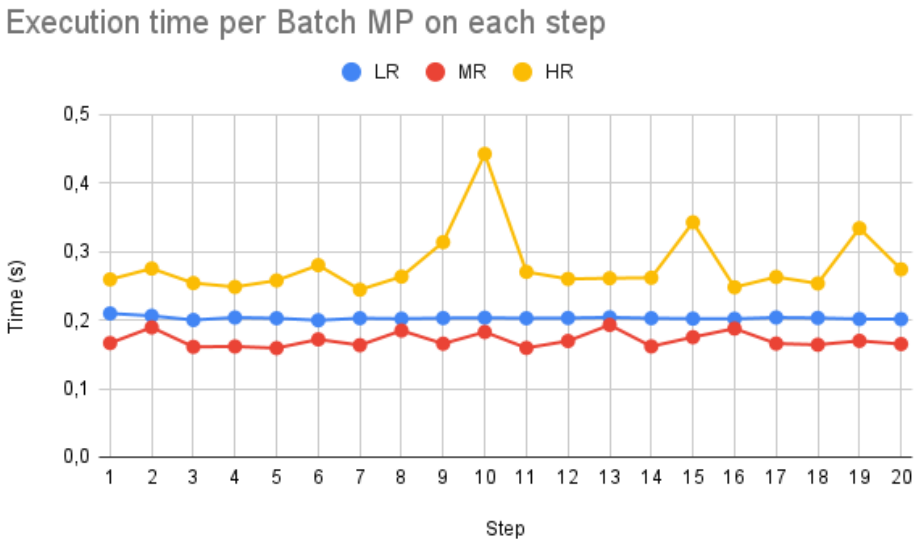

Figure 14: Batch megapixel execution times. Own creation.

Analyzing the HR execution time in Figure 14 we see that steps 10, 15 and 19 take longer to compute than other steps. These particular steps contain images with similar aspect ratios, which seem to have an impact on performance.

4.2 Pixel study

When trying to use images with their original resolution and shape, we find that almost all high resolution images tend to have a non squared geometry. We have explained that, in order to create a squared batch of images needed to feed the CNN, padding pixels are added into the batch through a technique called random batching. This method also has an effect on the amount of instructions executed, as these pixels must be processed because they belong to the batch.

After the execution of a full epoch, which consists on processing one time all the images in the dataset, we have built Figure 15, where you can find the total amount of informative and padding pixels computed during the full epoch on each dataset.

Dataset	Informative pixels	Padding pixels	Total pixels	% Info pixels in total
LRFS	1.325.203.456	0	1.325.203.456	100,00
MRVS	7.330.186.500	25.605.984.648	32.936.171.148	22,26
HRVS	204.148.274.758	107.305.057.588	311.453.332.346	65,55

Figure 15: Full epoch pixel count comparison. Own creation.

If we compare LRFS and MRVS, we can spot a 5,5x increase in the informative pixels count, directly related to the MRVS images shape. As those images maintain their original aspect ratio, this increase is expected even when resolution is still lower than the original.

Along with the informative pixels increase, we can also notice that the amount of padding pixels is higher than the informative ones. In fact, during the whole MRVS dataset execution, we we process too many padding pixels, as the informative ones only represent around 22% of the whole input.

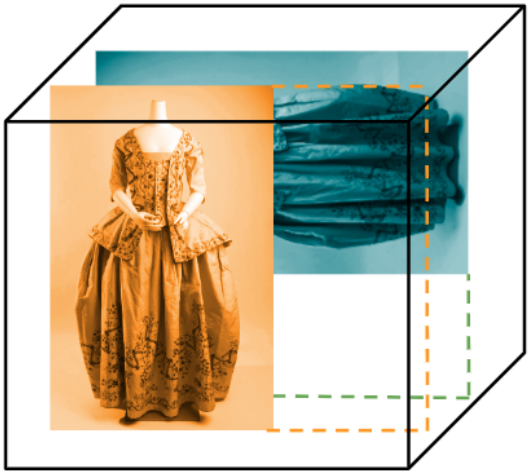

Figure 16: Theoretical batch representation with two images with different aspect ratios. Own creation.

In Figure 16 we have built a representation of a possible batch with two images, each with different shape, showing with dotted lines the gaps where padding pixels must be added to obtain a square.

In this scenario, we are computing way more padding pixels by filling the batch without considering the arrangement of the informative pixels, generating holes. Better arrangements could reduce the amount of padding pixels at the cost of executing a certain arrangement algorithm, and its costs must be paid at each batch loading phase if no further modifications are implemented.

Referring again to Figure 15, if we compare MRVS with HRVS we can observe not only the effect of variable shapes, but also the impact of the total resolution of each image. With HRVS we are computing approximately 28x more informative pixels than MRVS, but only around 4x more padding pixels, increasing the relation up to 65% of useful information processing during the full epoch.

For AI practitioners, the execution time of the model training is key in order to fasten the model's adaptation capabilities. Adding new data to the training sets allows to re-train the model to get better predictions but, if training requires a huge amount of time, this process must be delayed or scheduled in order to train only when the prediction gain should be noticeable, if any.

5 Detailed analysis per phase

5.1 Batch loading

The first test to be made is based in varying the batch size value to see its impact on the execution. We executed the LRFS dataset with half the batch size used in the first tests in order to see if this value has an impact in the overall execution time. This comparison is found in Figure 17 with two execution traces. The top trace belongs to an execution with a batch size of 24 and the bottom one with a batch size of 12. Both traces show the useful IPC for each thread involved in the computation.

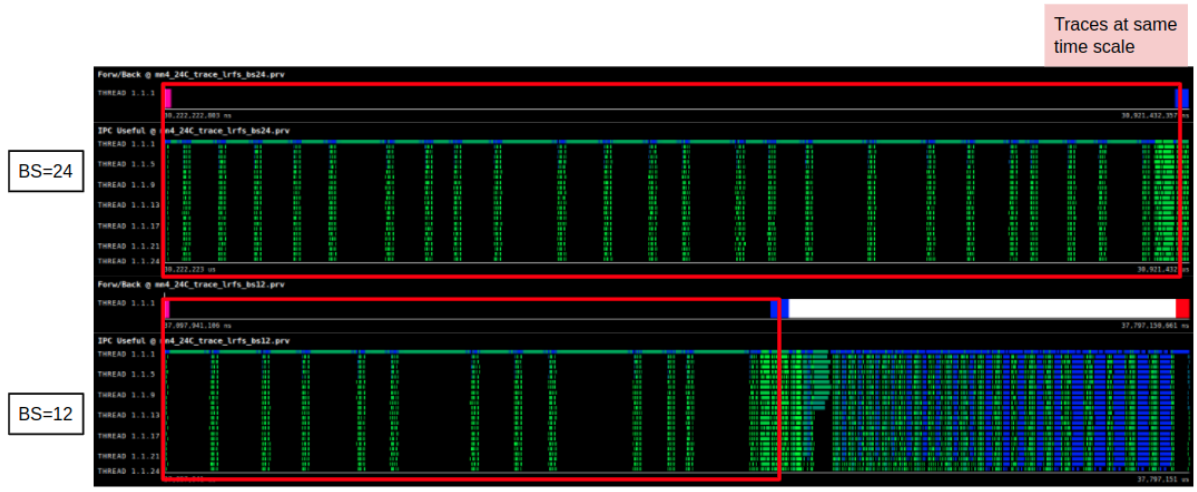

Figure 17: Batch size value effects. Own creation.

Comparing just the batch loading region, we can say that halving the batch size also reduces its execution time, but it is reduce 1/3 of the original one, not half as expected if the execution time was propotional to the batch size.

In the traces we observe some parallel regions that are opened and closed that identify where the different threads are working (the vertical columns with green colour in the trace). We can see that the number of parallel regions coincides with the batch size used in each execution, but also there is a good portion of time where only one thread is active.

If we zoom into the red rectangles from Figure 17 we can create Figure 18 showing a trace view of the IPC at the top and another view of the OpenMP calls at the bottom. These traces reveal how the first thread opens parallel regions for each one of the images, indicating that each individual image is, in fact, split to be processed in parallel, creating and closing a parallel region for each one.

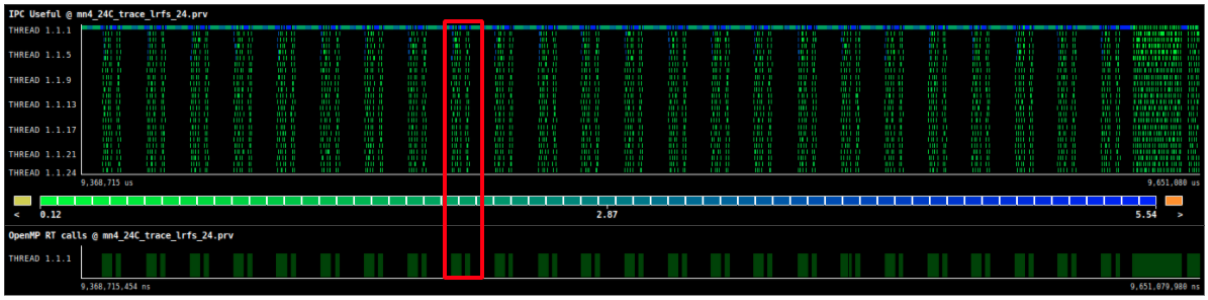

Figure 18: Batch loading columns matching batch size value. Own creation.

If we zoom into one of the parallel regions we can observe how each thread is performing on the task by using the IPC view of our trace. In Figure 19 we show a comparison of the batch loading phase on each dataset, focusing in one of the image loading bursts.

When looking at MRVS and HRVS datasets, we can spot a difference with LRFS: all threads can achieve a high IPC value in the first part. In fact, LRFS can only use 6 of the available threads during the loading process.

An explanation to this can be found taking into account that each parallel region is used to load just one image:

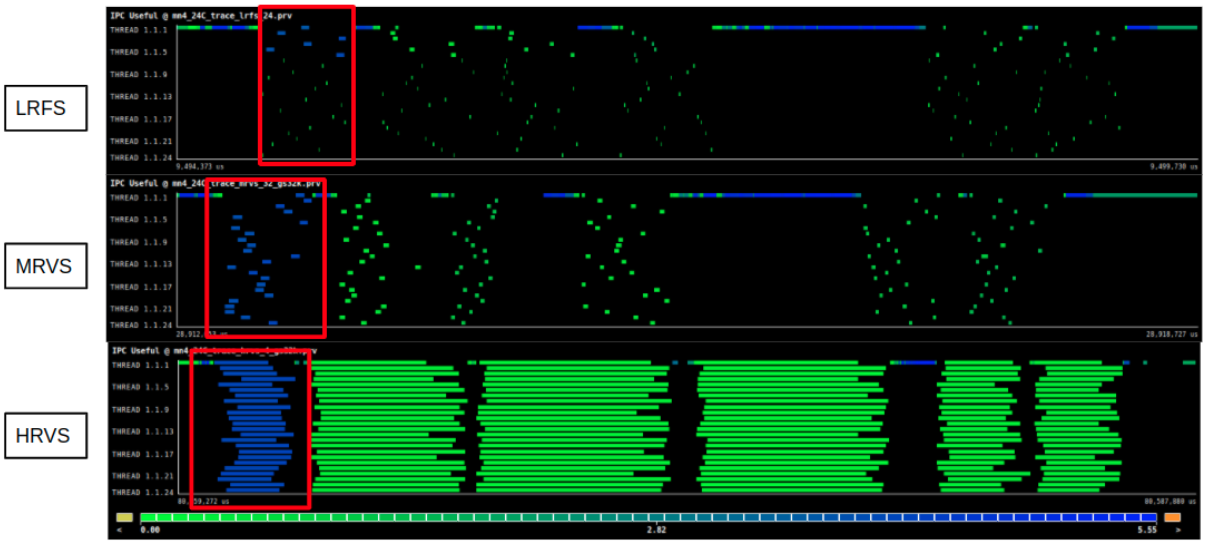

Figure 19: Batch loading datasets parallelism comparison. Own creation.

- Each one of the LRFS images has 256^2 pixels, and each pixel needs 3 bytes to express the RGB color, therefore the work chunk for this process results in a block of 192 KBytes.
- By checking PyTorch's parallelism policy, we found that the default grain size for each spawned thread cannot be lower than 32 KBytes. Grain size is further studied in following sections.
- Being that $\frac{192kB}{32KBxThread} = 6$ threads, we can prove that LRFS is only using 6 of its threads because images are too small.

With that in mind we can state that LRFS images will not take profit of a high core count during executions, but MRVS and HRVS will. It is also clear that increasing batch size value will not affect in any way to this phase's parallelism when using low resolution images.

	LRFS	MRVS	HRVS
Global efficiency	3,89	198,26	123,45
-- Parallel efficiency	3,89	11,07	15,23
-- Load balance	4,38	15,12	16,06
-- Communication efficiency	88,78	73,20	94,82
-- Computation scalability	100,00	1.791,74	810,40
-- IPC scalability	100,00	69,94	43,20
-- Instruction scalability	100,00	1.475,12	694,72
-- Frequency scalability	100,00	173,66	270,05
-- Average IPC	2,639	1,846	1,140
-- Average frequency (GHz)	0,560	0,972	1,512
-- Runtime (s)	35,285	0,691	1,110

Figure 20: Batch loading metrics comparison. Own creation.

In Figure 20 we see a comparison of the different metrics we can obtain using the POP modelfactors script from the Basic Analysis tools, regarding the batch loading phase of each dataset. This table reveals how MRVS and HRVS can achieve better load balance than LRFS during this phase, expected once seen Figure 19 parallel behaviour.

Variable shaped datasets require less instructions to complete this phase and also achieve higher frequencies than LRFS executions. This instruction count reduction may be related with the amount of pixels to be loaded in the batch. Although IPC is higher when using LRFS dataset, computation scalability is greatly improved when using variable shaped images.

Figure 21 shows the total pixel count when computing the particular step used to create Figure 20. We can see that pixel count follows LRFS > HRVS > MRVS, and coincides with the instruction scalability order. This pattern is repeated when measuring the batch loading phase of other steps.

Dataset	Pixels computed on this step
LRFS	67108864
MRVS	26587008
HRVS	30669912

Figure 21: Pixel count comparison in the test step. Own creation.

The IPC reduction could be highlighting the effects of adding the padding pixels during the batch creation, because it represents an extra step during this process that LRFS dataset does not perform, but MRVS and HRVS do. Another explanation for this IPC reduction is related to the individual images size. As MRVS and HRVS images have increased sizes when compared with LRFS ones, we could expect extended waiting times until a particular image is loaded in memory, therefore reducing the average IPC of this particular region when using bigger images.

5.2 Forward propagation

This phase receives the batched images and applies a series of mathematical operations over the image pixels. These operations are performed inside the so-called interconnected layers.

In Figure 22 you can find two views of the same trace from an LRFS execution, showing the layers traversed during the forward propagation phase in the first row and the IPC achieved by each thread in the second. Each executed layer is related to a color in the first row and all layers are always executed in the same order: conv1, bn1, ReLu, maxpool, all convolutional layers, averagepool and the fully connected one.

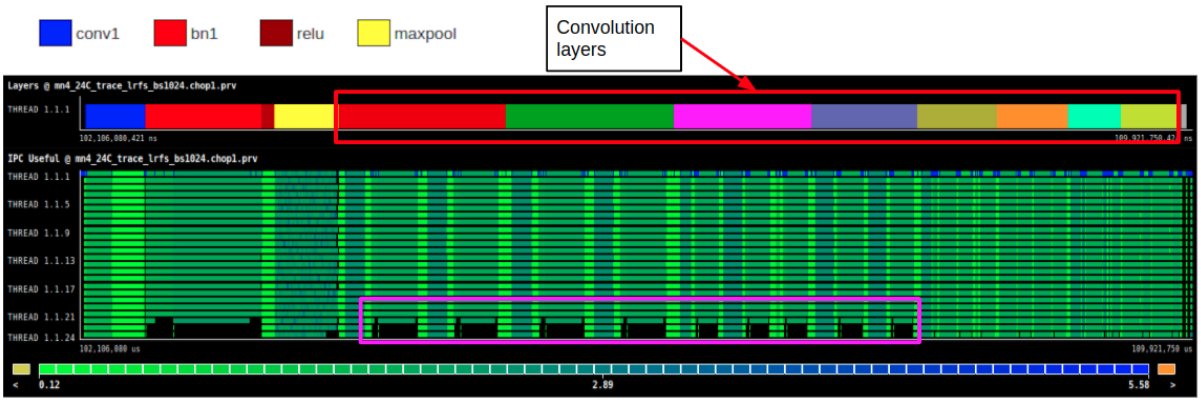

Figure 22: IPC and layers views on Forward propagation. Own creation.

We can spot how in the first layer (conv1) all threads are active, but this changes during batch normalization, leaving the last 2 threads unused. This pattern is repeated in other layers.

During the first convolution layers we can see that not all threads are being used at some points, showed in the pink rectangle. When reaching the last convolution layer, more threads can be profited and also require less time to finish. As each convolution reduces the output volume, requiring less execution time near the last convolution layers is expected. In fact, if the volume is being reduced while trespassing convolution layers, why more threads can be used when reaching the end?

In Figure 23 we see a zoomed in view of two convolution layers. The first row represents the operations that PyTorch framework has launched, the second one shows the layers selected just to clarify at what time the layer changes, and the third is the IPC view of all working threads. Thanks to PyTorch operations tracing (manually done with custom Extrae events), we can see how convolution operations are taking profit of all our threads on both layers. Thread imbalance occurs when select operations are executed, just after convolutions.

Both convolution and select operations tend to require less execution time at each layer. Convolution time depends on the input size and the filter submatrix used and, as all filters have the same dimensions, this time reduction can only come from the input size reduction from previous layers.

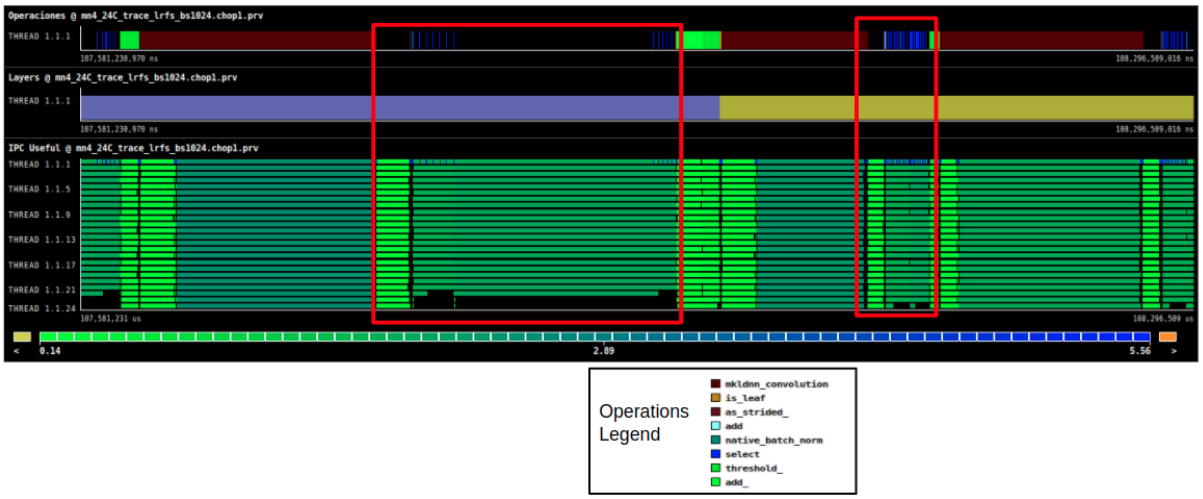

Figure 23: Convolution and select operations. Own creation.

The select operation accesses the output tensors of the convolution to slice them through a specific dimension, offering a view of the input tensor without this dimension. If tensors contain less values due to batch volume reduction after the convolution, then select operations will also execute faster. As dimension count in those tensors does not change, we can say that parallelism in this operations is not related to the amount of dimensions, but it could be related with the amount of elements on each dimension.

In order to better understand the effects of our datasets in this phase, we have Figure 24 showing a whole forward propagation phase using MRVS and HRVS datasets.

The first two rows belong to the MRVS execution, and the last two to the HRVS one. When comparing layers of MRVS with LRFS from Figure 22, we see a similar pattern, with an extension of the conv1 layer duration in the MRVS dataset. We can also spot a huge difference between the three executions regarding the maxpool layer, where only LRFS is able to use all available cores.

When counting the number of active threads in maxpool layer, we see that LRFS uses all available cores; MRVS is able to use 16 cores and HRVS uses only 2 of them. Further testing shows that maxpool layer parallelism depends on the batch size value, so parallelism is at image

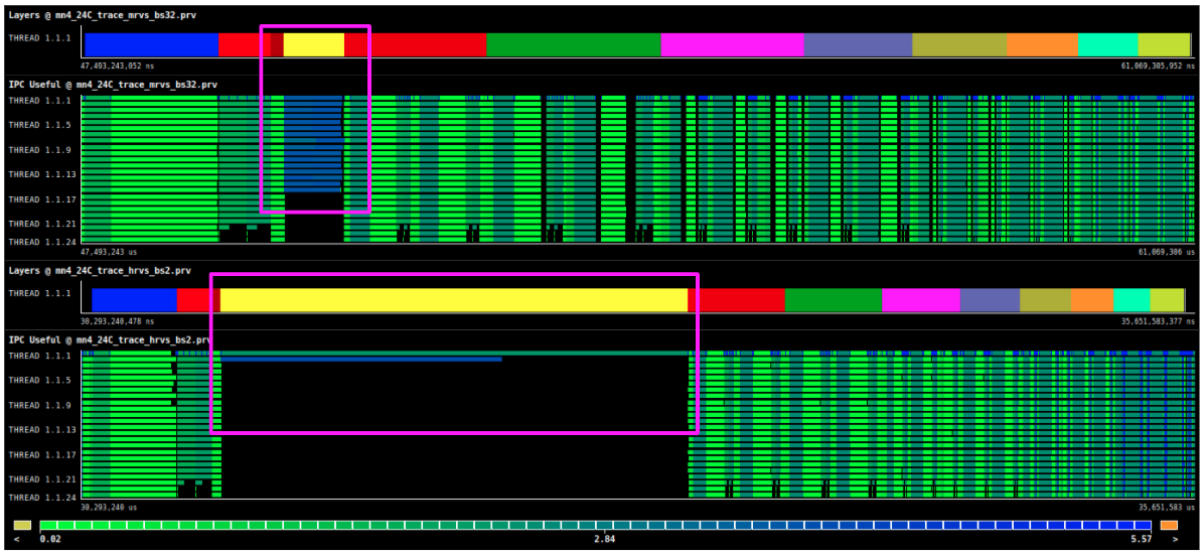

Figure 24: HRVS and MRVS forward phases comparison. Own creation.

level. This means that each thread can process one image while another thread processes another image.

To clarify, our LRFS execution is using all 24 cores with a batch size of 1024 images, but the last threads are finishing earlier because, even when all images have the same size and shape, 1024 is not multiple of 24, therefore these last threads will end their jobs earlier due to lack of more images. In Figure 25 you can see the IPC of the maxpool layer of LRFS (second view) and MRVS (first view), showing the points where a thread finishes processing an image to start with the next one, in pink rectangles. This behaviour reveals that this layer is easier to balance when using more images (increasing batch size).

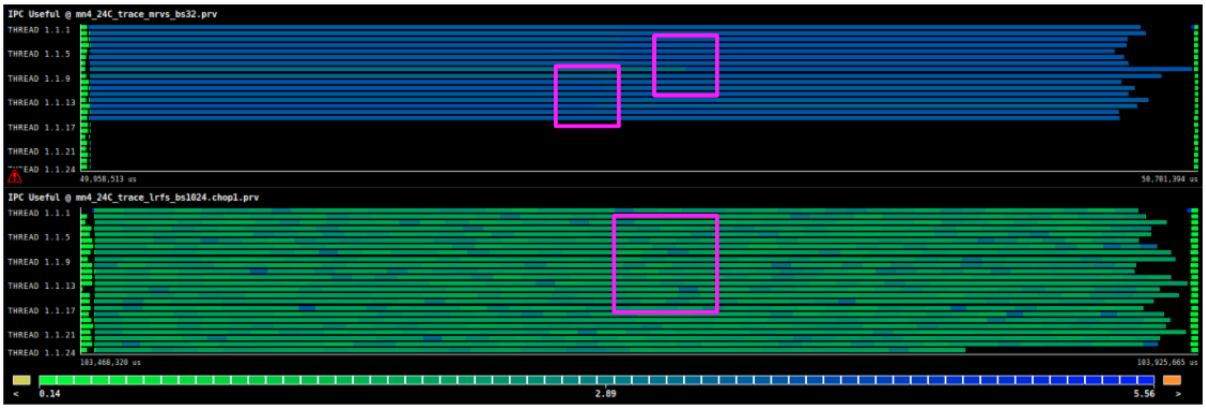

Figure 25: LRFS and MRVS maxpool layer IPC comparison. Same semantic scale. Own creation.

As MRVS is executed with a batch size value of 32, we can ensure each of the 16 working threads will be processing two images. In this case more threads could be profited, resulting in a lot of threads processing one image and some of them processing two. This scenario would not throw any execution time gain compared with the 16 thread execution, as all images must be processed before finishing the layer anyways.

In LRFS execution view is easier to spot the points where an image has been finished and the new one starts, just by looking when the IPC has a variation inside the pink rectangle. In

this execution we see the maximum thread usage possible during the maxpool layer. We can confirm that even when each thread processes more than one image, the least possible execution time is given by the thread with more workload, forcing others to wait.

Convolution operations behave similarly with all datasets, even when changing core count and batch size values, offering no differences in trace views.

	LRFS	MRVS	HRVS
Global efficiency	92,47	145,94	90,81
-- Parallel efficiency	92,47	89,75	71,87
-- Load balance	95,03	93,99	74,98
-- Communication efficiency	97,31	95,48	95,86
-- Computation scalability	100,00	162,61	126,35
-- IPC scalability	100,00	90,69	82,02
-- Instruction scalability	100,00	129,42	106,14
-- Frequency scalability	100,00	138,54	145,14
-- Average IPC	1,621	1,470	1,330
-- Average frequency (GHz)	1,249	1,731	1,813
-- Runtime (s)	7,582	4,804	7,720

Figure 26: Forward phase metrics comparison. Own creation.

In Figure 26 you can find a summarised table with our beloved metrics, regarding the forward phase of each dataset. As seen with the batch loading phase, HRVS dataset tends to have lower IPC than LRFS and MRVS, but also has better frequency values. Load balance decreases when using HRVS dataset, as seen in Figure 24, this decrease comes from the execution of maxpool layer, which represents a good chunk of the forward phase computation time and, in our case, can only use 2 threads of the 24 available.

5.3 Backward propagation

Backward is the function which actually calculates the gradients by passing its argument through the backward graph all the way up to every leaf node traceable from the calling root tensor. This backward graph is generated dynamically during the forward pass, therefore backward only calculates the gradients using the already made graph and stores them in leaf nodes.

In essence, this phase consists on following the path made by the forward phase in order to prepare to update the weights, or neuron activation probabilities, based on a loss function. This function will calculate the difference between the expected output and the obtained one, generating the gradient value for each neuron of the network and storing it to update weights later.

Figure 27 shows the whole execution of the backward propagation phase, using LRFS dataset with a batch size of 1024 images. The first row shows the operations performed by PyTorch and the second the IPC obtained for each active thread.

We can see a familiar pattern in thread usage that behaves as forward phase from Figure 22, but inverted. This was expected given that backward phase executes the same layers as the forward phase but in reverse order, but some points must be highlighted here.

At first glance we can see that this phase executes convolution operations, but those are different from the ones executed in forward phase. One of them uses the output of the previous layer to generate the loss gradient and the other uses this loss value to propagate it to superior layers following the chain rule adapted to convolution operations.

Both convolutions are needed to generate the output from the previous layer, which will be

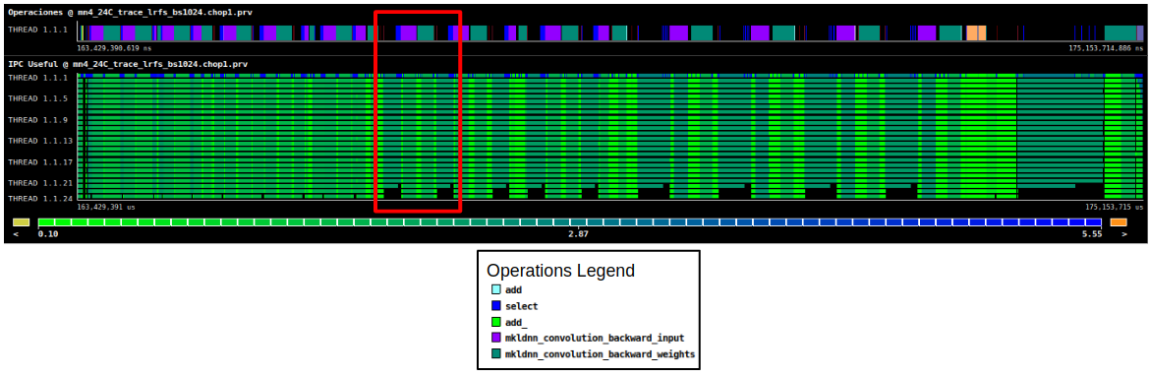

Figure 27: LRFS backward phase. Own creation.

tackled again with both convolutions. As seen during forward propagation phase, convolutions can be executed using all available threads, but select operations cannot.

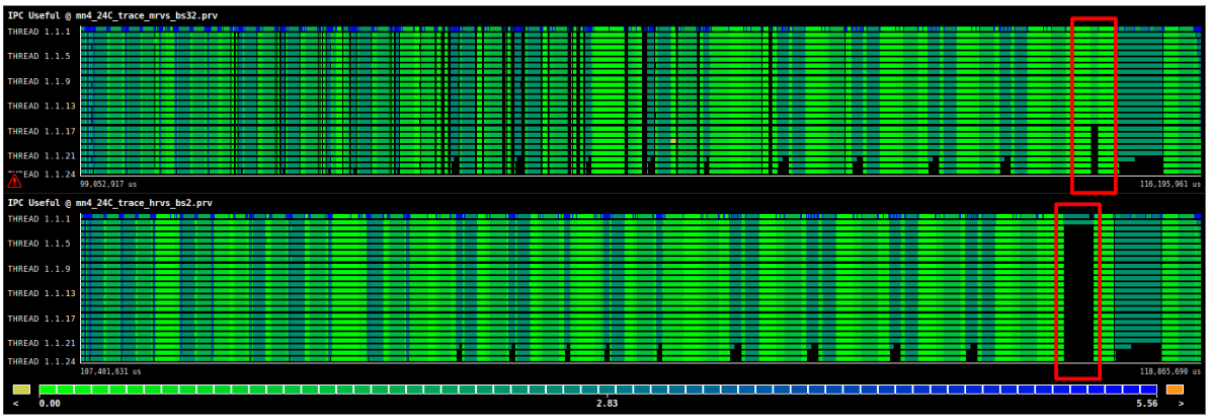

Figure 28: MRVS and HRVS backward phases. Same semantic scale. Own creation.

In Figure 28 you can see the whole execution of the backward propagation phase when executing MRVS (first row) and HRVS (second row) datasets. As explained, backward operations are derived from the forward pass design. With this in mind, we can ensure that the regions marked with the red squares belong to the maxpool layers from forward pass, as they show the same parallel behaviour. Although these regions use the same amount of cores, the impact on the overall backward phase execution time is lesser than the impact on forward phase.

	LRFS	MRVS	HRVS
Global efficiency	92,03	134,88	103,33
-- Parallel efficiency	92,03	91,26	91,65
-- Load balance	96,11	96,38	96,22
-- Communication efficiency	95,76	94,69	95,25
-- Computation scalability	100,00	147,80	112,74
-- IPC scalability	100,00	93,69	82,86
-- Instruction scalability	100,00	131,35	106,78
-- Frequency scalability	100,00	120,10	127,43
-- Average IPC	1,466	1,373	1,215
-- Average frequency (GHz)	1,401	1,683	1,786
-- Runtime (s)	11,518	7,859	10,259

Figure 29: Backward phase metrics comparison. Own creation.

Figure 29 shows a table with the efficiency metrics regarding the backward propagation phase. In this case, differences between datasets are lesser than those observed in forward phase. During this phase, HRVS dataset is able to execute less instructions with a higher average frequency than LRFS, but IPC is lower. Also, differences in load balance are lower than the ones seen in forward phase in Figure 26, probably due to the maxpool layer execution time being less meaningful in this region.

5.4 Optimization

Optimization in this particular scope refers to the learning capability of the Resnet model. Our case runs with an optimization algorithm called Adam [27], which is a replacement optimization algorithm for training deep learning models by combining previously developed algorithms.

Once all gradients have been stored during backward propagation, the next step belongs to neuron activations optimization. During this phase, all neurons must be visited in order to modify their weight to better fit predictions once the model's training step finishes. This iterative pattern following the layer tree upwards leads to think that this phase may not depend on the input but on the network size (amount of neurons), as no operations must be performed over the images here.

In Figure 30 you can find two trace views representing IPC at the same time scale from two different executions using HRVS and LRFS datasets. The top one shows the execution with batch size 2 using HRVS and the bottom one uses batch size 1024 with LRFS images. We can see how LRFS and HRVS are using all threads during really short bursts. This algorithm visits all our network's neurons performing mathematical operations such as additions and multiplications, that are executed during these short bursts.

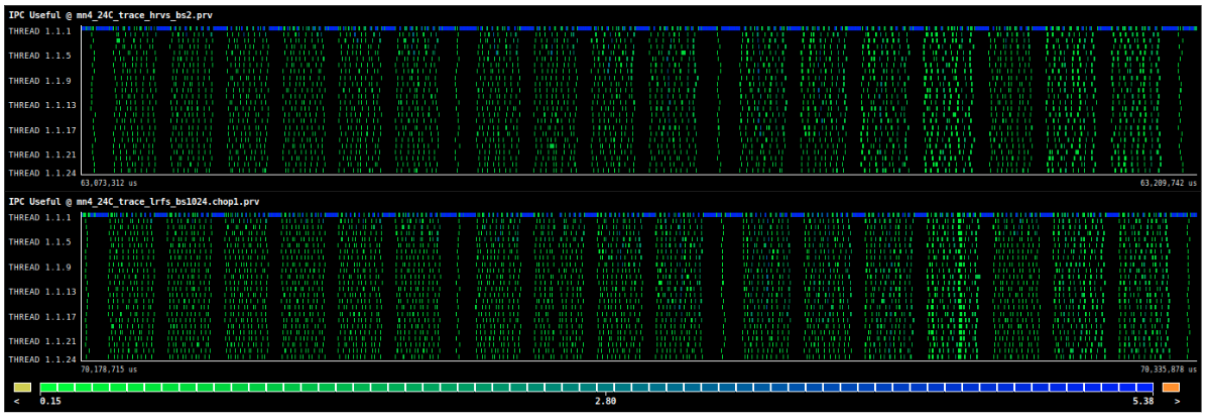

Figure 30: LRFS and HRVS optimization phase comparison. Own creation.

When analysing more training steps in all datasets and using batch size variations, no differences are found in this phase's execution, making clear that during the optimization phase no difference can be made by using distinct input sets.

Parallelism behaviour remains equal with all datasets and the execution time of this phase has very little variance on every executed step and every dataset used, therefore we can state that this phase will not be affected by the images size or shape, nor the amount of images in a batch.

6 Core scaling study

In these tests we are assigning more cores to the application in the form of OpenMP threads and limiting the hardware concurrency so that, when using 2 threads, these will be assigned to the first 2 cores of the machine and no other cores. All tests run the application using 15 steps and batch sizes 1024, 32 and 2 for LRFS, MRVS and HRVS datasets respectively.

Figure 31 shows the efficiency metrics obtained when training the CNN with the LRFS dataset using different number of cores (OpenMP threads). We can see that parallel efficiency decreases when adding more cores due to load balance issues, showing huge drops when going from 4 to 8 cores. This fact can be explained by addressing the parallelism policy of the batch loading phase. Assigning more threads leads to more cores in idle state when processing the images, as work distribution relies on each single image size.

The gains in execution time scale until hitting around 3x speedup when using 16 or more cores, but no improvements are measured past this value. We can see frequency and IPC drops when surpassing 16 cores.

	LRFS dataset							
Core count	**2**	**4**	**8**	**12**	**16**	**24**	**36**	**48**
Global efficiency	91,60	75,25	51,18	43,69	35,48	22,92	17,10	11,71
-- Parallel efficiency	91,60	77,43	56,78	51,72	47,24	40,08	28,68	23,17
-- Load balance	91,96	78,21	58,26	54,11	50,48	43,97	34,27	29,43
-- Communication efficiency	99,62	99,00	97,46	95,57	93,58	91,15	83,69	78,70
-- Computation scalability	100,00	97,18	90,13	84,49	75,10	57,18	59,62	50,56
-- IPC scalability	100,00	98,93	96,55	92,88	87,05	77,59	74,14	68,95
-- Instruction scalability	100,00	100,01	100,02	100,01	100,01	100,00	99,98	99,98
-- Frequency scalability	100,00	98,22	93,34	90,95	86,26	73,69	80,42	73,35
-- Average IPC	1,911	1,891	1,845	1,775	1,664	1,483	1,417	1,318
-- Average frequency (GHz)	1,683	1,654	1,571	1,531	1,452	1,241	1,354	1,235
-- Runtime (s)	2.153,92	1.311,03	963,85	752,62	695,22	717,53	699,34	701,88
-- Speedup	1,000	1,643	2,235	2,862	3,098	3,002	3,080	3,069

Figure 31: Core scaling metrics with LRFS dataset. Own creation.

If trying to use an optimal configuration for this runs from an AI practitioneer perspective, they could decide using from 2 to 6 cores to maintain the hardware usage efficiency. Although, if the goal is to train the model in the shortest time possible without taking into account the efficient use of resources, they could use 16 cores.

In Figure 32 we can see a comparison between 4 and 24 cores executions, in this order, and both views represent the same time scale. White and red colours represent forward and backward phases respectively, and the batch loading occurs during the black zones. The batch loading phase can be executed faster when using 4 cores, but the convolutional layers are computed faster using 24 cores. Now we know that the speedup gained with the scaling comes from the time difference in forward and backward phases.

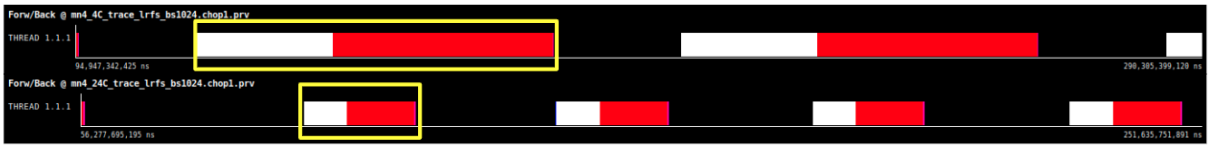

Figure 32: 4 and 24 cores phases duration comparison. Own creation.

Previous analysis have shown how the amount of useful threads varies during the different phases of a training step. In Figure 33 we can see the IPC comparison between mentioned executions representing the same semantic and time scale. With this view we can see different

void zones where the 24 cores execution is losing parallel efficiency that are not present in the 4 cores view. Also, batch loading phase seems to require more or less the same time to finish in both executions, getting a huge performance increase when executing forward and backward phases.

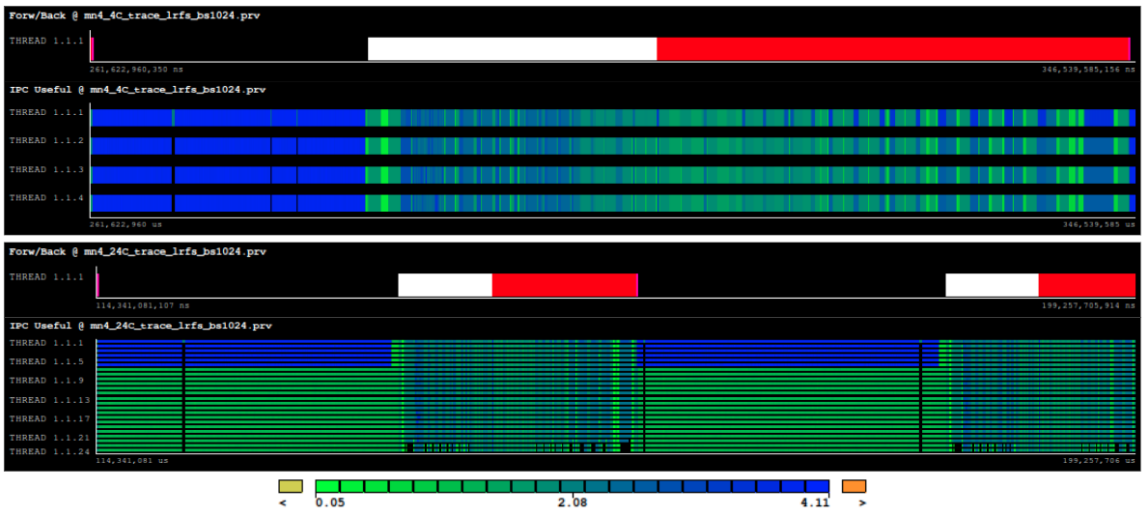

Figure 33: 4 and 24 cores full step IPC comparison. Own creation.

Although the 4 cores execution is able to achieve 77% parallel efficiency, this lacking 23% must be somewhere. Further inspections lead us to show Figure 34, where we can see the behaviour of each execution when loading a single image. Both traces represent the same time and semantic scale. The 4 cores run is able to finish the load way faster, showing less sparsity in the thread burst. This sparsity seems to be related to the thread work assignation. The 4 cores execution shows how each thread starts subsequently, finishing in the same order, but the 24 cores one seems to start its threads without a pattern. This behaviour is repeated along all batched images and all batch loading phases.

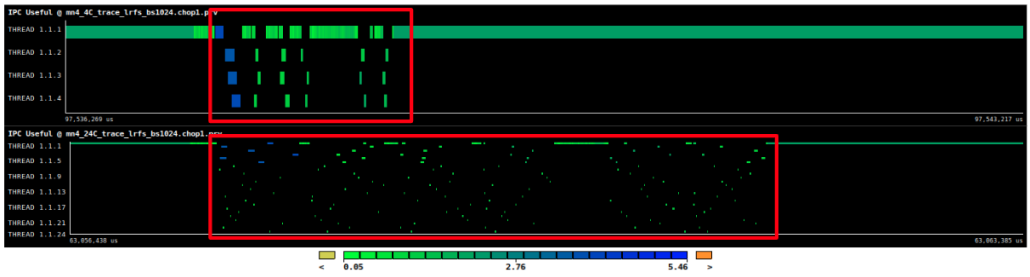

Figure 34: 4 and 24 cores image loading comparison . Own creation.

When taking a look into MRVS executions metrics, summarised in Figure 35, we can see that global efficiency is not decreasing in the same way than LRFS. In this case, the critical point is found between 16 and 24 cores, with a global efficiency drop around 20%, caused by the IPC scalability drop and a slight 4% decrease in parallel efficiency.

Parallel efficiency gets even lower when reaching 36 cores, but this behaviour is expected, as this core increase involves using the second socket in our system. This may force some threads to look for data in the main memory of the other socket, generating imbalances due to data accesses. This effect is also found in Figure 31, when LRFS executions run with 36 and 48 cores.

	MRVS dataset							
Core count	**2**	**4**	**8**	**12**	**16**	**24**	**36**	**48**
Global efficiency	98,99	96,85	90,96	81,80	74,84	56,18	51,67	43,02
-- Parallel efficiency	98,99	98,33	96,74	93,69	94,15	90,28	78,73	73,64
-- Load balance	99,11	98,59	97,61	94,91	95,89	93,09	83,81	80,84
-- Communication efficiency	99,88	99,74	99,12	98,72	98,19	96,98	93,94	91,09
-- Computation scalability	100,00	98,49	94,02	87,30	79,49	62,23	65,62	58,42
-- IPC scalability	100,00	97,95	93,40	87,23	79,70	66,61	69,82	63,91
-- Instruction scalability	100,00	100,01	100,02	99,37	99,37	99,11	98,96	98,97
-- Frequency scalability	100,00	100,54	100,64	100,71	100,36	94,27	94,97	92,37
-- Average IPC	1,930	1,890	1,802	1,683	1,538	1,285	1,347	1,233
-- Average frequency (GHz)	1,835	1,845	1,847	1,848	1,842	1,730	1,743	1,695
-- Runtime (s)	2.058,84	1.052,17	560,16	415,27	340,41	302,29	219,15	197,39
-- Speedup	1,000	1,957	3,675	4,958	6,048	6,811	9,395	10,431

Figure 35: Core scaling metrics with MRVS dataset. Own creation.

IPC scalability metric reduction can be explained by looking at the batch loading phase. Figure 36 shows a trace of the IPC when using 16 and 36 cores in this order, focusing on the batch loading of some images. As both traces represent the same time and semantic scales, we can notice how some threads are able to achieve a higher IPC in the 16 cores run, marked with the orange colour. Although in other phases you can also find subtile variances, only the batch loading one shows this behaviour repeatedly.

Also, the effect of different image shapes can be spotted if looking into the second trace. The second and third images that are being loaded can't use all available threads. Although parallel efficiency decreases in a slower pace in MRVS than LRFS, now the input arrangement will have direct consequences on the parallel behaviour.

With a time perspective of these metrics, we can say that using more than 16 cores won't take full profit of our hardware, but execution time speedups are better than those shown during the LRFS execution when increasing core count. In this case and using this specific dataset, executing with a full node can be worth given a total speedup of around 10x.

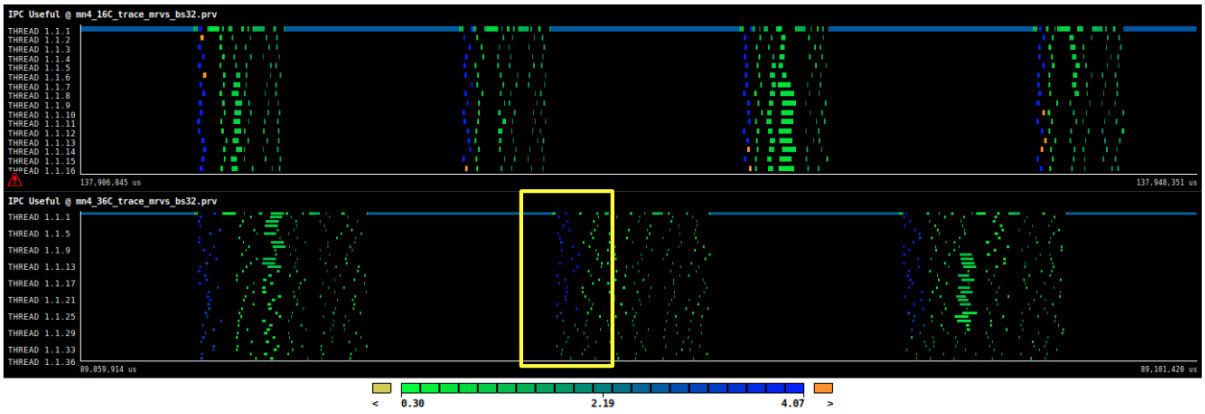

Figure 36: 16 and 24 cores batch loading with MRVS dataset. Own creation.

HRVS dataset efficiency metrics are shown in Figure 37. This image shows a degradation of around 10% of its global efficiency when we increase the core count above 4 cores. IPC scalability is quite similar than the one found in MRVS executions but timewise we are getting worse benefits when using 48 cores, up to around 7x of speedup.

Although we have found similar IPC variances, HRVS dataset doesn't show the IPC spikes found during the batch loading phase of MRVS executions. Figure 38, contains a whole step comparison between the 8 and 24 cores executions, with the same time and semantic scales.

	HRVS dataset							
Core count	**2**	**4**	**8**	**12**	**16**	**24**	**36**	**48**
Global efficiency	97,63	91,44	79,95	67,9	59,85	43,7	35,75	29,57
-- Parallel efficiency	97,63	92,30	84	76,48	72,61	67,32	51,79	45,91
-- Load balance	98,55	93,94	87,03	80,38	77,31	73,07	58,58	54,56
-- Communication efficiency	99,07	98,25	96,52	95,15	93,91	92,13	88,41	84,15
-- Computation scalability	100,00	99,07	95,17	88,78	82,43	64,92	69,03	64,42
-- IPC scalability	100,00	98,46	94,29	87,74	81,6	67,35	71,65	68,58
-- Instruction scalability	100,00	99,92	99,89	99,81	99,81	99,82	101,69	99,83
-- Frequency scalability	100,00	100,70	101,05	101,38	101,21	96,56	94,74	94,1
-- Average IPC	1,831	1,803	1,727	1,607	1,494	1,233	1,312	1,256
-- Average frequency (GHz)	1,841	1,854	1,860	1,866	1,863	1,778	1,744	1,732
-- Runtime (s)	814,15	434,63	248,56	195,12	166,02	151,56	123,51	111,98
-- Speedup	1,000	1,873	3,275	4,173	4,904	5,372	6,592	7,270

Figure 37: Core scaling metrics with HRVS dataset. Own creation.

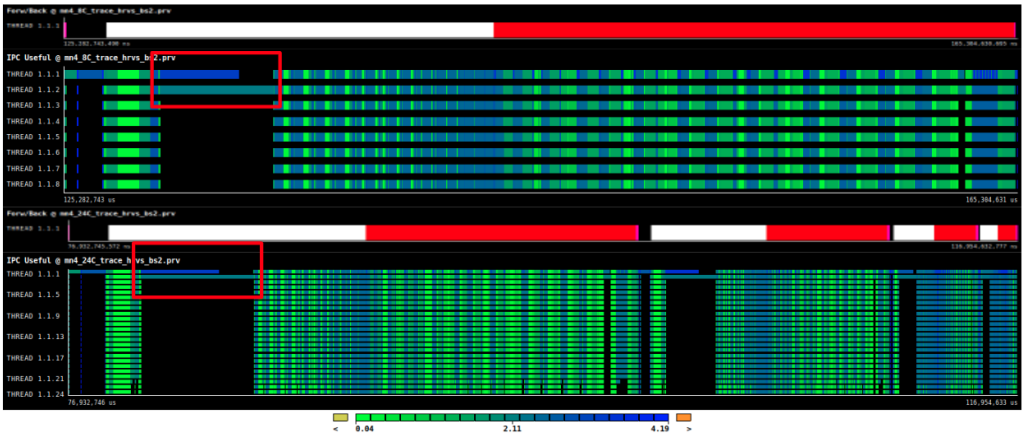

Figure 38: 8 and 24 cores step comparison with HRVS dataset. Own creation.

With these tests we can see how increasing the amount of cores won't improve execution time of the batch loading phase. Also, forward phase reduces its execution time but not proportionally to the amount of cores assigned. Comparing forward and backward phases, we observe that backward phase seems to scale better.

Parallel efficiency decreases for two reasons in this case. The first one, which is shared with other datasets, consists on the dimensions reduction during the convolutional layers, when we assign more cores, we are increasing the size of the regions where not all of them can be used, reducing the overall parallel efficiency of the execution.

The second reason is related to the marked region of the IPC views in the figure. This particular layer is called maxpool, and we have shown how its parallelism relies on the batch size value, not the image size.

When trying to understand the IPC decreases, we have seen that regions with low IPC values tend to elongate when increasing the core count. If the portion of time of this regions increases, the average IPC value for the execution will decrease, as shown in the metrics.

Given the huge amount of time required to train a model with this dataset, using the maximum possible resources is a good option. We have seen that batch loading phase may suffer a performance penalty when increasing core count, but convolutional layers are executed way faster, enough to overcome the penalty.

7 Hyperparameters study

7.1 Batch size

In these tests we will set the amount of cores to 24 for all executions, varying the batch size for each dataset. The idea is to compute the same amount of images with all datasets by varying the amount of steps needed to finish the execution. Using different batch size values may directly impact on the amount of system memory used during the execution, but also could modify the overall parallel efficiency.

	LRFS dataset										
Batch Size value	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Global efficiency	19,40	36,60	65,06	105,08	157,1	214,94	283,96	323,84	338,73	257,99	309,22
-- Parallel efficiency	19,40	19,93	20,31	21,13	22,32	27,88	35,46	40,16	43,73	34,29	40,53
-- Load balance	35,58	34,34	32,91	31,5	30,51	35,54	43,75	47,96	50,54	37,74	45,05
-- Communication efficiency	54,52	58,02	61,73	67,09	73,13	78,45	81,05	83,75	86,54	90,86	89,96
-- Computation scalability	100,00	183,70	320,32	497,17	703,95	770,93	800,81	806,36	774,51	752,46	762,97
-- IPC scalability	100,00	95,40	89,14	79,92	68,37	56,09	46,5	40,02	36,31	34,34	33,03
-- Instruction scalability	100,00	194,26	369,33	671,29	1135,44	1735,59	2364,32	2902,11	3262,56	3478,38	3597,94
-- Frequency scalability	100,00	99,13	97,30	92,67	90,68	79,19	72,83	69,42	65,38	62,99	64,2
-- Average IPC	4,769	4,550	4,251	3,811	3,261	2,675	2,218	1,909	1,732	1,638	1,575
-- Average frequency (GHz)	2,023	2,005	1,968	1,875	1,834	1,602	1,473	1,404	1,322	1,274	1,299
-- Runtime (s)	1.490,52	789,96	444,42	275,18	184,06	134,52	101,83	89,29	85,36	112,08	93,51
-- Speedup	1	1,887	3,354	5,417	8,098	11,080	14,638	16,694	17,461	13,299	15,940

Figure 39: Batch size scaling with LRFS dataset. Own creation.

Figure 39 shows efficiency metrics regarding the executions of the LRFS dataset, starting with just 1 image per batch. We can clearly see a sustained global efficiency increase until reaching a value of 256. Although parallel efficiency seems to increase also until this batch size value, we gain around 15% at maximum, which is a good increase but not good enough to explain the global improvement.

When looking at the computation scalability we see the same tendence, derived from the instruction scalability values, which continue improving while increasing the batch size. Although IPC tends to decrease substantially, this decrease should come from the selection operations, which have to traverse the batched images to slice all tensors, generating more waiting times when acessing different portions of the batch.

The reduction of instructions can be explained by the amount of needed steps to finish the execution. While increasing batch size we also reduce the amount of training steps to compute. With that in mind, we can see that computing less steps leads to 'pay' fewer times the overhead created by the whole training system, including opening and closing parallel regions, forward and backward propagation phases and optimizing the network.

In Figure 40 we show two traces comparing the cycles per microsecond achieved when executing all phases in a training step. The first row belongs to the batch size 4 execution and the second to the batch size 1024, both using the same semantic scale. In this traces we can see how using a higher batch size limits the frequency when passing through convolutional layers. In fact, these zones with less cycles belong to 'select' operations, generating waiting times when accessing to memory in order to slice the output dimensions.

By increasing batch size, our executions are able to finish with less execution time, getting a 15x time improvement when using batch size 1024. Although the speedup is noticeable from the beginning, the sweet spot for this matter seems to be at batch size 256, the same value where our load balance gets the best value.

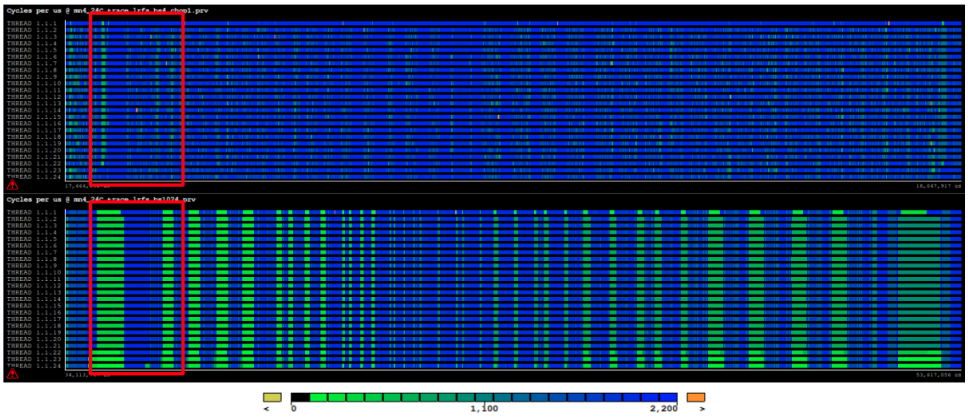

Figure 40: LRFS cycles comparison with different batch sizes. Own creation.

In Figures 41 and 42 we can see the metrics for the MRVS and HRVS executions.

	MRVS dataset					
Batch Size value	**1**	**2**	**4**	**8**	**16**	**32**
Global efficiency	22,21	37,06	50,30	52,94	47,06	33,19
-- Parallel efficiency	22,21	27,90	40,97	58,98	74,49	81,01
-- Load balance	35,61	41,37	54,49	69,83	82,12	88,66
-- Communication efficiency	62,37	67,45	75,20	84,46	90,71	91,38
-- Computation scalability	100,00	132,81	122,76	89,76	63,17	40,97
-- IPC scalability	100,00	81,33	59,56	43,64	36,98	32,46
-- Instruction scalability	100,00	158,70	213,03	225,03	196,37	146,84
-- Frequency scalability	100,00	102,91	96,75	91,4	86,99	85,94
-- Average IPC	4,073	3,312	2,426	1,778	1,506	1,322
-- Average frequency (GHz)	1,921	1,976	1,858	1,755	1,671	1,651
-- Runtime (s)	1.739,04	1.042,30	767,97	729,60	820,90	1.163,85
-- Speedup	1	1,430	1,941	2,043	1,816	1,281

Figure 41: Batch size scaling with MRVS dataset. Own creation.

When using MRVS dataset we can see a sustained increase in parallel efficiency and load balance along the batch size scalation, but this improvements are overshadowed by the IPC drop. The sweet spot for this specific dataset seems to be using a batch size of 8 images. Above this value we see increases in the amount of executed instructions, making even worse the IPC.

Load balance and parallel efficiency increases can be caused by the step number reduction, because the impact of the single threaded zones produced at each training step will be lesser. We also know that work distribution occurs at images level, and as these images will be bigger than any LRFS image, parallelism will be better from the start, as seen with batch size 1 global efficiency.

Due to memory constraints, HRVS executions can only scale up to two images per batch. In this case we find the same tendency regarding load balance and parallel efficiency, as well as the IPC and instructions drop.

Our metrics reveal an improvement of 20% in global efficiency and a speedup of 1,4x with a slight decrease in computation scalability. HRVS is the only dataset that shows good efficiency values when using the maximum batch size possible. Even when MRVS was able to achieve a good parallel performance too, the IPC scalability was lower.

Parallel efficiency with HRVS dataset improves drastically due to the well known now max-pool layer. All executed steps during these executions use 1 and 2 threads respectively during this layer, as seen during previous analysis of the forward phase.

Figure 43 shows a comparison at the same semantic scale of a full training step between

	HRVS dataset	
Batch Size value	**1**	**2**
Global efficiency	48,79	71,50
-- Parallel efficiency	48,79	78,48
-- Load balance	53,59	84,28
-- Communication efficiency	91,04	93,12
-- Computation scalability	100,00	91,11
-- IPC scalability	100,00	76,94
-- Instruction scalability	100,00	110,55
-- Frequency scalability	100,00	107,12
-- Average IPC	1,563	1,203
-- Average frequency (GHz)	1,668	1,786
-- Runtime (s)	10.439,58	7.123,39
-- Speedup	1	1,466

Figure 42: Batch size scaling with HRVS dataset. Own creation.

HRVS executions with batch sizes 1 and 2 respectively. This traces reveal the importance of increasing the batch size due to its relation with parallelism. Although the effects during backward propagation phase are not so noticeable (marked with the yellow rectangle), this cannot be overlooked when executing the forward phase (marked with the red rectangle). Increasing the batch size value not only increases the parallel efficiency of this layer, it also reduces the total amount of steps to execute.

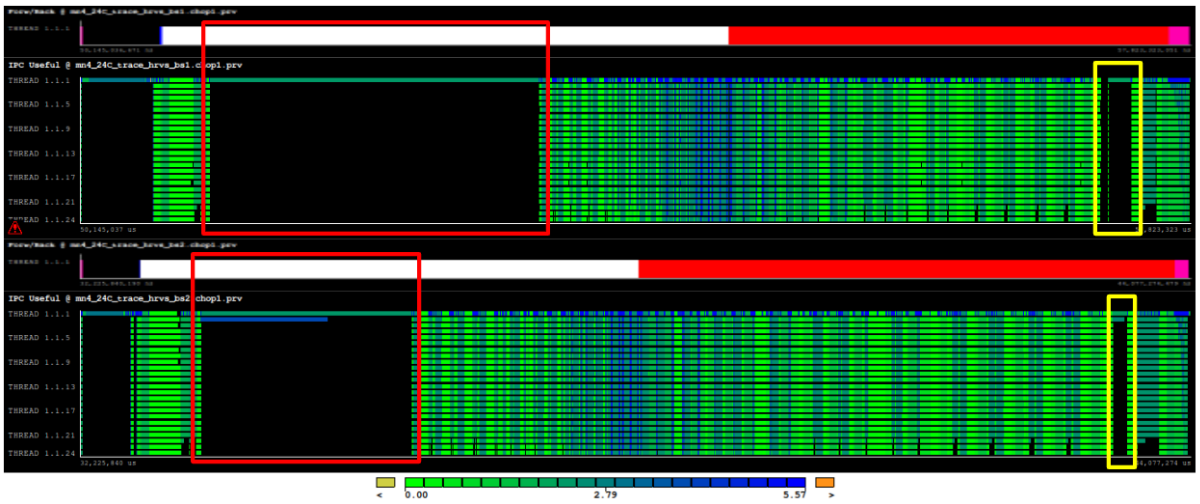

Figure 43: HRVS batch size traces comparison. Up: batch size=1, Down: batch size=2. Own creation.

Although we have noticed huge benefits when increasing batch size when using HRVS dataset, this is not the case for the others. LRFS executions show a huge increase in global efficiency due to instruction scalability gains, but load balance achieved a peak value of 40% efficiency when using a batch of 256 images.

Figures 44 and 45 show a comparison of a full training step when using HRVS and LRFS executions in this order. In this case we will compare those executions that showed better results in the measured metrics, using a batch size of 256 for LRFS and 2 for HRVS.

When watching Figure 44, the batch loading phase (black zone) of the LRFS run uses around 40% of its step time, while HRVS step spends more than 90% of its time in forward and backward propagations (white and red zones), which consist on convolutional layers.

Figure 44: HRVS(up) & LRFS (down) training phase comparison. Own creation.

Referring to Figure 45 where the IPC of each dataset step is represented in the same semantic scale, we spot clear differences in their behaviours. On each view we have highlighted with yellow, red and pink rectangles the zones where batch loading, forward and backward phases are being executed. It is clear that maxpool layer is working against HRVS in this comparison, as LRFS is able to use more parallelism there due to its batch size value, but the load imbalance during the batch loading phase of LRFS execution shown in previous sections has a stronger effect over parallel efficiency that overcomes the convolutional layers gains.

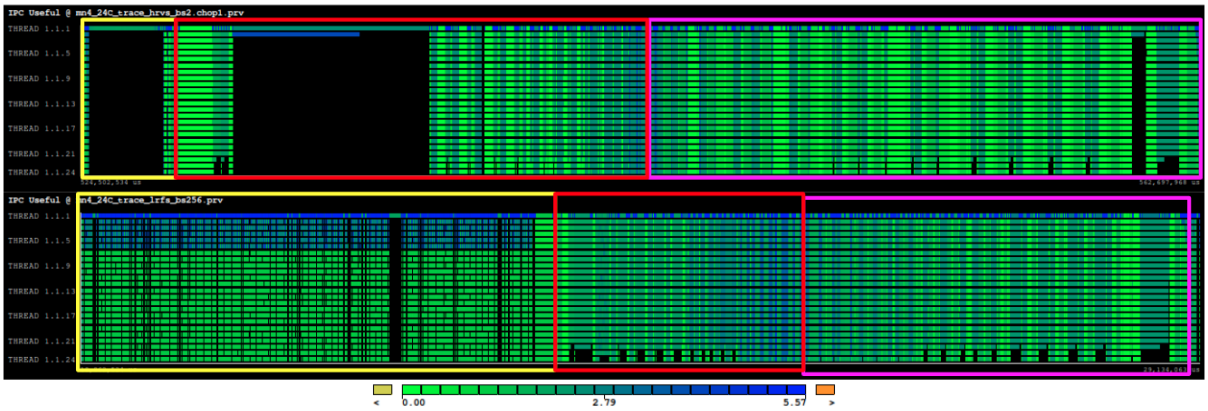

Figure 45: HRVS (up) & LRFS (down) training phase IPC comparison. Own creation.

So far we have seen different flaws in both HRVS and LRFS executions that impact their efficiency, all related to parallelism behaviour in the different execution phases. Core and batch size scalability tests have shown us that we can optimally train the model with LRFS data using a machine with less memory and processing capacities. On the other hand, when using HRVS data we have seen that we can take advantage of more cores, but we do not have enough main memory to continue scaling the batch size. Despite this, we see that HRVS runs benefit greatly from increasing the number of images processed at each step in terms of taking advantage of the hardware.

Even if HRVS processing can take profit of the machine, execution time will prevent AI developers from using it if no other advantages can be obtained. Therefore, our next step is to analyze how our model predictions are affected by the input dataset change.

7.2 Grain size

When training a deep learning model like the one we are using in this project, parallel execution is of huge importance. During the first tests we found out that PyTorch's parallelism policy relies on a global variable called "GRAIN SIZE", located in the ATen module of the Torch library, which in fact is a constant expression. This avoids the possibility of changing the value dynamically, so each test value requires a recompilation of this specific package. This value determines the minimum amount of work needed to spawn a new thread if the execution is not using all of them yet.

PyTorch's developers have set the value of the work distribution variable to 32 KBytes, arguing that this value has been heuristically chosen to offer the lowest execution times in a general scenario, so we tested different values to verify if they where correct about this matter.

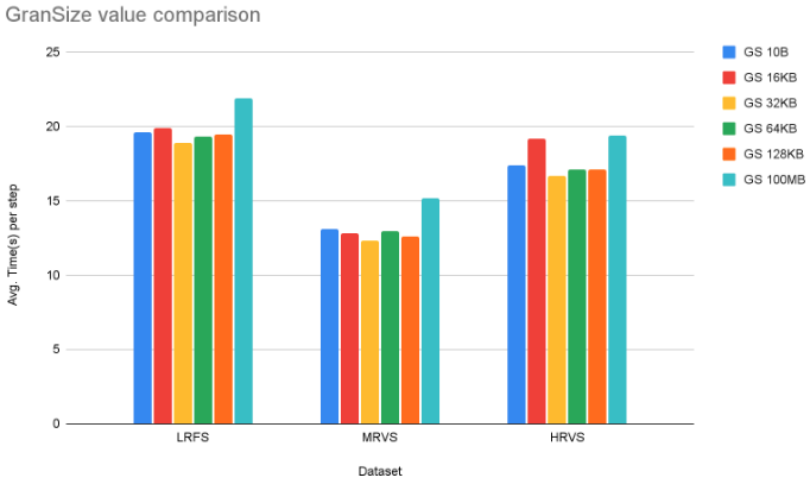

Figure 46: Grain size value comparison in all datasets. Own creation.

Figure 46 shows the average execution time per training step on the Y axis and the executed dataset in the X one. We tested values near the default and also two extreme values, being 10 Bytes and 100 MBytes, and found out that developers where right. In all executions, the 32 KBytes value is able to finish with slightly lower execution times than other values, regardless the used dataset.

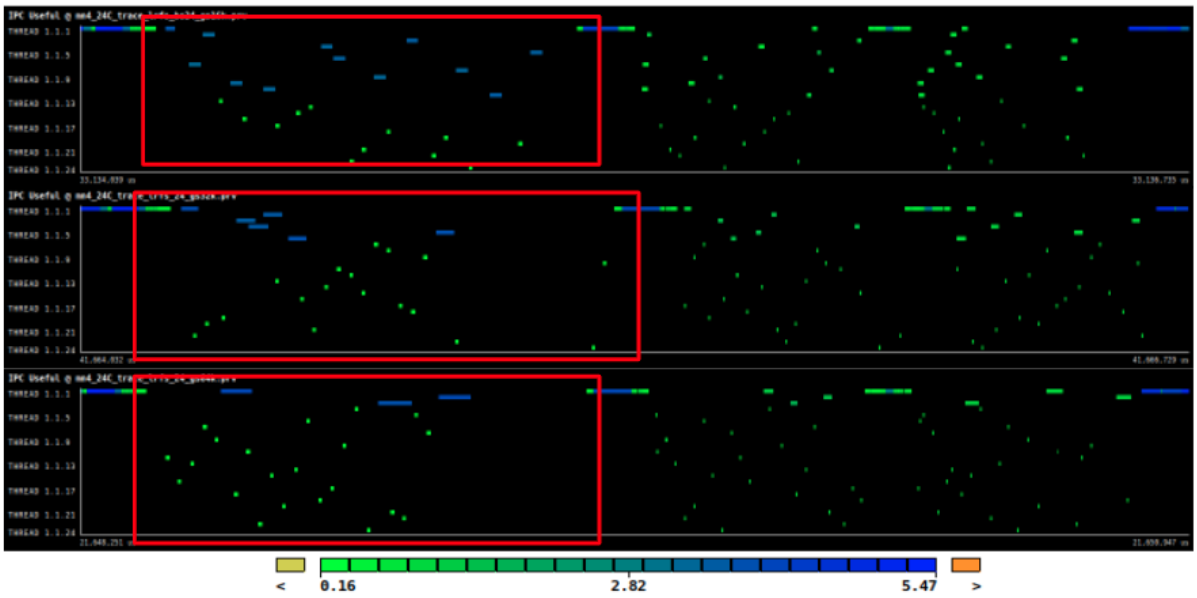

Figure 47: GrainSize effects on Batch loading phase with LRFS. Own creation.

In Figure 47 we can see the effects of varying the grainsize value by looking at the IPC traces. As explained during batch loading section, LRFS dataset can only take profit of 6 threads when using a 32 KBytes value. Now we can see on the first row an execution with a grainsize value of 16 KBytes that, as expected, is using 12 threads this time. The second row represent the execution with the default value and the third one shows the execution using a value of 64 KBytes, only using 3 threads.

In order to evaluate how the grainsize value is affecting our executions, we have created Figures 48, 49 and 50 showing the POP metrics applied to our set of testing executions with each dataset.

	LRFS dataset					
GrainSize value	**10 B**	**16 KB**	**32 KB**	**64 KB**	**128 KB**	**100 MB**
Global efficiency	37,99	34,32	37,62	38,35	37,43	38,44
-- Parallel efficiency	37,99	35,47	37,42	38,10	37,63	36,49
-- Load balance	42,59	38,97	41,26	41,98	41,20	37,09
-- Communication efficiency	89,19	91,00	90,68	90,75	91,33	98,38
-- Computation scalability	100,00	96,77	100,55	100,66	99,48	105,36
-- IPC scalability	100,00	99,18	99,98	100,06	99,80	106,73
-- Instruction scalability	100,00	100,05	100,04	100,06	100,02	100,07
-- Frequency scalability	100,00	97,53	100,52	100,54	99,65	98,64
-- Average IPC	1,563	1,550	1,563	1,564	1,560	1,668
-- Average frequency (GHz)	1,284	1,252	1,290	1,291	1,279	1,266
-- Runtime (s)	500,719	554,199	505,611	496,027	508,168	494,820

Figure 48: GrainSize metrics using LRFS dataset. Own creation.

	MRVS dataset					
GrainSize value	**10 B**	**16 KB**	**32 KB**	**64 KB**	**128 KB**	**100 MB**
Global efficiency	79,23	78,97	80,06	79,86	81,41	68,39
-- Parallel efficiency	79,23	79,74	80,59	80,13	80,91	64,30
-- Load balance	87,04	85,00	86,90	86,68	85,97	67,35
-- Communication efficiency	91,02	93,81	92,74	92,44	94,11	95,47
-- Computation scalability	100,00	99,04	99,34	99,66	100,62	106,36
-- IPC scalability	100,00	98,21	99,83	100,12	99,43	105,85
-- Instruction scalability	100,00	100,03	100,03	100,03	100,04	100,04
-- Frequency scalability	100,00	100,81	99,47	99,51	101,16	100,45
-- Average IPC	1,413	1,388	1,411	1,415	1,405	1,496
-- Average frequency (GHz)	1,693	1,707	1,684	1,685	1,712	1,701
-- Runtime (s)	145,705	146,190	144,200	144,556	141,801	168,808

Figure 49: GrainSize metrics using MRVS dataset. Own creation.

	HRVS dataset					
GrainSize value	**10 B**	**16 KB**	**32 KB**	**64 KB**	**128 KB**	**100 MB**
Global efficiency	75,30	67,41	76,54	76,22	76,32	63,53
-- Parallel efficiency	75,30	75,01	76,51	76,08	76,27	59,75
-- Load balance	80,86	79,61	81,23	80,71	80,83	62,40
-- Communication efficiency	93,12	94,22	94,19	94,27	94,36	95,76
-- Computation scalability	100,00	89,88	100,04	100,19	100,07	106,32
-- IPC scalability	100,00	99,00	99,97	100,04	100,05	106,05
-- Instruction scalability	100,00	90,91	100,03	100,03	100,03	100,03
-- Frequency scalability	100,00	99,86	100,04	100,11	99,99	100,23
-- Average IPC	1,271	1,259	1,271	1,272	1,272	1,348
-- Average frequency (GHz)	1,794	1,791	1,795	1,796	1,794	1,798
-- Runtime (s)	190,587	212,882	187,500	188,272	188,023	225,887

Figure 50: GrainSize metrics using HRVS dataset. Own creation.

We can appreciate certain variations in the load balance metric on all datasets, reaching the top values when using grainsize values between 32 and 128 KBytes. None of the datasets shows a reasonable improvement in any metric nor in execution time, except for the slight IPC increase showed in the 100 MB columns. This can be explained by knowing that using a higher grainsize value will spawn less threads, allowing the used ones to complete tasks with less data sharing conflicts.

With these tests we can ensure that, for our purposes, PyTorch developers have hit the bulls-eye with their default grainsize value. Nevertheless, we are using a specific kind of DNN

(CNN) and we can not ensure that this default value would throw the best performance with any network.

7.3 Accuracy tests

The goal of training a model with meaningful data is to enhance its capacity to detect certain features from data that wasn't used to train it. When watching the training process from another perspective, each training step has to face data never processed before, therefore, we can measure the accuracy increases at each training step and calculate an accuracy average for the training process at the end of an epoch.

Predictions are computed during the forward phase and are used to calibrate gradients that will affect neuron activations in the next step. When training a specific model, an AI developer may test a model by executing the first epoch and check the accuracy values as a guide to evaluate further development decisions. We know that training models with HRVS data consumes a huge amount of time and energy when compared with LRFS training but, if some meaningful gains are achieved in the accuracy perspective, using this data may be worth the effort in some specific study fields.

Until this point we have seen how batch size variations imply different effects on the execution's behaviour and hardware utilization, but now we want to check if the effects spread to accuracy values by measuring accuracy in the first training epoch with different batch size values.

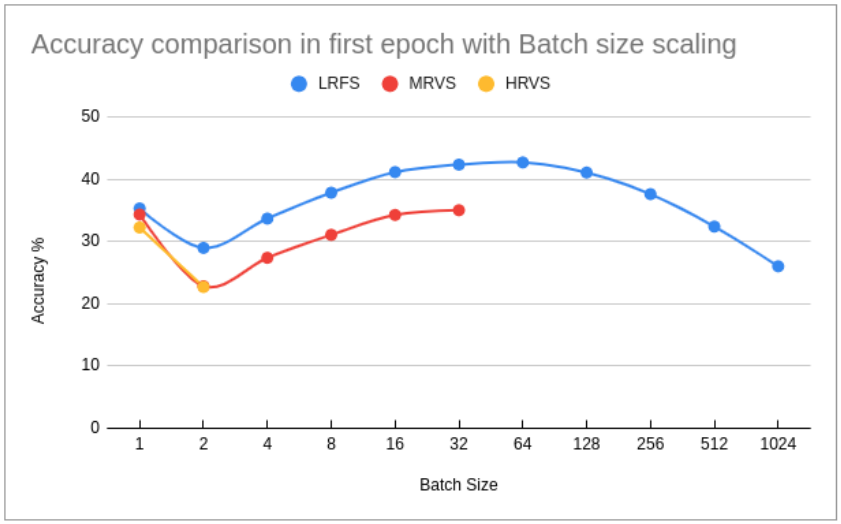

Figure 51: Accuracy comparison plot. Own creation.

In Figure 51 we represent the accuracy of the first epoch in percentage through the Y axis, and batch size values over the X axis. From the start, with batch size 1, we clearly see that LRFS dataset achieves better accuracy. All three datasets seem to follow the same curved tendence with batch size variations between 2 and 32. Unfortunately, main memory restrictions make HRVS batch size variations insuficient to confirm this tendence.

We can also check Figure 52 where accuracy values are represented in percentage. When using batch size 1, LRFS and MRVS accuracy results only differ 1% but, when more than one image is used, results distance starts to grow at each batch size increase. This effect can be explained with the huge amount of padding pixels contained in MRVS batches shown in Figure 15, as those are computed in the same way that informative ones and could be altering gradients during the backpropagation phases. When checking batch size 2 executions we find that MRVS

	Batch Size										
Dataset	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
LRFS	35,34	28,98	33,71	37,87	41,19	42,39	42,75	41,12	37,64	32,42	26,02
MRVS	34,36	22,81	27,39	31,09	34,29	35,05	-	-	-	-	-
HRVS	32,27	22,70	-	-	-	-	-	-	-	-	-

Figure 52: Accuracy comparison table. Own creation.

and HRVS hit almost the same value.

Given this data we can state that batch size value is meaningful when training the first epoch. LRFS dataset gets around 42% accuracy when using batch sizes between 16 and 128. If we refer to Figure 39 we can mix our results and find that this specific dataset will achieve its best hardware and accuracy performances when using from 32 to 128 images per batch.

Another hyperparameter comes into play when talking about accuracy: learning rate. Now we know that batch size variances alter accuracy results, but an AI developer will be modifying learning rate values in order to find the best value, so we should too.

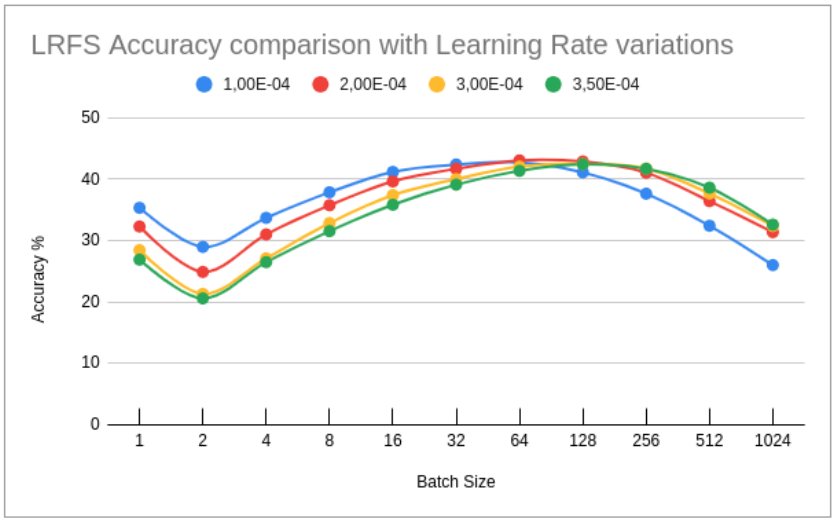

Figure 53: Accuracy plot with Learning Rate variances. Own creation.

Figure 53 represents the accuracy values with different LRFS executions but slightly altering learning rate values from 0.0001 to 0.00035. Contrary to our first thoughts, we haven't surpassed the maximum accuracy found before. Instead, we can see how accuracy values are shifted to the right, getting better values with higher batch sizes. In fact, using a batch size of 256 is a better option from hardware and accuracy perspectives if we set the learning rate to 0.00035.

With these tests we have seen not only how the amount of images per training step may affect its accuracy performance, but also spottted a possible relation between the learning rate and the batch size hyperparameters.

The main goal of these tests was to find some advantage when using high resolution images but, given our current results, we can say that using HRVS datasets won't produce better accuracy results. Although this statement might be true in our case, this cannot be ported to other scenarios. If we are using a different dataset or a slightly different CNN (for example with extra or less layers) this study should be repeated.

8 Conclusions

8.1 Project conclusions

Motivated by the need of processing high resolution images in certain study fields, we have highlighted some relevant performance differences and needs. By tracing and analysing each phase of a CNN model's training process we have seen how paralelism is used in different ways. In fact, our tests with batch sizes unveiled how the image loading phase opens parallel regions for each image, creating possible overheads when using high batch size values.

We have also seen how forward and backward phases have different impacts on resources usage by comparing their metrics, inspite of traversing the same layers during their execution. Traces have shown how backward phase uses a different approach with convolution operations in order to update gradients and, during all the tests, the optimization phase shows the same behaviour independently of the dataset used and the amount of images processed per step. As we know that optimization algorithms traverse all layers to update weights, we can confirm that this phase will only depend on the neural network configuration and size, not on the data used.

Scalability tests regarding the amount of CPU cores have shown how the best value depends tightly on the phase we focus on. When creating a batch, image size defines how many threads can be used, therefore using bigger images results in better load balance during this phase.

The usage of padding pixels, needed to create a batch of variable shaped images, has a significant impact in computational performance. While creating a batch with non squared images, we are building a bigger structure containing information that doesn't belong to the meaningful data. This implies not only higher memory requirements and execution times, but also affects the overall prediction accuracy of the model, at least during the first epoch. Given the effect of padding pixels, it is worth trying to apply different batching techniques that, even if they require more processing during batch creation, may allow to perform better in the overall training.

Even though low resolution images have shown better accuracy results, we have confirmed that this kind of data isn't able to profit all our hardware capabilities in the same way that high resolution images do, when using this specific framework in CPU devices. Although this may push us to keep using low resolution images, we have to keep in mind that most of the existing images have really high pixel counts, forcing AI practitioneers to process their datasets to reduce resolution. This process could imply information loss when the resolution difference is extremely high, creating non meaningful data.

8.2 Personal Conclusions

This project has introduced me to research on the high performance computing field and I have developed my performance analysis skills. Now I am more fluent when adding code changes related to instrumentation with external tools.

Before this project, deep learning models acted like a blackbox to me. Being able to understand how the information flows under the hood has increased my interest in this particular field. Knowing that this study is tightly related to a specific model designed for a specific task, I would like to explore other model designs to see if our conclusions are applicable in any way. Although this project is focused on performance in HPC clusters, I'm interested in exploring how to adapt this kind of programs to be used in embedded systems.

Also, developing the analysis has improved my working methodology related to data handling, as all traces generated by the executions were huge in size due to the time needed by the training process, creating some troubles when working on trace visualization.

9 References

[1] "IBM", "Ibm learn platform," https://www.ibm.com/cloud/learn/neural-networks, [Online].

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] "POP-COE", "Performance optimization and productivity coe in hpc," https://pop-coe.eu/, [Online].

[4] K. J. Geras, S. Wolfson, Y. Shen *et al.*, "High-resolution breast cancer screening with multi-view deep convolutional neural networks," *CoRR*, vol. abs/1703.07047, 2017. [Online]. Available: https://arxiv.org/abs/1703.07047

[5] W. Lotter, G. Sorensen, and D. Cox, "A multi-scale CNN and curriculum learning strategy for mammogram classification," in *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support, 17 Sept. 2017, Québec City, QC, Canada*. Springer, 2017, pp. 169–177. [Online]. Available: https://doi.org/10.1007/978-3-319-67558-9_20

[6] M. Treml, J. Arjona-Medina, T. Unterthiner *et al.*, "Speeding up semantic segmentation for autonomous driving," in *MLITS, NIPS Workshop*, vol. 2, 2016, p. 7.

[7] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 21-26 July 2017, Honolulu, HI, USA*. IEEE, 2017, pp. 1907–1915. [Online]. Available: https://doi.org/10.1109/cvpr.2017.691

[8] M. Reichstein, G. Camps-Valls, B. Stevens *et al.*, "Deep learning and process understanding for data-driven earth system science," *Nature*, vol. 566, no. 7743, pp. 195–204, 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-0912-1

[9] F. Parés, A. Arias-Duart, D. Garcia-Gasulla, G. Campo-Francés, N. Viladrich, E. Ayguadé, and J. Labarta, "The mame dataset: on the relevance of high resolution and variable shape image properties," *Applied Intelligence*, pp. 1–22, 2022.

[10] F. Parés, A. Arias-Duart, D. Garcia-Gasulla *et al.*, "A Closer Look at Art Mediums: The MAMe Image Classification Dataset," *CoRR*, vol. abs/2007.13693, 2020. [Online]. Available: https://arxiv.org/abs/2007.13693

[11] M. . technical information", "Barcelona supercomputing center," https://www.bsc.es/marenostrum/marenostrum/technical-information, accessed: 2022-02-20. [Online].

[12] U. A. labs Pyro", "Uber technologies inc." https://eng.uber.com/pyro, accessed: 2022-02-20. [Online].

[13] O. A. webpage", "Openmp architecture review board," https://tools.bsc.es/downloads, accessed: 2022-02-11. [Online].

[14] P. A. tools", "Barcelona supercomputing center," https://tools.bsc.es/downloads, accessed: 2022-01-14. [Online].

[15] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: https://arxiv.org/abs/1409.0473

[16] J. Dean, G. S. Corrado, R. Monga *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc., 2012.

[17] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *CoRR*, vol. abs/1604.06174, 2016. [Online]. Available: https://arxiv.org/abs/1604.06174

[18] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, *Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training*, 2020, pp. 151–166.

[19] I. N. Sotiropoulos, "Handling variable shaped & high resolution images for multi-class classification problem," Master's thesis, Universitat Politècnica de Catalunya, 2020.

[20] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 852–863.

[21] M. Fowler, J. Highsmith *et al.*, "The agile manifesto," *Software development*, vol. 9, no. 8, pp. 28–35, 2001.

[22] O. L. Editor", "Overleaf online editor," https://www.overleaf.com, [Online].

[23] "Thinkmate", "Thinkmate rack configurator," https://www.thinkmate.com, [Online].

[24] "Syslabs", "Singularity," https://sylabs.io/singularity, [Online].

[25] "Docker", "Docker hub," https://www.docker.com, [Online].

[26] F. P. Pont, P. Megias, D. Garcia-Gasulla, M. Garcia-Gasulla, E. Ayguadé, and J. Labarta, "Size & shape matters: The need of hpc benchmarks of high resolution image training for deep learning," *Supercomputing Frontiers and Innovations*, vol. 8, no. 1, pp. 28–44, 2021.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.