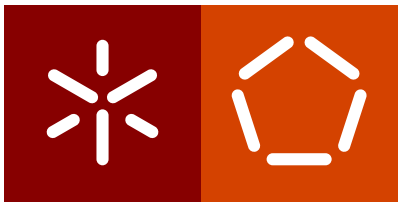
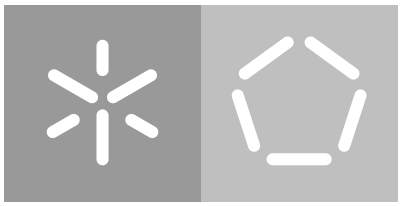


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Cláudia Sofia Mendonça de Sá Correia

PRISMA: A Prefetching Storage Middleware for Accelerating Deep Learning Frameworks

February 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Cláudia Sofia Mendonça de Sá Correia

PRISMA: A Prefetching Storage Middleware for Accelerating Deep Learning Frameworks

Master dissertation
Integrated Master in Informatics Engineering

Dissertation supervised by
João Tiago Medeiros Paulo
António Luís Pinto Ferreira Sousa

February 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

This dissertation is the culmination of five years of hard work, along with new and rewarding experiences. It would not have been possible without the help of a diverse group of people, both in and out of the academic environment, to whom I could not fail to express my gratitude.

First of all, I would like to thank my supervisor, Professor João Paulo, for the opportunity of working on this project, for always guiding me in the right direction, and for all the encouraging words throughout this journey. I would also like to thank my co-supervisor, Ricardo Macedo, for all the advice and support, and for always being available to help with any obstacle that might appear. This dissertation would not be concluded if it were not for the countless meetings and discussions, which were crucial to achieve the best possible results. Furthermore, I would like to thank Professor António Sousa for supporting this project.

Thank you to Cláudia Brito, whose help and knowledge were vital to outline the first steps of this dissertation, and to Marco Dantas for taking the time to contribute to the project. I would also like to thank all my colleagues in HASLab for being welcoming and providing an amazing work environment.

To the National Institute of Advanced Industrial Science and Technology (AIST) team, especially Doctor Jason Haga and Doctor Yusuke Tanimura, I would like to express my gratitude for all the relevant input and for providing the opportunity of conducting the experiments on the AI Bridging Cloud Infrastructure (ABCI).

A special thank you to Armando Santos, the person who has accompanied me from day one, who supported me unconditionally, and who had the patience to listen and help me whenever I needed to. This would certainly not be possible without you.

To my sister, thank you for the never-ending advice and encouragement and for always supporting me in the most difficult moments.

To my family, I would like to express my endless gratitude for the opportunities they have provided me, both academically and personally, and for the constant concern and affection throughout these years.

I would also like to thank my friends for all the memorable times we shared and for making these five years an unforgettable experience.

Last but not least, I would like to thank the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT), for financing this work within project UIDB/50014/2020.

ABSTRACT

Deep Learning (DL) is a widely used technique often applied to many domains, from computer vision to natural language processing. To avoid overfitting, DL applications have to access large amounts of data, which affects the training performance. Although significant hardware advances have already been made, current storage systems cannot keep up with the needs required by DL techniques. Considering this, multiple storage solutions have already been developed to improve the *Input/Output (I/O)* performance of DL training. Nevertheless, they are either specific to certain DL frameworks or present drawbacks, such as loss of accuracy. Most DL frameworks also contain internal I/O optimizations, however they cannot be easily decoupled and applied to other frameworks. Furthermore, most of these optimizations have to be manually configured or comprise greedy provisioning algorithms that waste computational resources.

To address these issues, we propose PRISMA, a novel storage middleware that employs data prefetching and parallel I/O to improve DL training performance. PRISMA provides an autotuning mechanism to automatically select the optimal configuration. This mechanism was designed to achieve a good trade-off between performance and resource usage. PRISMA is framework-agnostic, meaning that it can be applied to any DL framework, and does not impact the accuracy of the training model. In addition to PRISMA, we provide a thorough study and evaluation of the TensorFlow Dataset *Application Programming Interface (API)*, demonstrating that local DL can benefit from I/O optimization.

PRISMA was integrated and evaluated with two popular DL frameworks, namely TensorFlow and PyTorch, proving that it is successful under different I/O workloads. Experimental results demonstrate that PRISMA is the most efficient solution for the majority of the scenarios that were studied, while for the other scenarios exhibits similar performance to built-in optimizations of TensorFlow and PyTorch.

Keywords: Deep Learning, Storage Systems, I/O, TensorFlow, PyTorch, Prefetching, Parallel I/O

RESUMO

Aprendizagem Profunda (AP) é uma área bastante abrangente que é atualmente utilizada em diversos domínios, como é o caso da visão por computador e do processamento de linguagem natural. A aplicação de técnicas de AP implica o acesso a grandes quantidades de dados, o que afeta o desempenho de treino. Embora já tenham sido alcançados avanços significativos em termos de *hardware*, os sistemas de armazenamento atuais não conseguem acompanhar os requisitos de desempenho que os mecanismos de AP impõem. Considerando isto, foram desenvolvidas várias soluções de armazenamento com o objetivo de melhorar o desempenho de *Entrada/Saída (E/S)* do treino de AP. No entanto, as soluções existentes possuem certas desvantagens, nomeadamente perda de precisão do modelo de treino e o facto de serem específicas a determinadas plataformas de AP. A maioria das plataformas de AP também possuem otimizações de E/S, contudo essas otimizações não podem ser facilmente desacopladas e aplicadas a outras plataformas. Para além disto, a maioria destas otimizações tem que ser configurada manualmente ou contém algoritmos de provisionamento gananciosos, que desperdiçam recursos computacionais.

Para resolver os problemas anteriormente mencionados, esta dissertação propõe o PRISMA, um *middleware* de armazenamento que executa pré-busca de dados e paralelismo de E/S, de forma a melhorar o desempenho de treino de AP. O PRISMA providencia um mecanismo de configuração automática para determinar uma combinação de parâmetros ótima. Este mecanismo foi desenvolvido com o objetivo de obter um bom equilíbrio entre desempenho e utilização de recursos. O PRISMA é independente da plataforma de AP e não afeta a precisão do modelo de treino. Além do PRISMA, esta dissertação providencia um estudo e uma avaliação detalhados da *Interface de Programação de Aplicações (API)* Dataset do TensorFlow, provando que AP local pode beneficiar de otimizações de E/S.

O PRISMA foi integrado e avaliado com duas plataformas de AP amplamente utilizadas, o TensorFlow e o PyTorch, demonstrando que este *middleware* tem sucesso sob diferentes cargas de trabalho de E/S. Os resultados experimentais demonstram que o PRISMA é a solução mais eficiente na maioria dos cenários estudados, e possui um desempenho semelhante às otimizações internas do TensorFlow e do PyTorch.

Palavras-chave: Aprendizagem Profunda, Sistemas de Armazenamento, E/S, TensorFlow, PyTorch, Pré-busca, E/S Paralela

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

A

ABCI AI Bridging Cloud Infrastructure.

AI Artificial Intelligence.

AIST National Institute of Advanced Industrial Science and Technology.

ANN Artificial Neural Network.

API Application Programming Interface.

C

CDF Cumulative Distribution Function.

CNN Convolutional Neural Network.

CPU Central Process Unit.

CTL Custom Training Loop.

D

DL Deep Learning.

DNN Deep Neural Network.

DTT Data tracking tool.

E

EOO Entropy-aware Opportunistic Ordering.

G

GPFS General Parallel File System.

GPU Graphical Processing Unit.

H

HDD Hard Disk Drive.

HDF5 Hierarchical Data Format version 5.

HPC High-Performance Computing.

HTTP Hypertext Transfer Protocol.

I

I/O Input/Output.

ILSVRC2012 ImageNet Large Scale Visual Recognition Challenge 2012.

K

KV Key-Value.

L

LMDB Lightning Memory-Mapped Database.

M

ML Machine Learning.

N

NVM Non-Volatile Memory.

NVME Non-Volatile Memory Express.

O

OS Operating System.

P

POSIX Portable Operating System Interface.

R

RAM Random-Access Memory.

RDMA Remote Direct Memory Access.

RELU Rectified Linear Unit.

REST Representational State Transfer.

RNN Recurrent Neural Network.

S

SDS Software-Defined Storage.

SELU Scaled Exponential Linear Unit.

SGD Stochastic Gradient Descent.

SSD Solid-State Drive.

T

TACC Texas Advanced Computing Center.

TANH Hyperbolic Tangent.

TBB Threading Building Blocks.

TPU Tensor Processing Unit.

1

INTRODUCTION

Artificial Intelligence (AI) has been subject of great attention in the past few years [106]. When we hear of AI, we immediately think about chatbots and self-driving cars, and imagine a utopian future where most of our daily tasks are handled by robots. Although there is no universal definition of AI, we can ultimately describe it as the development of computer systems capable of simulating activities that usually require human intelligence [99].

Contrarily to popular belief, AI has been around for over 60 years, since it was first coined in 1956 by the computer scientist John McCarthy, and has experienced both optimistic and disappointing periods along the way [25, 80]. Today this field is expanding substantially and, as a matter of fact, from January 2015 to January 2018, active AI startups increased by 113% [106]. Furthermore, multinational companies such as Google, Microsoft, Facebook, and Amazon are also investing in AI, representing a huge boost for its growth. Apart from rising as a research field, AI has a significant economic impact, with contributions to the global economy estimated to reach up to $15.7 trillion by 2030 [93].

Throughout its history, AI research has been branched into a variety of subfields, according to specific goals (e.g., computer vision, robotics) and tools (e.g., *Artificial Neural Network (ANN)*, probabilistic reasoning) [99]. *Machine Learning (ML)* is one of the AI subfields, which focus on building algorithms that allow computer systems to automatically learn and improve from experience [72]. Unlike human learning, when we discuss about a machine's ability to learn, we are referring to finding a mathematical formula which, when applied to a collection of inputs (training data), produces the desired output [13]. Under this design, there are three main types of learning: supervised, unsupervised, and reinforcement. Supervised learning trains the model with labeled data, while unsupervised learning uses unlabeled datasets. On the other hand, reinforcement learning uses occasional positive and negative feedback to reinforce behaviors [33].

ML was grounded on mathematical concepts, long before the emergence of computers. Since the breakthroughs regarding Bayes' theorem in the 18^{th} century [5], there have been great victories in the ML field. One of the most recent achievements was AlphaGo [107], a computer program that plays the ancient board game Go, combining supervised and reinforcement learning.

From content filtering (e.g., e-mail spam filtering [100], fraud detection [34]) to online recommendations based on user's interests (e.g., shopping suggestions [21], targeted advertising [84]), ML technology is present in our day-to-day life without we even noticing. Most of these applications use a specialized form of ML called *Deep Learning (DL)* [60].

Inspired by the design of the human brain, DL is a technique suitable for automatically recognizing patterns in sound, text, or images, using ANNs and a large amount of data. The term neural network is a reference to neuroscience and indicates a set of layers that are chained together. Each layer is composed of neurons with specific parameters (weights and biases). All networks share a common structure, consisting of an input layer, one or more hidden layers and an output layer. ANNs are fed data through the input layer, process that information and provide an output. During training, the network adapts itself by updating the neuron parameters, to produce a more accurate output [60].

Depending on the network dimension, the training process may require extensive computations to calculate and update millions of parameters in runtime. DL models also rely on large and diverse volumes of data to improve the accuracy of their predictions. Consequently, although DL was first theorized in the 1980s, at the time the hardware and data requirements could not be met.

When it comes to data scarcity, the rise of the Internet was the turning point, making it possible to collect all kinds of data (e.g., image, video, natural-language datasets). Moreover, the Internet also helped to promote data labeling (e.g., using Flickr image tags and Google reCAPTCHA service), which is crucial for training DL models [33]. Similarly, recent advances in hardware have been triggered by the video game industry, which required high computing power to render photorealistic 3D scenes in real-time. To address this issue, companies like NVIDIA and AMD invested billions of dollars in developing high-performance graphics chips (GPUs) [33]. As a result of these advances, DL has been growing at an accelerated pace, and is expected to become the most influential ML approach [104].

There are several DL architectures, such as *Convolutional Neural Networks (CNNs)* and *Recurrent Neural Networks (RNNs)*, which have been applied in multiple domains, including computer vision [94, 53], speech recognition [36, 39] and natural language processing [131]. With its capability of performing predictions, DL has also been used to address real world problems, achieving remarkable results in terms of image classification [22] and cancer detection [63], and even surpassing human expertise when playing video games [73].

Despite being possible to develop DL solutions from scratch, there are some ANNs that turn out to be quite challenging to implement, due to their large and detailed structure [23, 124]. Existing DL frameworks (e.g., TensorFlow [1], PyTorch [83], Caffe [46]) abstract the underlying algorithms by offering built-in components that ease the development of complex models and the loading of training data. Additionally, most frameworks also support pre-trained models, allowing inexperienced users to take advantage of DL applications. Since these tools are optimized to improve model training performance and to efficiently manage computational resources, they provide a more productive and safe way of using DL techniques.

Given that DL training is a time-consuming task, DL frameworks usually comprise a fault tolerance mechanism, which allows resuming training upon failures [1, 91]. Furthermore, most of these frameworks also provide optimized data loading features, such as TensorFlow `tf.data` *Application Programming Interface (API)* [112] and PyTorch `DataLoader` class [90], since *Input/Output (I/O)* represents a major bottleneck in DL training [129, 78, 88].

Although DL frameworks are widely employed by the science community, they are also of interest for several organizations (e.g., Twitter [64], Airbus [15]).

1.1 PROBLEM

One of the most common obstacles in DL is overfitting [14]. This phenomenon occurs when a model learns the details and noise of the training dataset, almost "memorizing" the data instead of learning from it. Regardless of achieving great accuracy with the training dataset, an overfitted model does not generalize well to unseen data. The smaller the training dataset is, the more models can fit the data, therefore being more likely to occur overfitting [32]. Considering this, models should be trained with large and diverse datasets [111]. Another strategy to avoid overfitting is to use *shuffling* [8], which consists of reading the data multiple times in a random order, creating batches representative of the overall dataset.

To keep pace with the increasing complexity of DNN models and accelerate their execution, numerous advances have been made to provide enhanced computing power. From high-performance processors (e.g., NVIDIA GPUs [125], Google *Tensor Processing Units (TPUs)* [47], Arm Cortex-M Microcontrollers [108]) to fast storage devices (e.g., *Non-Volatile Memory (NVM)* [134], Persistent Memory devices [43]) and low-latency networks (e.g., Mellanox InfiniBand [109], Intel OmniPath [11]), most efforts are based on hardware improvements. Since DL model training requires significant amounts of data, it does not only depend on computing power but also demands fast I/O. Although advanced computing units are supposed to accelerate training, slow data loading will often keep processors idle, extending the training time. Ultimately, if the data loading stages of the DL pipeline are not as efficient as the computational ones, it can lead to a waste of resources.

While small datasets (e.g., CIFAR-10/100 [52], MNIST [58]) can be cached, larger ones (e.g., ImageNet [30], YouTube-8M [2], Open Images [56]) do not always fit in memory. Since the entire dataset is fetched multiple times during training, in case there is not enough memory to cache all the samples, some data must be read from backend storage systems. When it comes to random file access, none of the storage devices (e.g., *Solid-State Drive (SSD)*, *Hard Disk Drive (HDD)*) are as fast or efficient as *Random-Access Memory (RAM)* [44]. Therefore, having to access backend storage using conventional file systems (e.g., Ext4 [67], XFS [110]) adds a significant overhead to the training process. This is even more noticeable on distributed infrastructures, such as the ones provided by *High-Performance Computing (HPC)* centers, where shared file systems like *General Parallel File System (GPFS)* [103] and Lustre [127] have to deal with massive concurrent I/O. For example, in one of its research projects [135], Texas Advanced Computing Center (TACC) highlighted this issue:

> *"As the dataset grows larger, the metadata and data traffic of thousands of directories and millions of files can easily saturate the existing shared file system due to the high access frequency, concurrency, and the sustained I/O behavior."*

Thus, fully training a DNN until convergence is reached is an extremely slow process that can take days or even weeks [136]. To address this issue, DL frameworks support a variety of file systems and take advantage of optimized data formats (e.g., TFRecord [120]) and structures (*Lightning Memory-Mapped Database (LMDB)* [20]). Several frameworks also provide I/O optimizations (e.g., TensorFlow `tf.data` API [112], PyTorch `DataLoader` class [90]) that minimize the data loading overhead. These optimizations are embedded in the framework itself and cannot be decoupled and applied to other frameworks. This monolithic approach demands a wide understanding about the framework internal operation model, to be able to extend and adapt its features, causing this to be a complex and time-consuming task.

Having multiple I/O optimizations for the same problem, generates redundant code and prevents the sharing of efficiency gains from optimizations between different frameworks [9]. Moreover, most I/O optimizations have to be manually configured, which requires additional time to understand each configuration parameter (e.g., number of parallel calls, prefetch factor) and to find the optimal configuration. Other optimizations, such as TensorFlow `tf.data.experimental.AUTOTUNE` feature [113], comprise greedy provisioning algorithms that allocate resources based on the hardware capacity, instead of taking into account the I/O requirements of the workload. Such an approach may lead to shared file system saturation, due to excessive concurrent metadata access [19, 135, 19].

To fully address these challenges, we would require an I/O optimized framework-agnostic middleware that provides an autotuning mechanism to determine the optimal configuration. This strategy would prevent the user from unnecessary system tuning and waste of computational resources.

1.2 Objectives and Contributions

The main goal of this dissertation is to improve the current design of I/O optimizations used by DL frameworks. First, I/O optimizations should be decoupled from the framework itself, to improve their extensibility (i.e., extend existing mechanisms with innovative features) and adaptability (i.e., to attend the requirements of diverse DL workloads). Second, due to the current monolithic design of DL frameworks, it is increasingly challenging to reuse I/O optimizations between different DL frameworks. As such, exporting these mechanisms outside the DL framework would also allow them to be applied to other DL systems. Third, as several I/O optimizations require different tuning parameters for distinct workloads and use cases, these should be self-tuned and dynamically adapted throughout time, preventing waste of computational resources.

This dissertation proposes a novel framework-agnostic storage middleware that improves the I/O performance of local DL applications. To achieve this, this dissertation makes the following contributions:

- As a first contribution, we have conducted a thorough study of the I/O optimizations that impact the performance of local DL training. Being one of the most widely used DL frameworks, TensorFlow features an extensive range of data loading optimizations that are provided by its `Dataset` API. Thus, an in-depth analysis of the TensorFlow `Dataset` API was carried out, followed by its evaluation on a local setup using diverse training workloads. This study proved that local DL applications, depending on the model used for training, can benefit significantly from parallel I/O and data prefetching.
- As a second contribution, based on the insights of the conducted study, we propose PRISMA, a novel storage middleware that decouples existing I/O optimizations of DL frameworks, performing parallel I/O and data prefetching to accelerate DL training. Furthermore, PRISMA provides an autotuning mechanism to define the number of threads used for reading data and the size of the buffer where prefetched data is stored. This algorithm was designed to achieve a good trade-off between the performance of DL training and resource usage. PRISMA is framework-agnostic, meaning that it can be easily extended and applied to other frameworks, which prevents the production of redundant code.

- The third contribution of this dissertation consists of integrating a PRISMA prototype with TensorFlow and PyTorch, proving that PRISMA design allows to apply I/O optimizations to different frameworks and storage contexts.
- As a final contribution, we conducted an extensive experimental evaluation comparing PRISMA with TensorFlow and PyTorch under different workloads. The experiments were carried out on the *AI Bridging Cloud Infrastructure (ABCI)*, as part of a collaboration that has been established with the *National Institute of Advanced Industrial Science and Technology (AIST)*, regarding the optimization of storage systems to improve the performance of HPC infrastructures. The evaluation was conducted using the *ImageNet Large Scale Visual Recognition Challenge* 2012 *(ILSVRC*2012*)* [98] dataset, as well as several training models, namely LeNet [59], AlexNet [53], and ResNet-50 [38], which are classified as being either I/O-bound or compute-bound. Results demonstrate that PRISMA outperforms the baseline version of TensorFlow by up to 54%. Additionally, it performs similarly to an I/O optimized version of TensorFlow, with the benefit of consuming less 15% of *Central Process Unit (CPU)* and allocating fewer threads. When compared to PyTorch, PRISMA achieves a performance advantage of up to 63%, for a typical configuration with less than 8 worker processes.

1.3 Document structure

The rest of this document is organized as follows. Chapter 2 surveys the basic concepts of DL as well as the impact of the current I/O operation mode, and describes existing related work on I/O optimizations for DL. Chapter 3 describes the preliminary study that was performed concerning the I/O operation model of TensorFlow. Chapter 4 presents the PRISMA storage middleware, providing both design and implementation details about the developed prototype, and describing the integration process with TensorFlow and PyTorch. Chapter 5 provides a comprehensive experimental evaluation conducted with PRISMA when applied to the TensorFlow and PyTorch DL frameworks. Finally, Chapter 6 presents the final remarks of this dissertation and discusses open challenges and prospects of future work.

2

STATE OF THE ART

Since the main topics of this dissertation concern DL, I/O optimizations, and storage systems, this chapter surveys basic concepts of DL as well as the impact of the current I/O operation mode. We also revisit related work on improving the I/O performance of DL applications, identifying current problems, and explaining how does our solution differ from existing ones.

2.1 BACKGROUND

As previously stated in the document, DL is a technique suitable for automatically recognizing patterns in sound, text, or images, using ANNs and a large amount of data. With this in mind, this section clarifies what an ANN and its training process consist of, as well as describes the implications of using small datasets. Furthermore, common techniques used to optimize DL training performance are also presented. The content provided below can easily be skipped by readers who are already acquainted with these subjects.

2.1.1 *Artificial Neural Networks*

An ANN is a model inspired on the human brain, commonly represented by a collection of interconnected nodes, called *artificial neurons*, that are organized in *layers* [33]. Each layer processes the data fed into it, extracting meaningful features for the problem it is trying to solve. Therefore, all the layers chained together implement a form of progressive "data distillation" [33]. The first layer of a network is often called *input layer* and the last layer is the *output layer*. All layers in between the input and output layers are called *hidden layers*. Neural networks with two or more hidden layers are usually called DNNs.

Using data, a neural network is capable of training itself to produce a certain output. Figure 2.1 illustrates the architecture of an ANN with 4 layers and 14 neurons.

What a layer does to its input data is specified by a set of attributes, called *parameters*. Every connection of the neural network, commonly referred to as *synapse*, has a *weight* value associated with it that can either be a negative or a positive number. Essentially, the weight represents how much influence the input carried by that connection will have on the output, i.e., the importance of the input for the problem at hand. Changing an input whose weight is almost zero will not affect the output result. On the other hand, increasing an input with negative weight will decrease the output whereas increasing one with positive weight will do the opposite.

The output of the network is defined by an *activation function*. Each neuron uses this activation function to define the output that will pass on to the next layer of neurons. These functions also

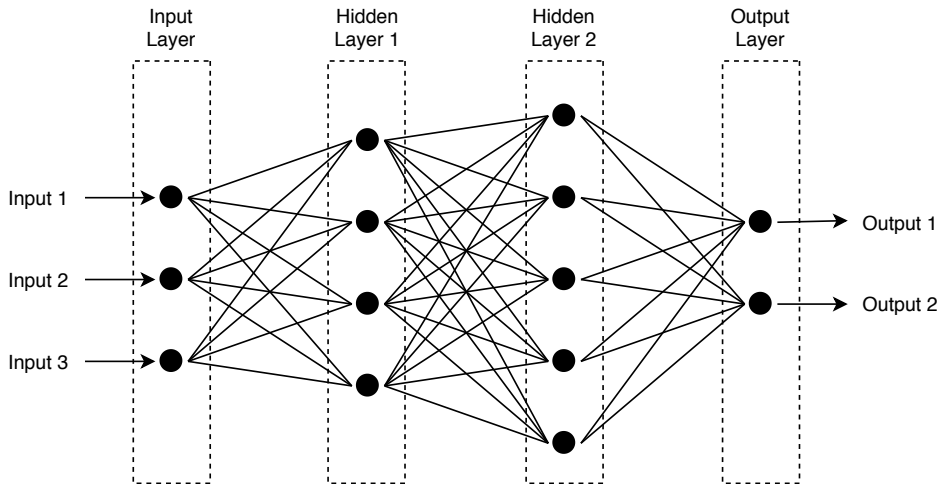


Figure 2.1: Architecture of an ANN.

determine whether the neuron should be activated or not, based on its inputs and parameters. Activation functions also help normalize the output of the neurons to a certain range. *Rectified Linear Unit (ReLU)* [133], *Scaled Exponential Linear Unit (SELU)* [50] and *Hyperbolic Tangent (TanH)* [132] are three activation functions frequently used in the DL industry.

Apart from the weights, another parameter is called *bias*. Bias is simply a constant value that is added to the product of inputs and weights, guaranteeing that even when all the inputs are zero, the neuron is still going to be activated. This is used, for instance, when we want our model to return a positive value, although it received zero as input. This parameter can be seen as an extra input neuron, which simply stores the value '1'. Similarly to other neurons, each connection with the bias neuron has its own weight. Consequently, as '1' is the absorbing element for multiplication, the bias value of each neuron will be equal to the weight of its connection with the bias neuron. Figure 2.2 details the operations that occur in one neuron of the neural network.

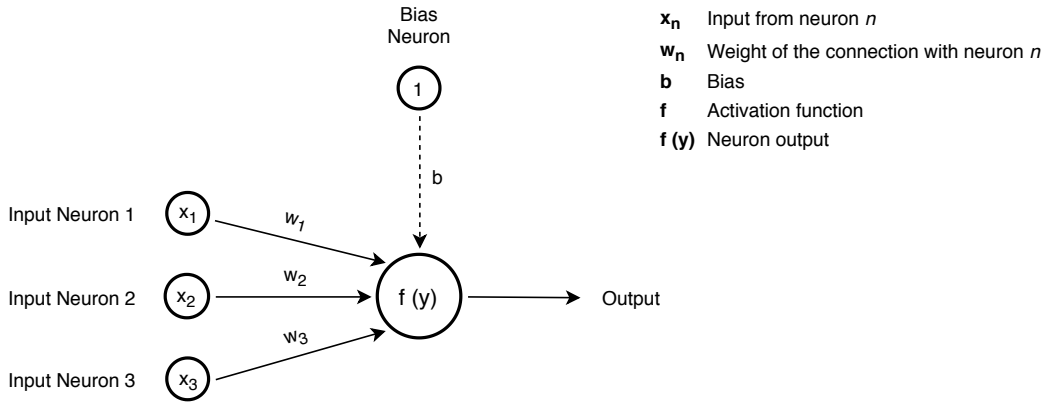


Figure 2.2: Operations that occur in one neuron.

Considering that the activation function is applied to the weighted sum of its inputs, in the above example, y can be represented by Equation 1, where N is the number of input neurons.

$$y = \sum_{i=1}^{N=3} x_i w_i + b = x_1 w_1 + x_2 w_2 + x_3 w_3 + b \tag{1}$$

2.1.2 *Types of Learning*

There are three main types of learning: *supervised learning*, *unsupervised learning* and *reinforcement learning*.

A supervised learning algorithm maps input data (e.g., images) to known targets (also known as labels), so the output of the model can be compared with the correct output that should be obtained [33]. For example, supervised learning can be compared to the typical school learning process, where a teacher informs whether the students' responses are correct or incorrect and they learn from it.

In contrast, unsupervised learning uses only unlabeled input data and instead of learning from targets, it finds similarities or associations between the data samples and organizes them accordingly [33]. An analogy of unsupervised learning is when a doctor diagnoses a disease based on a patient's symptoms. When analyzing the similarities between what the patient described and the common effects of the disease, the doctor is capable of classifying the patient's situation. If instead of concluding on his own the doctor resorted to medical exams that determined what was the patient's condition, this would be a case of supervised learning.

Finally, reinforcement learning finds the best possible behavior to maximize the reward in a certain scenario, i.e., uses positive and negative feedback to determine what actions to perform [33]. As reinforcement learning has been widely applied to video games [73, 107, 10], this is the best example to understand how it works. When used on a game, reinforcement learning attempts to find the actions that allow winning the game or reach the maximum score, depending on the context.

Even though there are different learning methods, the majority of applications that use DL have adopted supervised learning [33]. As such, the following sections will be described only considering supervised learning.

2.1.3 *Training Process*

The diagram presented in Figure 2.3 describes how the training process of a DL application occurs.

To understand how well an ANN can make predictions about a given dataset, a *cost function* is used to measure how far the output of the ANN is from the expected one. To achieve this, the cost function calculates the difference between the obtained predictions and the actual targets, obtaining a value known as *cost*. After determining the cost, that value is used as a feedback signal to adjust the network parameters, which is the responsibility of the *optimizer* (e.g., *Stochastic Gradient Descent (SGD)* [96], Adam [49], RMSProp [122]). The optimizer is usually based on the Gradient Descent algorithm, consisting of an algorithm that is used to determine the weights and biases that minimize the cost function. Until finding the optimal parameter values, the optimizer repeatedly computes the gradient that relates changes in the network parameters to changes in the cost function, and uses that gradient to find the minimum of the cost function [76]. Backpropagation [97] is the method

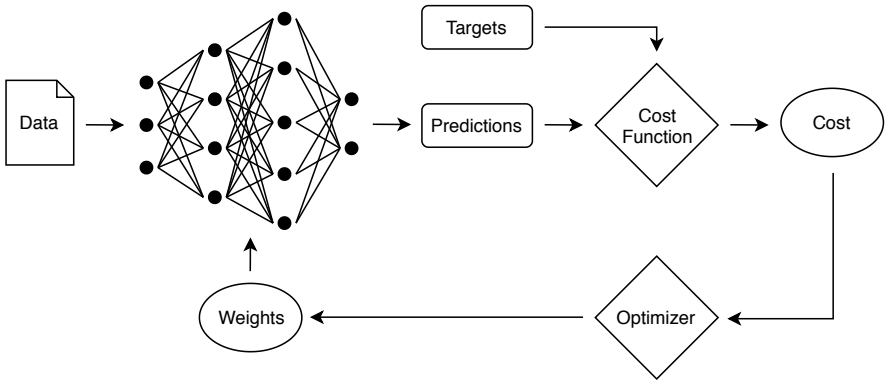


Figure 2.3: DL training process.

employed by the optimizer to efficiently compute gradients, propagating the cost value backward from the output layer to the input layer.

At the beginning of the training process, the network parameters are assigned random values. Therefore, at an early stage, most of its predictions will be wrong, presenting a high cost value. As the network is being trained, i.e., as more samples are processed, the parameters are updated and the cost decreases, improving the quality of the predictions. Accordingly, a trained model is a network with a minimal cost, which outputs are as close as possible to the targets [33].

When the dataset used to train the model is so large that there is not enough memory to compute the gradients, it is divided into smaller batches. A *batch* consists of a subset of samples taken from the training dataset. Usually, before training, a *batch size* is defined, indicating the total number of training samples present in a single batch. The process of passing the entire dataset through the network is called a training *epoch*. Taking this into account, the number of *iterations* is equal to the number of batches required to complete an epoch. To clarify this terminology, with a dataset composed of 5000 samples and a batch size of 100, each epoch would need 50 iterations to go through all the samples. In addition, the batch size also specifies the number of samples to process before adjusting the network parameters, hence the parameters are updated at the end of each iteration. The batch size and the number of epochs used in the training process are configurations external to the model whose value cannot be estimated from data, therefore they are called *hyperparameters*.

2.1.4 *Avoiding Overfitting*

As mentioned in Section 1.1, when the training conditions are not the most adequate, a phenomenon called overfitting may occur. Overfitting happens when a model "memorizes" the data instead of learning from it, not generalizing well for new and unseen data. On the other hand, underfitting is also a problem characterized by extremely simple models, which have negligible variance in their predictions and exhibit high bias towards wrong outcomes. Figure 2.4 exemplifies the difference between overfitting, underfitting and appropriate fitting. The batch size, the number of epochs and the training data are some of the elements that may cause the model to overfit.

There is no ideal number of epochs or batch size to choose for all scenarios. In fact, these hyperparameters must be selected taking into account the dataset used. However, poorly chosen

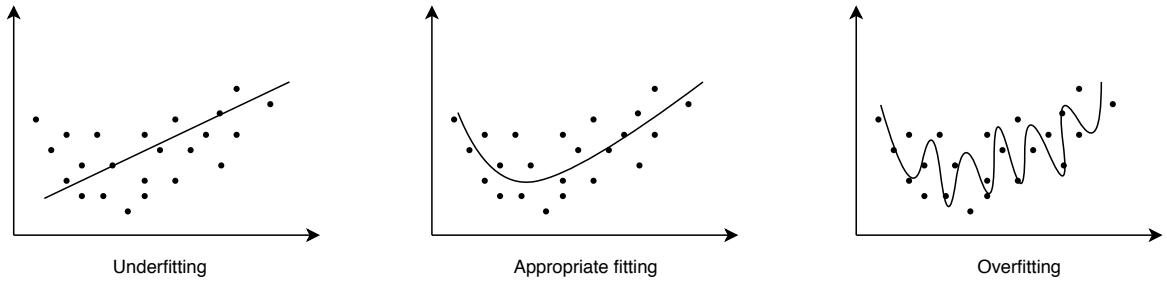


Figure 2.4: Difference between overfitting, underfitting and appropriate fitting.

configurations may cause overfitting. There are certain factors that must be considered when selecting a dataset, since the quality of the training data is crucial to the success of the model predictions. First of all, training data must be diverse to cover as many cases as possible, and it must also be sufficient so the model can learn from it. As a result, small datasets generally are not the best option for training a neural network. On the other hand, larger datasets usually do not fit in memory, being necessary to collect training data from the backend storage system during training. Apart from this, especially when the data is sorted by their class/target, data shuffling is required to obtain batches representative of the overall dataset. To guarantee that no sequence of samples is repeatedly used across many iterations, the dataset must be shuffled at the end of each epoch. Specially when the dataset is composed of small samples, fetching shuffled data is one of the most costly processes of DL training, since traditional storage systems are designed for large and sequential reads rather than small and random ones. Ideally, shuffling should be performed using the entire dataset (i.e., global data shuffling), otherwise the accuracy of the model may be affected.

In summary, although large and diverse datasets must be used to train neural networks, frequently accessing the backend storage system adds a substantial overhead to the training process. The same applies to the shuffling mechanism, which is essential to prevent overfitting.

2.1.5 *Parallel Deep Learning Training*

DNN training can be parallelized using two approaches: *data parallelism* or *model parallelism*. Data parallelism is often used when the dataset is considerably large, i.e., when there is a large number of batches to process [51]. To speed up the training process, every compute node trains the full model (i.e., all parameters) with a different portion of the batch. At the end of each iteration, the compute nodes have to synchronize and communicate with each other, in order to estimate the average gradient with respect to the whole batch [7]. This process causes communication overhead, possibly impacting training performance [51].

When the DL model has numerous layers and parameters, being too large and complex to fit in the GPU memory, it is common to resort to model parallelism [29]. Model parallelism involves the partition of the neural network across multiple compute nodes. However, unlike data parallelism, with this method all nodes use the same training batch.

Figure 2.5 describes the two ways of parallelizing training, data parallelism (top) and model parallelism (bottom).

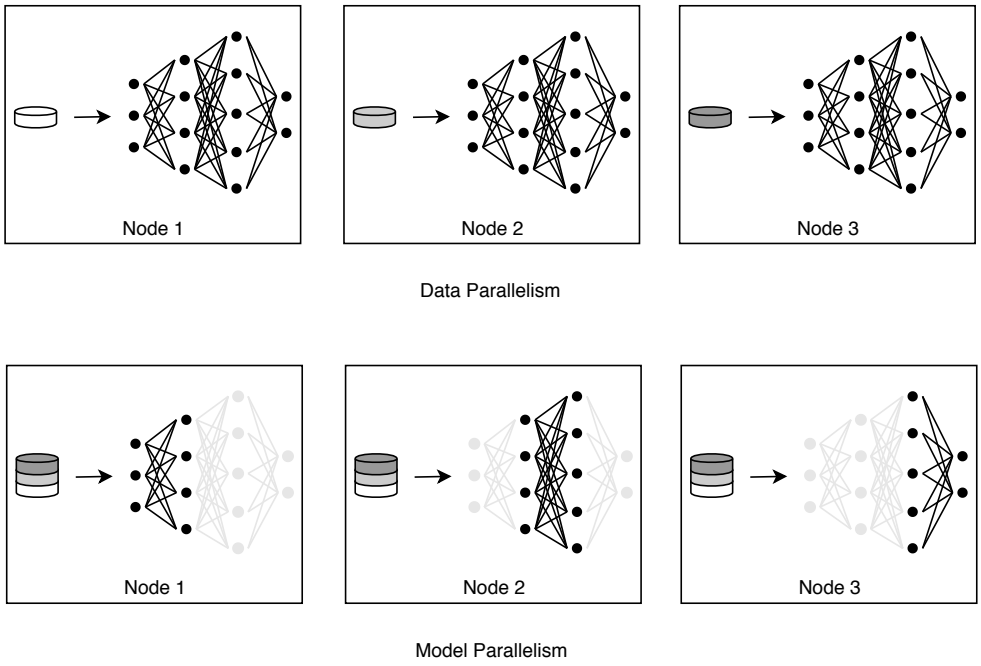


Figure 2.5: Data and model parallelism for DNN training.

Given that when using model parallelism each compute node sits idle waiting for data from previous nodes, there is no actual parallel computation. On the contrary, with data parallelism each node performs training simultaneously. Given this, and since data parallelism scales better for CNNs than model parallelism [51], the former approach is currently supported by the vast majority of DL frameworks [7].

2.1.6 *Optimized Data Formats*

DL frameworks resort to optimized data formats and structures, such as TFRecords [120], *Hierarchical Data Format version 5 (HDF5)* [121], and LMDB [20], to improve the I/O performance of DL training. Optimized data formats allow multiple samples to be grouped into a single file, promoting sequential I/O to large files. Since traditional file systems are designed for large and sequential reads rather than small and random ones, training a model using optimized formats achieves significantly higher performance than using raw image files [62].

While LMDB and HDF5 were not specifically designed for DL, the TFRecord file format was created by TensorFlow [1] as its own binary storage format. Binary data can have a positive impact on training performance since it takes less space to store and can be accessed faster. Nevertheless, using TFRecords also has downsides. To benefit from TFRecords it is necessary to convert the dataset to this format in the first place. A TFRecord is a large batched file with several training samples inside, which allows performing large sequential I/O. However, when the dataset does not fit in memory, using TFRecords may have a negative impact on the accuracy of the model, since it prevents from performing global data shuffling. This happens due to the fact that TensorFlow uses a buffer to

shuffle data, causing the buffer size to be a limiting factor. If the size of the shuffle buffer is not large enough, only partially shuffled samples are obtained [138].

LMDB consists of a B+ tree based *Key-Value (KV)* storage, while the HDF5 file format makes it possible to store multiple samples in a single file. Although LMDB and TFRecord have similar performances [37], HDF5 falls slightly behind when compared to LMDB [62].

2.2 Related work

According to the vision for storage research conducted in 2018 [4], the growing demands of AI workloads need to be met by more complex and efficient storage technologies. In this regard, there has been a number of storage solutions to improve the performance of AI applications [61, 68, 102, 48], however only a few of them focus on a major performance issue: reading large amounts of random data.

2.2.1 *Storage I/O Performance Studies*

To understand what may be preventing DL training from being more efficient, several studies have been conducted concerning the I/O performance and scalability of DL applications. These studies approach both local and distributed setups, and are mainly focused on HPC environments.

I/O performance of DL applications

Recent studies have shown that the I/O performance of DL applications is non-negligible and can easily become the bottleneck, especially in single node scenarios [129, 78, 88].

According to Han et al. [37], during training the GPU utilization increases with the number of layers of the DL model, whereas the throughput decreases. The larger the parameter size of a model, the more expensive is the communication between the processing units, given that more data has to be transferred, and a higher percentage of GPU is used.

Han et al. [37] compare the performance of Lustre parallel file system [127] to a local *Non-Volatile Memory Express (NVMe)* storage drive. Both storage solutions experienced similar performances while training DL models with the same dataset, yet NVMe can slightly enhance the training throughput of DL models where computational overhead is not significant. This can be justified by the fact that Lustre is accessed by multiple nodes and might create a network bottleneck with the increasing number of I/O requests. NVMe SSDs are local to each node, so there is no network overhead.

In terms of scalability, training performance improves when increasing the number of GPUs on each node. Nonetheless, this only happens to a certain extent, as eventually, the communication overhead between the participating GPUs outweighs the benefits of parallelizing training [37].

A performance characterization of DNN training using TensorFlow and PyTorch [83] was also carried out in [45]. The experiments showed that the training throughput does not increase linearly with the batch size and that, generally, increasing the number of worker processes and batch size has a positive effect on performance. As a result, Jain et al. [45] suggest to use multiple processes even with a single node setup. Furthermore, in all the conducted experiments, PyTorch outperformed TensorFlow.

Analysis of data formats

Training DNNs with raw image files on local file systems revealed to be a major disadvantage [62], especially when compared to storage solutions with efficient indexing and caching strategies, such as the LMDB KV store [20]. On the other hand, the TFRecord file format [120] and LMDB KV store have very similar performances on Lustre file system [37].

Surprisingly, with LMDB the read time, process sleep time, and number of context switches all increase with the number of application processes [88]. Thus, using LMDB with a multi-processing approach might lead to performance degradation [88].

I/O characterization of TensorFlow

A study about the I/O performance of TensorFlow was also conducted in [17], showing that increasing the number of parallel I/O threads also increases the bandwidth utilization for file ingestion, regardless of storage drive or storage system (e.g., HDD, SSD, Intel Optane, Lustre file system). Moreover, increasing the batch size reduces the execution time due to a higher utilization of the GPU. Overlapping I/O and computation using prefetching generally increases the number of bytes read. However, for more efficient storage devices, the benefits of this technique are less obvious, as a consequence of its high I/O ingestion rate.

While assessing the performance of BeeGFS [6], some important observations were drawn about TensorFlow [19]. TensorFlow helps optimize file read time by lessening the read access size, however the metadata overhead remains high, because the files are required to be opened anyway. Additionally, the total amount of data read increases proportionally with the number of processes involved in the application, consequently increasing the bandwidth. As the number of nodes rises, although read time suffers a minor decrease, the metadata operation latency grows considerably, preventing the training performance from scaling with the number of nodes. This being said, metadata handling represents the main bottleneck of TensorFlow input pipeline. It is also important to keep in mind that a massive amount of small reads eventually impairs the overall throughput of the file system.

2.2.2 *Storage I/O Optimizations*

Given that I/O represents a major bottleneck in DL training, several solutions have been developed to address this issue. Therefore, below will be presented existing state of the art optimizations designed to improve the I/O performance of DL applications. Despite the focus of this dissertation being local DL training, we also discuss optimizations for distributed scenarios that are worth mentioning.

Data loading pipeline

A commonly used optimization consists of overlapping I/O and model computation by introducing a data loading pipeline in the training process.

Serizawa and Tatebe [105] developed a method that extends the `MultiprocessIterator` class of the Chainer framework [123], by introducing a pipeline with three stages: generating index lists, prefetching data, and generating mini-batches. Each index presented in the index list refers to a sample that will be part of the mini-batch. At the prefetching data stage, the samples in the index list

are retrieved in parallel by multiple processes from the shared storage and placed in the local storage. At the generating mini-batches stage, the samples are read from local storage and a mini-batch is created. In the first and second stages, the index lists are queued to be processed in the next stage and, at the generating mini-batches stage, the resulting mini-batch is placed in a queue to be consumed for model training. Despite the fact that this solution outperforms reading the dataset directly from Lustre using Chainer standard class, it cannot be applied to other DL frameworks.

DeepIO [137] also uses a pipeline to overlap data loading and training. However, this framework was designed for large-scale DL on HPC systems, so it is more focused on scenarios where there are multiple compute nodes, each storing a subset of the training data. To define which samples will be part of the next mini-batch, DeepIO uses a method called *Entropy-aware Opportunistic Ordering (EOO)*. To reduce access to the storage system, EOO uses only samples that are available in an in-memory cache. If the total memory available in all compute nodes is much smaller than the training dataset, this mechanism can affect the randomization level of the mini-batches, having a negative impact on the accuracy of the model. In general, the DeepIO API has better performance than TensorFlow Dataset API [137]. Nevertheless, DeepIO was implemented as a prototype over TensorFlow, and although it achieved promising results with this framework, its applicability and success with other DL frameworks are not guaranteed to be the same.

AIStore [3] is a storage system that provides a *Representational State Transfer (REST)* interface to read and write data, relying on *Hypertext Transfer Protocol (HTTP)* redirects and sharded datasets to achieve high I/O performance. Once again, this storage system features an integrated data processing pipeline, only this one can be executed directly on storage, while posting tensors via *Remote Direct Memory Access (RDMA)* into GPU memory. A storage format, called WebDataset, was also defined so that the adoption of sharded sequential storage would be faster. WebDataset is supported by a Python library that provides a replacement for the built-in PyTorch Dataset class. While training PyTorch-based ResNet-50 [38] models, AIStore performs worse than reading data directly from a local SSD [3]. Moreover, AIStore was only tested with PyTorch, being unknown its impact on other DL frameworks.

Parallel I/O

Many storage solutions focus on parallelizing I/O to maximize throughput performance, therefore improving data reading efficiency.

PyTorch `DataLoader` class [27] uses concurrent background worker processes to load multiple batches in parallel, however within the loading of a batch the individual samples are preprocessed sequentially by a single thread. Given this, the PyTorch `DataLoader` implementation was modified so that each background worker used multiple threads to preprocess samples in parallel [130]. The experiments performed using this method with different worker/thread combinations when loading the ImageNet dataset [98] show that, in general, using more threads and workers increases the data loading rate. Despite this, with 8 or more workers, increasing the number of threads beyond 4 has no effect on the data loading rate [130]. It is important to mention that during these experiments no training was performed, only data loading. Since this technique was implemented on PyTorch, it cannot be applied to other DL frameworks.

Although TensorFlow allows concurrent processing of multiple input files using its `map` operator, HDF5 [121] serializes all operations, inhibiting parallel executions. To overcome this problem, the

`multiprocessing` Python module was used to enable the concurrent preprocessing of multiple input files [55]. Although it appears to be effective in improving the reading performance of DL training files, when using other data formats than HDF5 this optimization is not necessary.

LMDBIO [88] is a plugin designed for optimizing the LMDB KV store for DL. Despite the fact that LMDBIO provides some innovative strategies (e.g., speculative parallel I/O, I/O staggering, shared-memory buffers) to take better advantage of I/O parallelism, it was specifically implemented for LMDB, so it cannot be used when the dataset is stored in other data structure or format.

Replace I/O requests

Discarding delayed I/O requests or replacing them for data that is already available in memory is also used to improve the training performance of DL workloads.

Quiver [54] is a distributed cache for DL training that reuses data across multiple jobs and users operating on the same dataset. This storage solution dynamically prioritizes cache allocation to jobs that benefit the most from caching. To achieve this, Quiver uses controlled probing to measure the performance of DL jobs with and without caching. In addition, Quiver employs a technique of replaceable cache hits that allows taking more advantage of the cache content, providing thrash-free caching. Quiver partitions the datasets into a fixed number of chunks. Even though the chunking of a dataset ensures randomness of the data within each chunk, this may not be sufficient to prevent the model from overfitting. Also, Quiver requires an additional processing step before initiating the training phase for partitioning the datasets. When training the ResNet-50 model, Quiver can achieve better performance compared to reading data from the remote Azure storage blob [69]. Despite that, Quiver is implemented in PyTorch and its effect on other DL frameworks is unknown.

Apart from Quiver, another proposed I/O optimization consists of mitigating the effect of I/O tail latency, by monitoring I/O requests of DL applications and discarding the delayed ones [79].

Data echoing

Data echoing [18] consists of reusing intermediate outputs from earlier data pipeline stages in order to reduce the upstream computation. To use data echoing an extra stage needs to be added to the pipeline, repeating data from the previous stage. Data echoing is only efficient if placed after a low performance stage so, to take advantage of this strategy, it is first necessary to identify the major bottleneck of the pipeline. When using data echoing there can be duplicate samples in each batch, possibly leading to model overfitting. To reduce the probability of duplicate samples, the amount of shuffling can be increased for echoed samples, at the cost of additional memory.

Based on the experiments that were conducted in [18], when training the ResNet-50 model with ImageNet, performing batch echoing actually requires more fresh samples (i.e., dataset samples read from disk) than the baseline version with no data echoing. This may delay the training process, contrary to what would be expected. In addition, data echoing can only be applied to DL frameworks that support flexible data loading pipelining.

Transfer data directly to GPU

An alternative strategy to improve the I/O performance of DL workloads consists of reducing the data path from storage to GPUs [57]. This strategy uses GPUDirect RDMA [35] to reduce the CPU

involvement and the latency of the system by allowing the GPU to read data directly from storage. To achieve this, a *Data tracking tool (DTT)* that provides the data pointers for each training iteration was developed. DTT allocates memory on the GPU to store the batches that are going to be used in future iterations.

During each training iteration several memory transfers are performed between the host and the GPU, which can interfere with the memory copies orchestrated by DTT. This interrupts training until the memory copy is finished, delaying the process. Additionally, two memory copies cannot be executed simultaneously, even if submitted by different streams. Using DTT with large batch sizes may also decrease performance, due to saturation of the GPU. Aside from this, when using DTT, no data preprocessing can occur online, requiring some extra time and storage space to perform the preprocessing beforehand.

According to [57], when comparing Caffe using DTT with its original version, the former provides more than a 2× speed up than the latter. However, in the version with DTT there was no online preprocessing, which practically justifies the performance improvement obtained, since the original version had the overhead of preprocessing data during training. Identical to other solutions, this one was also based on a specific framework, Caffe, so its impact on other DL frameworks is unknown.

Metadata Management

The larger the dataset used for training, the more metadata needs to be handled. Metadata management proves to be a problem in distributed setups where high access frequency and concurrency can easily saturate a shared file system [135]. In particular, several parallel file systems, such as Lustre, are designed with a centralized metadata server (replicated for availability) and multiple storage servers. Because the metadata server comprises the namespace of the file system, all metadata requests are destined towards it. This causes the metadata server to receive thousands to millions of requests per second, thus representing a major point of contention [127]. To address this issue, numerous solutions provide efficient metadata management by maintaining a copy of the file system namespace [135, 138] or a metadata snapshot [128] in each compute node. Nevertheless, such an approach may require specific hardware support and custom design for certain applications and computing environments.

2.3 SUMMARY

Before analyzing the storage optimizations implemented within the scope of DL, it is necessary to understand that a neural network is composed of several layers that have parameters associated with them. These parameters define the behavior of the network and need to be updated to improve the quality of the model predictions. The training stage consists of processing training samples repeatedly and adjusting the network parameters according to a cost function.

One of the most common obstacles in DL is overfitting. This phenomenon can have multiple causes, however using a large and diverse dataset and shuffling the dataset before each training epoch may prevent the issue. Since training complex models with large datasets is a time-consuming task, numerous techniques can be applied to improve DL training performance, such as adopting optimized data formats (e.g., TFRecord, LMDB) and using data or model parallelism.

Multiple studies have been conducted focusing on the performance and scaling of DL applications [37, 45, 129, 78, 88], which prove that I/O can easily become the bottleneck of DL training. Other studies analyze the I/O performance of TensorFlow [17, 19] and LMDB [88], as well as present its scalability limitations. There is also some research about specific data formats, showing that training DL models with raw image files has a major impact on performance [62].

Data prefetching [105] is one of the I/O optimizations applied in I/O-bound DL workloads. However, this strategy is only effective if the reading and preprocessing of data are faster than computation. Moreover, parallelizing I/O operations also revealed to be efficient in speeding up the training process [88, 130, 55], although the number of concurrent threads/processes may be limited by the underlying backend storage.

Other optimizations concern data loading pipelining [105, 3, 137], data echoing [18], discarding [79] or replacing [54] delayed I/O requests, transferring data directly to the GPU [57], and efficiently managing metadata [135, 138, 128]. Nevertheless, all these strategies present drawbacks.

Most of the optimizations already proposed are specific to certain DL frameworks [105, 130, 55, 57] or data formats [3, 88], and can cause model overfitting [137, 79, 54, 18]. Other solutions are designed for scenarios where data is stored in a shared file system, not having an impact on the I/O performance of local data storage [135, 138, 128]. Additionally, it is often difficult to configure storage solutions to obtain the best performance possible, without wasting computational resources (e.g., memory, CPU), especially for DL users who are less familiar with storage. Therefore, it is necessary to find a solution that is agnostic of the DL framework and data format used, that does not interfere with the model accuracy, and that is capable of automatically selecting the optimal configuration.

3

PRELIMINARY STUDIES

TensorFlow [1] is a widely used DL framework, being employed in several sectors, such as healthcare [85], social networks [139], and e-commerce [12]. Due to its vast adoption, TensorFlow is one of the most I/O optimized DL frameworks [17]. As such, to understand if local DL training could benefit from improved I/O performance, this chapter presents a thorough analysis and evaluation of TensorFlow `tf.data.Dataset` API [113]. The experiments were performed in a local setup using diverse training models, comprising both I/O-bound and compute-bound workloads.

3.1 TENSORFLOW DATASET API

TensorFlow `tf.data.Dataset` API allows to create efficient input pipelines. Inside a TensorFlow input pipeline occur multiple I/O operations, known as transformations, that involve reading and further preprocessing of training data. As these transformations can be performed by different threads while computation is being handled by the GPU, TensorFlow input pipeline accomplishes the so-called overlapping of I/O and computation [17, 137]. We now present three main transformations of the input pipeline, namely `prefetch`, `interleave` and `map`. Furthermore, we also describe the autotuning algorithm of the TensorFlow input pipeline, that is used to configure the transformations' parameters.

3.1.1 *Prefetch*

One of the TensorFlow's input pipeline transformations is `prefetch` [117]. This transformation is performed by a background thread that fetches data from the previous pipeline operation and stores it in an in-memory buffer. This buffer uses a double-ended queue implementation so that while the prefetcher thread is buffering elements on one end of the queue, an `iterator` is consuming those same elements on the other end of the queue (i.e., similar to a producer-consumer model). As data is being removed from the buffer, the prefetcher thread is notified, restoring these elements as soon as data is available from the upstream operation. This allows later elements to be prepared while the current element is being processed. As a result, `prefetch` can improve latency and throughput, at the cost of using additional memory to store prefetched elements [113]. Usually, `prefetch` should be executed at the end of the input pipeline, so that there are always batches in the buffer ready to be consumed by the GPU. This prevents the GPU from having to waste time requesting batches to the CPU, every time it finishes a training step.

The size of the internal buffer of the `prefetch` transformation can either be manually set by the user or automatically tuned by the system using the *autotune* feature, which will prompt `tf.data` to

tune the value dynamically at runtime [112]. *Autotune* can be enabled by setting the `buffer_size` argument to `tf.data.experimental.AUTOTUNE`.

To avoid unnecessary processing of elements, `prefetch` uses a variable called `slack_period` that determines how long the thread will sleep before prefetching another element. The `slack_period` is based on the time that an element takes to be consumed by the GPU since it was placed in the buffer. As a result, the slower the elements are being consumed, the greater the `slack_period` will be, and the longer the thread will wait until it prefetches another element, therefore preventing wasting resources [87].

3.1.2 *Interleave*

Another important transformation of the input pipeline is `interleave` [115], which maps a function across the dataset and interleaves the result. Typically, this transformation is used to load training data. To help understand how `interleave` works, let us assume that: a dataset is composed of multiple input elements and each of these consists of several records. Among others, `interleave` uses the following arguments:

CYCLE_LENGTH Number of input elements that will be processed concurrently (i.e., number of input elements whose results will be interleaved).

BLOCK_LENGTH Number of records to interleave from each input element before cycling to another input element.

NUM_PARALLEL_CALLS Level of parallelism used to process the input elements.

Figure 3.1 shows a high-level example of how `interleave` operates, based on the examples provided in the `interleave` documentation [115].

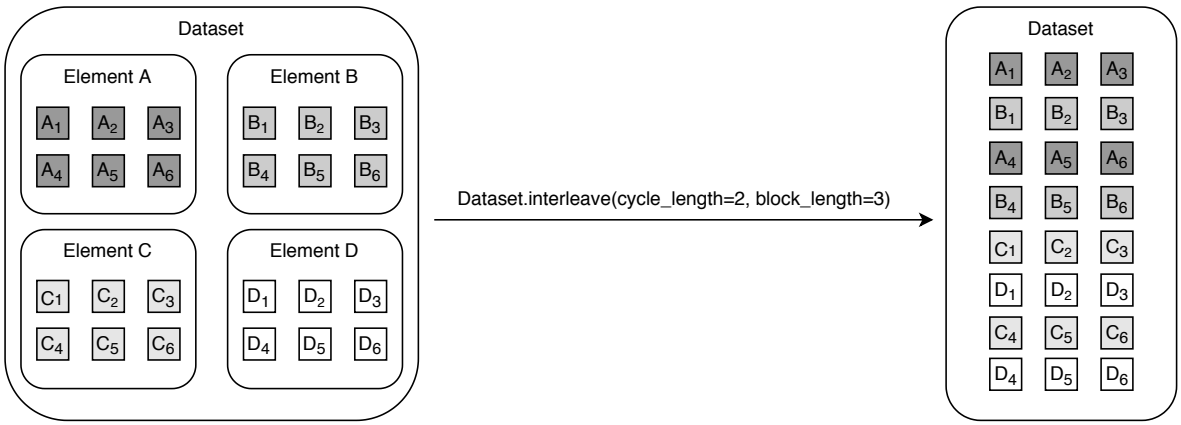


Figure 3.1: Interleave high-level example.

The number of interleave cycles depends on the number of input elements and on the `cycle_length`:

$$\#cycles = \frac{\#elements}{cycle\_length} \tag{2}$$

In the example presented in Figure 3.1, two interleave cycles were performed, one to interleave the records of elements A and B, and another for elements C and D.

If `num_parallel_calls` is set to `None` (default value) then *sequential interleave* [42] will be performed. On the other hand, if `num_parallel_calls` is set to 1 or more (or to `tf.data.experimental.AUTOTUNE`), *parallel interleave* [81] will be executed instead. With *sequential interleave* the elements are synchronously fetched by a single thread. In contrast, *parallel interleave* loads elements concurrently using multiple threads. Similarly to `prefetch`, `interleave` supports an *autotune* feature that delegates the decision about what level of parallelism to use to the `tf.data` runtime [112]. The *autotune* feature can be enabled by setting `num_parallel_calls` to `tf.data.experimental.AUTOTUNE`.

In addition to the number of threads defined by `num_parallel_calls`, referred to as `current_workers`, *parallel interleave* allocates $2 \times cycle\_length$ background threads, known as `future_workers`. The `future_worker` threads are responsible for prefetching the first $2 \times block\_length + 1$ consecutive records of each input element, while the `current_worker` threads fetch the remaining records of the input elements. Moreover, `future_workers` store the prefetched elements in a `future_elements` buffer. As new interleave cycles start, `future_elements` are moved to a `current_elements` buffer, to be processed by `current_workers`. Usually, the input elements are composed of a large number of records, so while a `current_worker` is expected to be executing constantly, a `future_worker` is expected to be short-lived since it only reads a few initial records. The purpose of prefetching future cycle elements is to overlap expensive initialization (e.g., opening of a remote file) with other computation [81].

The argument `num_parallel_calls` controls the maximum number of `current_workers` that can be processing concurrently at any given moment. If *autotune* is enabled, then a maximum of `cycle_length` threads can be executed concurrently. Given this, with *parallel interleave* the maximum number of parallel threads is given by the following equation:

$$max\_parallelism = 2 \times cycle\_length + min(cycle\_length, num\_parallel\_calls) \tag{3}$$

With $2 \times cycle\_length$ being the number of `future_worker` threads and $min(cycle\_length, num\_parallel\_calls)$ the number of `current_worker` threads. Consequently, Equation 3 also represents the maximum number of concurrent reads that *parallel interleave* may perform at a given moment.

3.1.3 *Map*

The `map` transformation [116] is very similar to `interleave`, in a way that it also applies a user-defined function to each element of the dataset, however it does not interleave the result. There are two implementations of `map`, *sequential map* [66], which processes elements sequentially, and *parallel map* [82], which processes multiple elements simultaneously. *Parallel map* also performs prefetching by storing the results of the parallel invocations in an internal buffer until they are consumed by the next pipeline stage. The size of the buffer used by *parallel map* is determined by the level of parallelism. As with `interleave`, when using `map` the `num_parallel_calls` argument can be set to `tf.data.experimental.AUTOTUNE`. While `interleave` is often used to parallelize I/O, `map` is used to preprocess training data.

3.1.4 *Autotuning*

When using `prefetch` with the *autotune* feature enabled, the buffer size is tuned using a legacy implementation that increases the buffer maximum size up to a (predefined) threshold [86]. However, parallel transformations (e.g., *parallel interleave*, *parallel map*) use a more sophisticated autotuning technique, that adapts the level of parallelism and the size of the internal buffers used by each transformation. To achieve this, `tf.data` gathers runtime information about each input pipeline transformation, which is later used to divide the available CPU across parallel transformations. The tuning process is performed periodically by a background task, that minimizes the input pipeline latency using an analytical model of its performance, while taking into account the CPU and RAM budget constraints [74]. Initially, the background task is executed every 10 milliseconds, and over time this interval increases up to a 60 second threshold, using exponential backoff [75]. After performing the background task, the results are propagated to the actual running pipeline. Naturally, different executions of the background task can result in different levels of parallelism and buffer sizes, for each input pipeline transformation.

3.1.5 *Input Pipeline*

So that the operation model of the TensorFlow input pipeline can be better understood, Figure 3.2 presents a high-level example of an input pipeline. In this example, each element is a TFRecord composed of 2 records, and each record corresponds to an image.

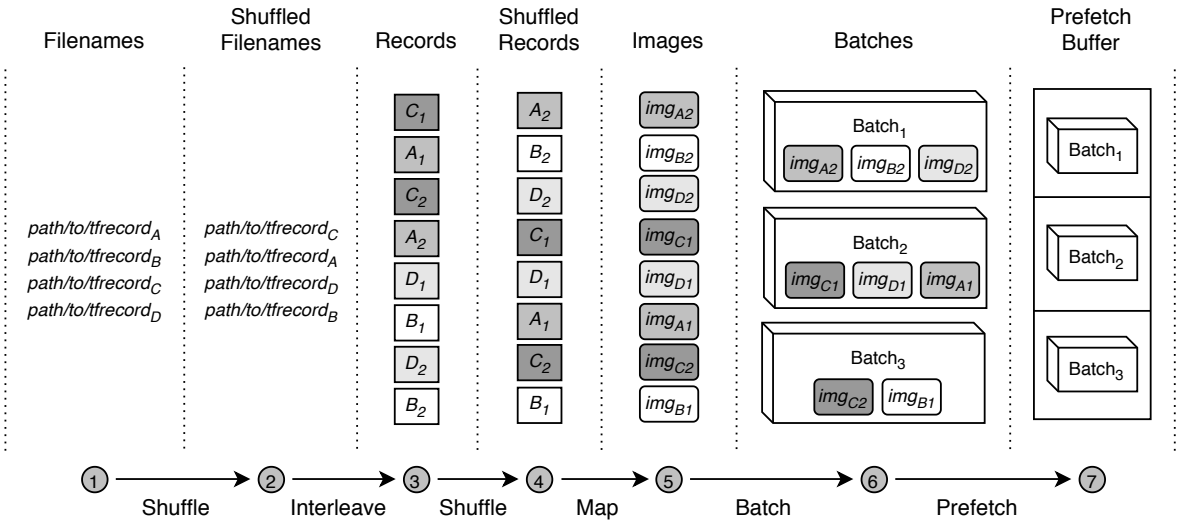


Figure 3.2: TensorFlow input pipeline example.

Apart from `prefetch`, `interleave`, and `map`, TensorFlow input pipeline also provides the `shuffle` [119] and `batch` [114] transformations. As the name implies, the `shuffle` transformation shuffles the dataset elements, and the `batch` transformation groups the elements into batches of a certain size. Briefly, the stages represented in the example are the following:

1. The input pipeline receives a list of filenames corresponding to the TFRecords containing the training samples.
2. The filenames are then *shuffled*.
3. The TFRecords are read and its internal records are interleaved with a `block_size` of 1 and a `cycle_length` of 2.
4. The records are *shuffled*.
5. The records are *preprocessed* (i.e., decoded, cropped, flipped, and resized) using the `map` transformation, resulting in the respective images (and labels, although not represented in the example).
6. The records are grouped into batches with a size of 3 (except for the last batch, which stores the remaining elements [114]).
7. Finally, the batches are *prefetched* and stored in a prefetch buffer.

Since `prefetch` is the last transformation of the example input pipeline, after being prefetched, the batches are consumed by the GPU.

For simplicity, the `repeat` [118] transformation was not represented in the example input pipeline. This transformation is usually placed after record shuffling, in order to repeat the dataset for the number of training epochs. The input pipeline presented in Figure 3.2 resembles the one employed in the TensorFlow *Custom Training Loop (CTL)* implementation for ResNet-50 [95].

3.2 EVALUATION

To understand the impact of I/O optimizations in local DL training, we conducted a thorough experimental evaluation of the TensorFlow framework. In this section we will present the methodology used to perform the experiments as well as discuss the obtained results.

3.2.1 *Experimental setup*

All experiments were carried out on a single compute node of the ABCI supercomputer. The ABCI supercomputer, hosted and managed by the AIST research center, is designed upon the convergence between AI and HPC workloads. Table 1 depicts the hardware and software specification of the compute node.

The evaluation was performed using the ILSVRC2012 [98] dataset, which is composed of approximately 144 GB of images. To ensure that the entire dataset would not fit in memory, we used the Linux Control Groups (cgroups [16]) to limit the Python process that executed TensorFlow to 64 GiB of memory. Additionally, the overall system resources and GPU utilization were observed using dstat [31] and nvidia-smi [77], respectively.

3.2.2 *Models*

According to Sarkar [101], the LeNet [59] and AlexNet [53] networks have high demands in terms of read throughput, while ResNet-50 [38] requires an intensive computing power. These networks have

Table 1: Specifications of the evaluation environment.

Item	Description
CPU	2× Intel Xeon Gold 6148 Processor 2.4 GHz, 20 Cores (40 Threads)
GPU	4× NVIDIA V100 for NVLink 16 GiB HBM2
Memory	384 GiB (12× 32 GiB) DDR4 2666 MHz RDIMM
Disk	Intel SSD DC P4600 1.6 TB u.2
Kernel	version 3.10
OS	CentOS 7.5
File system	XFS
TensorFlow	version 2.1.0
CUDA Toolkit	version 10.1.243
Python	version 3.6.5
GCC	version 7.4.0

different levels of complexity, considering their number of layers and parameters. Although ResNet-50 has more layers, AlexNet is the one with the most parameters among the three networks. By contrast, LeNet is the least complex network, with fewer layers and parameters. As such, these three models were used to perform the evaluation, with LeNet and AlexNet providing an I/O intensive workload (i.e., I/O-bound), and ResNet-50 a compute intensive workload (i.e., compute-bound).

3.2.3 *Dataset*

The models were trained with the ILSVRC2012 dataset, which is a subset of ImageNet [30] that includes 1.28 million images (approximately 138 GB) for training and 50 thousand images (approximately 6 GB) for validation, distributed across 1000 classes. On average, each ILSVRC2012 image contains 115 KB.

The evaluation was conducted using TensorFlow CTL implementation for ResNet-50, which at the time of the evaluation required the dataset to be converted to the TFRecord format. After the conversion, the dataset is composed of 1152 TFRecords (1024 for training and 128 for validation), each with an average size of 135 MB, and has a total size of approximately 144 GB. During the experiments, the dataset was stored in the local NVMe SSD disk of the compute node.

It is important to mention that LeNet was designed to recognize high-dimensional patterns with minimal preprocessing, such as handwritten characters [59]. Consequently, to perform this study, the LeNet network had to be adapted so that it could be trained with the ILSVRC2012 dataset. The modifications involved using RGB images instead of grayscale, and changing the dimension of the input samples from 32×32 to 224×224 pixels.

3.2.4 *TensorFlow Input Pipeline*

Among others, the input pipeline used in TensorFlow CTL implementation for ResNet-50 executes the following transformations:

```
# Fetch TFRecords
dataset = dataset.interleave(tf.data.TFRecordDataset,
                             cycle_length=10,
                             num_parallel_calls=tf.data.experimental.AUTOTUNE)

# Preprocess samples
dataset = dataset.map(lambda value: preprocess(value),
                      num_parallel_calls=tf.data.experimental.AUTOTUNE)
```

Listing 3.1: TensorFlow input pipeline transformations for ResNet-50.

Both transformations have its *autotune* feature enabled (Section 3.1.4). In the case of `interleave`, this means that *parallel interleave* is conducted. Considering that *parallel interleave* performs both parallel I/O and data prefetching, this represents the I/O optimized version of the TensorFlow input pipeline used in the experiments, which will be referred to as *TF optimized*. In contrast, a baseline version that performs sequential I/O and no data prefetching was also evaluated. To ensure this, we set the `interleave` `num_parallel_calls` argument to `None`. The baseline version of the input pipeline will be referred to as *TF baseline*.

The `prefetch` transformation was disabled in all experiments, since it caused TensorFlow to be silently killed after a few number of epochs, due to the cgroups memory restriction.

3.2.5 *Results*

For all experiments, we analyzed the impact of the I/O optimizations on the model training time and overall resource usage. Moreover, we also inspected the number of concurrent reads performed by *parallel interleave*. The experiments performed consisted of training LeNet, AlexNet and ResNet-50 on ILSVRC2012 converted to TFRecords. The train was conducted for 10 epochs, with a batch size of 256. To train with the 4 GPUs available on the compute node, `tf.distribute.MirroredStrategy` [71] was used. As this strategy divides the batch across all GPUs, each GPU trained with a batch size of 64. For each experiment were performed 5 runs. Reported results represent the average and standard deviation of each metric. It is important to mention that the CPU usage presented in this section corresponds to the sum of the user-level (application) and the system-level (kernel) usage.

Training Time

Figure 3.3 depicts the training time of the *TF baseline* and *TF optimized* versions of the input pipeline, for each training model. According to the results obtained, using the *TF optimized* version of the input pipeline has a positive impact on all I/O intensive models, in terms of training time. Since ResNet-50 is a compute intensive model, during its training the GPU computation is the bottleneck.

Consequently, ResNet-50 practically does not take advantage of the optimized I/O performance. On the other hand, under the *TF optimized* version, LeNet and AlexNet improve their training time by 67% and 62%, respectively, when compared to the *TF baseline* version. This is due to the fact that they are I/O-bound models, therefore decreasing data reading time improves the overall training performance.

The obtained results prove that local DL training with I/O intensive models can, in fact, benefit from I/O optimization.

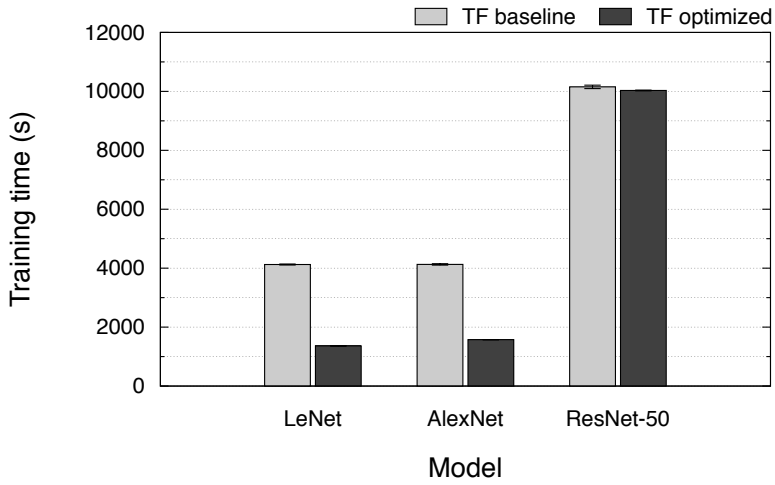


Figure 3.3: Training time of the LeNet, AlexNet, and ResNet-50 models under the baseline and optimized versions of the TensorFlow input pipeline.

Resource Usage

To evaluate the impact of parallel I/O and data prefetching on resource usage, we measured the average disk read throughput and CPU, GPU, and memory utilization for all testing scenarios. Disk write throughput was not included, since the only write operations performed by TensorFlow are related to the fault tolerance checkpointing mechanism, which is not affected by the I/O optimizations that are being studied. Figures 3.4 to 3.7 depict the resource usage of both TensorFlow baseline and optimized versions, for the LeNet, AlexNet, and ResNet-50 models.

As it would be expected, the I/O optimizations significantly increase the read throughput and GPU usage of the I/O-bound models, contrary to what happens with ResNet-50. This increase on the throughput and GPU usage reflects in a decrease of training time, justifying the results presented in Figure 3.3.

In contrast, optimizing I/O has the cost of increasing memory and CPU usage, having a maximum impact of approximately 5 GiB and 35%, respectively, with I/O-bound models. According to Figure 3.7, compute-bound models do not suffer a significant increase on CPU usage, when employing I/O optimization, whereas I/O-bound models do. This increase is caused by the number of threads fetching data from storage and serving the GPU, used for each model.

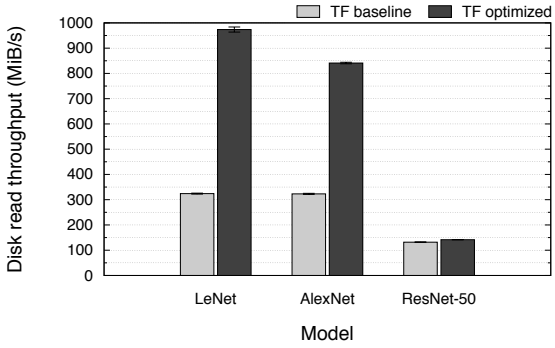


Figure 3.4: Average disk read throughput for TensorFlow with LeNet, AlexNet and ResNet-50.

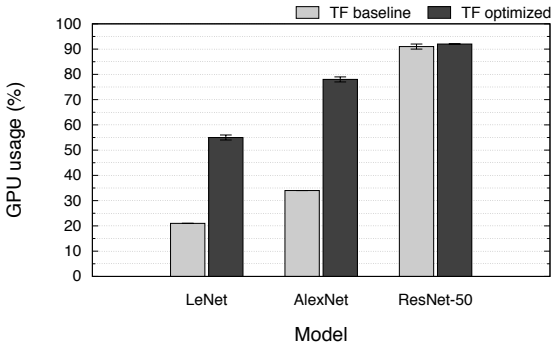


Figure 3.5: Average GPU usage for TensorFlow with LeNet, AlexNet and ResNet-50.

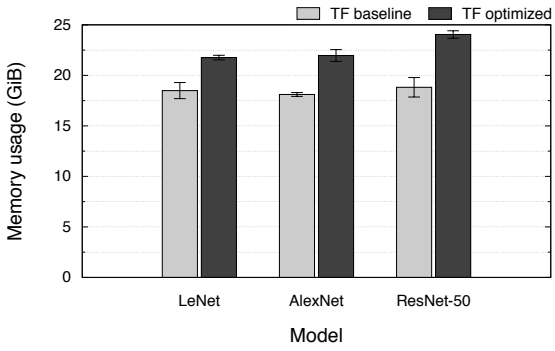


Figure 3.6: Average memory usage for TensorFlow with LeNet, AlexNet and ResNet-50.

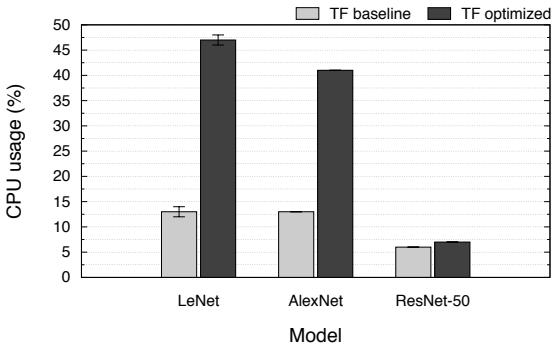


Figure 3.7: Average CPU usage for TensorFlow with LeNet, AlexNet and ResNet-50.

Concurrent Reads

Based on Equation 3, the number of concurrent read operations performed by *TF optimized* should not exceed 30, since *autotune* is enabled and `cycle_length` was set to 10 (Listing 3.1). Nevertheless, to better understand how the *autotune* feature operates, the number of concurrent threads used along the training process was evaluated.

To determine the periods of time in which each thread was reading data, we extended TensorFlow *Portable Operating System Interface (POSIX)* file system implementation to collect the starting (*start_ts*) and ending (*end_ts*) timestamps of each read operation sent to the file system. After having the timestamps, the following steps were performed:

1. Create a list with all the *execution periods* (i.e., pairs <*start_ts*, *end_ts*>).
2. Find the *points of intersection* of all *execution periods*. The *points of intersection* correspond essentially to the *start_ts* and *end_ts* timestamps of each *execution period*, since these are the points where the number of concurrent threads might change. Therefore, the collection of all intersection points is an ordered set (without repeated values) of all timestamps.
3. Create a list of *concurrency intervals* by pairing all the *points of intersection* found.
4. Validate how many *execution periods* overlap within each concurrency interval.

5. Sum the elapsed time of all *concurrency intervals* with the same number of concurrent threads.
6. Calculate the time percentage based on the total elapsed time of each number of concurrent threads and the elapsed time between the first and last timestamps.

Figure 3.8 depicts this process, demonstrating how the number of concurrent threads, on TensorFlow, was captured, and clarifies the concepts of *execution period*, *intersection point*, and *concurrency interval*.

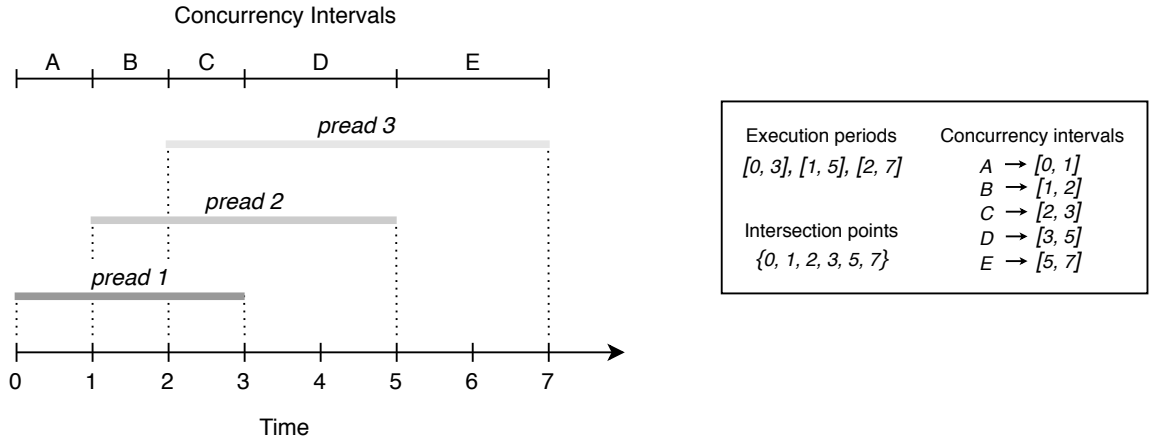


Figure 3.8: Strategy used to obtain the number of concurrent threads executed by TensorFlow parallel interleave transformation.

According to this strategy, for each *concurrency interval*, all *execution periods* had to be covered. Given that for each training run there were more than 11 million *concurrency intervals* and 5 million *execution periods*, this process is quite time consuming. As such, we developed a simple log analyzer program (in C++) to parallelize the execution and perform the concurrency analysis as efficient as possible.

Figure 3.9 depicts a *Cumulative Distribution Function (CDF)* with the percentage of time spent by N threads on actively reading data concurrently from the backend storage, where $0 \leq N \leq 30$. The results are represented for each training model, namely LeNet, AlexNet, and ResNet-50. The maximum number of concurrent threads reached by LeNet, AlexNet and ResNet-50 was 29, 30 and 28, respectively, which confirms the veracity of Equation 3.

Based on Figure 3.9, during most of the training, the *autotune* feature of *parallel interleave* uses between 4 and 8 threads for I/O intensive models, namely 55% for LeNet and 46% for AlexNet. On the other hand, under the ResNet-50 model, *parallel interleave* uses 0 threads (i.e., is not reading data) for almost 90% of the training time. This means that when training ResNet-50, the I/O threads are idle most of the time, proving once again that this is a compute intensive model. Although LeNet and AlexNet have a high percentage of active threads (concurrent or not), there is still 10% and 20%, respectively, of idle time. These percentages are relative to time spent performing computation and processing upstream input pipeline tasks.

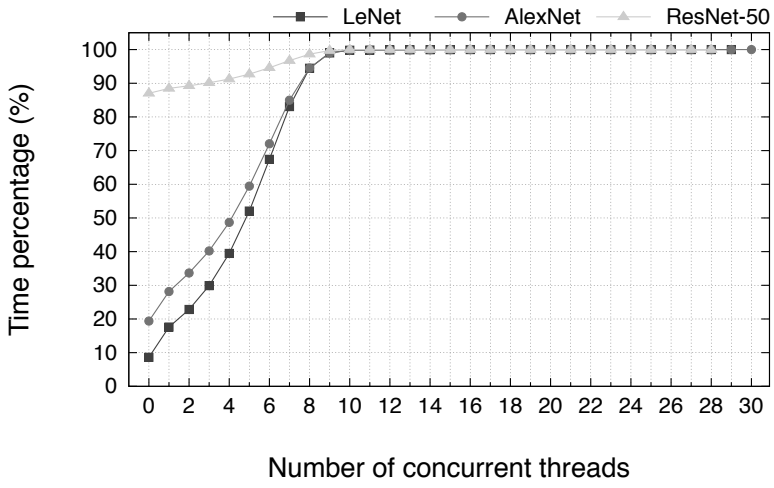


Figure 3.9: Time percentage of each number of TensorFlow concurrent threads.

Discussion

To understand if local DL applications can benefit from I/O optimization, two different versions of TensorFlow input pipeline were evaluated. The *TF baseline* version performs sequential I/O, while the *TF optimized* version performs data prefetching and parallel I/O. Although ResNet-50 maintains a very similar training time with both TensorFlow versions, *TF optimized* improves the training time of LeNet and AlexNet by 67% and 62%, respectively. This proves that local DL applications benefit from I/O optimization. In contrast, parallel I/O and data prefetching have an impact on resource usage, possibly increasing the memory consumption by 5 GiB and the CPU utilization by 35% (under the evaluated scenarios). The number of concurrent threads used by the *autotune* feature was also studied, indicating that with all three models, for the most part of the training process are used between 4 and 8 concurrent threads. In addition, the values rarely reach the maximum number of concurrent threads defined by Equation 3. Increasing resource usage is beneficial when it represents a significant improvement in performance, i.e., a decrease in training time. On the other hand, in a scenario where the hardware is being shared by multiple applications, it is crucial that the resource usage performed by one application does not impair the performance of others. In this case, *TF optimized* achieves considerably lower training times than *TF baseline*, justifying the increased use of resources.

Although the data prefetching and parallel I/O optimizations are effective, they cannot be applied to other DL frameworks, due to the fact that they are intrinsic to TensorFlow. Given that the workflow of these I/O optimizations does not depend on the TensorFlow internal operation model, they can be applied in other scenarios. Based on this, it would be beneficial to decouple these optimizations from TensorFlow and implement them on a framework-agnostic middleware, that can be applied to any DL framework. With this strategy, the middleware would be the one responsible for optimizing I/O, making it possible for the framework to focus entirely on its purpose of optimizing DL techniques.

Despite that the number of TensorFlow concurrent threads barely reached the maximum value during training, the TensorFlow *autotune* feature is greedy, since it allocates the maximum number of

threads at the beginning of the training process. This causes the CPU usage to be higher than needed, causing resources to be wasted. A common use case conducted on ABCI consists of running multiple TensorFlow applications on the same compute node. In this scenario, TensorFlow performance would possibly take advantage from using the lower number of threads possible, preventing the compute node from saturating as easily. That said, an important feature that should also be provided by the framework-agnostic middleware consists of an automatic provisioning mechanism that finds the lowest possible resource utilization, while ensuring an optimal I/O performance.

4

PRISMA

As previously stated, existing DL frameworks comprise internal I/O optimizations that cannot be decoupled and applied to other frameworks. Moreover, most I/O optimizations have to be manually configured, which requires additional time to understand each configuration parameter (e.g., number of parallel calls, prefetch factor) and to find the optimal configuration. Certain frameworks also encompass greedy provisioning algorithms, that allocate more resources than necessary. Therefore, DL frameworks could benefit from a framework-agnostic solution that is capable of automatically selecting the optimal configuration. Furthermore, the solution must not interfere with the model accuracy. To address these issues, this dissertation proposes PRISMA, a *Prefetching Storage Middleware for Accelerating Deep Learning Frameworks*. PRISMA was designed considering four main principles:

GENERIC PRISMA is a *generic* middleware, meaning that it can be applied to any DL framework or data format, preventing the development of redundant code.

SIMPLE PRISMA does not imply a steep learning curve and can easily be extended and adapted for different scenarios, which makes it a *simple* solution.

LIGHTWEIGHT PRISMA is a *lightweight* solution that aims at optimizing performance while spending as few computational resources as possible, providing additional resources for other applications that may be running on the same compute node.

TRANSPARENT PRISMA does not affect the accuracy of the model, since it does not impair the randomization level of the data reading operations, making it a *transparent* solution.

Throughout this section we provide a thorough description of PRISMA's architecture, describe the modules and workflow of PRISMA and present the implementation details of a PRISMA prototype.

4.1 ARCHITECTURE OVERVIEW

Parallel I/O and data prefetching demonstrated to improve the training performance not only of TensorFlow, but also of other DL solutions previously described in Section 2.2.2. Based on these insights, PRISMA was designed to provide both *parallel I/O* and *data prefetching* in order to optimize the I/O performance of DL applications.

The main purpose of PRISMA is to bring data to memory before the DL framework requests it. However, PRISMA is exclusively effective in scenarios where the entire dataset does not fit in memory, since otherwise the OS stores the dataset in the page cache after reading it the first time. Given that DL frameworks read shuffled data, the order in which files are accessed varies from epoch to epoch.

To take full advantage of PRISMA, each time the framework requests a file, it should already be in PRISMA internal buffer. For this to be possible, PRISMA has to know in advance the order in which the framework will read the files. More details about how this is accomplished will be provided in Section 4.2.1.

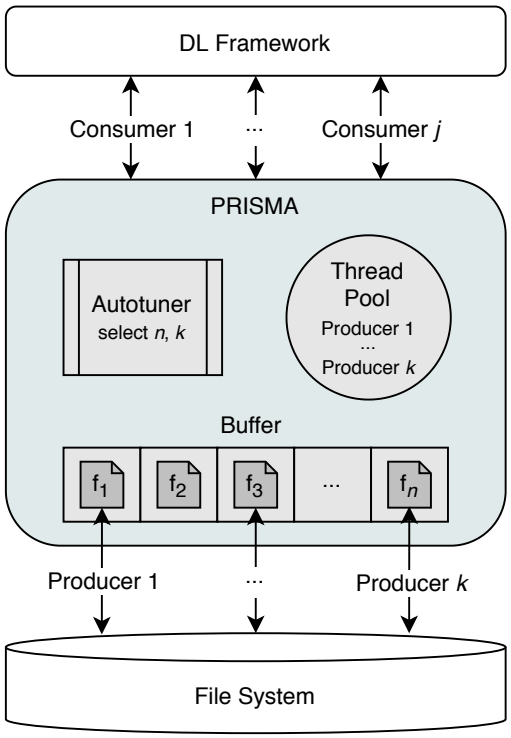


Figure 4.1: PRISMA high-level architecture.

PRISMA was designed as a storage middleware that sits between the DL framework and the file system. Figure 4.1 presents a high-level architecture of the PRISMA middleware. PRISMA receives requests from the DL framework and reads data from the backend storage. In other words, PRISMA intercepts read operations, and provides prefetched data to the DL framework. Therefore, instead of requesting data directly from the backend storage, the DL framework reads data from PRISMA. Each framework spawns one or more threads/processes to read the training data, which are called *consumers*. The total number of consumers, represented as j in the figure, is selected by the framework itself, and PRISMA has no access to this value. It is important to clarify that PRISMA does not have any control over the consumer threads.

PRISMA is composed of three different modules:

AUTOTUNER Instead of delegating to the user the responsibility of finding the optimal combination of maximum number of threads (k) and maximum buffer size (n) to use, PRISMA comprises an *autotuner* module. The autotuner selects k and n automatically to provide a good trade-off between performance and resource usage (i.e., preventing waste of resources). For this to be possible, the autotuner relies on an autotuning algorithm.

THREAD POOL PRISMA performs parallel I/O using multiple threads, called *producers*, that read data simultaneously. The producer threads are orchestrated by a *thread pool* module, which also manages the producers task queue. More specifically, the producers task queue stores the read operations to be executed by the producer threads. PRISMA supports a maximum of k threads executing concurrently, which can be configured by the user or selected by the autotuner.

BUFFER Each producer prefetches a training file (e.g., TFRecord, image) and stores it in an in-memory *buffer*. The buffer has a maximum number of files it can store, represented in the figure by n. Similarly to k, n can also be set by the user or selected by the autotuner.

Since the workload of DL frameworks is read-oriented, PRISMA only targets read operations, therefore it provides an interface with a single method, called `read()`. Therefore, whenever the framework needs to read data, it should simply invoke PRISMA `read()` method. Nevertheless, PRISMA design principles could also be employed to address write operations.

Fundamentally, PRISMA I/O flow consists of the following steps:

1. The producer threads continuously prefetch data from the backend storage and store the files in the internal buffer, so that the DL framework always has data available to read.
2. Each consumer thread (e.g., TensorFlow thread) accesses PRISMA internal buffer, using the `read()` method, reads a prefetched file and removes it from the buffer (in case no other consumer thread is expected to read that file soon, as described in Section 4.2). The frequency with which this step is performed depends on the internal operation model of the DL framework.
3. When the buffer reaches its maximum capacity, the producer threads temporarily block waiting for free space in the buffer.
4. The consumer threads also block temporarily when the files they are looking for are not yet in the buffer.

In addition to these steps, the buffer size and number of producer threads are periodically tuned by the autotuner, until an optimal configuration is found.

4.2 Module Design and Workflow

PRISMA provides an API that is merely composed by the `read()` method, as presented in Listing 4.1.

```
ssize_t read(const std::string& fn, char* res, size_t n, uint64_t offset);
```

Listing 4.1: PRISMA API.

For PRISMA to be seamlessly adopted, the signature of PRISMA `read()` method resembles that of the `pread()` system call. As such, the method reads `n` bytes from a training file with the filename `fn`, starting at `offset`, and stores the result in the memory address pointed by `res`. Additionally, the method returns the number of bytes read, with the return of zero indicating end of file. We decided

to use the filename as an argument instead of the file descriptor to simplify the usage of the method, avoiding having to invoke other methods to obtain the file descriptor.

To perform parallel I/O, we opted for a multi-threaded approach, instead of multi-process, since, contrary to processes, threads share memory. So that the producer threads can be reused between read operations, PRISMA comprises a thread pool. The size of the thread pool can be automatically tuned during training, using PRISMA autotuning mechanism. All the read operations that need to be executed by the producer threads are stored in a queue that is internal to the thread pool. Furthermore, PRISMA reads data from the backend storage using the `pread()` system call. Each `pread()` operation performed by PRISMA uses a fixed `read_block_size`, which can be set on the configurations.

Apart from the thread pool, PRISMA contains an in-memory *buffer* that stores prefetched data, mapping the filenames to the file content (Section 4.3). The caching policy used by PRISMA is quite straightforward: a file is stored in the buffer as soon as it is read by a producer thread, and is then evicted when a consumer thread fetches that same file. It can happen that when a producer initiates a read operation for a specific file, that file is still in the buffer because it was prefetched for the previous epoch and has not yet been consumed. To avoid reading files unnecessarily, each file has a variable associated with it, called `n_reads`, that specifies the number of times the file should be read before it is removed from the buffer. When a consumer thread finishes reading a file, its `n_reads` variable is decremented. After the decrement, if the value is equal to zero, then the file is removed from the buffer, otherwise it means that there are consumers expected to read the file soon, so it cannot be removed from the buffer. In the same way that the size of the thread pool (i.e., number of producer threads) can be automatically tuned during training, the maximum capacity of the internal buffer (i.e., buffer size), can also be tuned using the autotuning mechanism.

The autotuning mechanism provided by PRISMA is responsible for automatically tuning the number of producer threads and the buffer size. This mechanism is periodically performed by a background thread, called *autotuner*, that resorts to an algorithm in order to find the optimal combination of both parameters. After the optimal configuration is found, the autotuning is disabled, due to the fact that the workload remains considerably stable throughout the training process.

Figure 4.2 illustrates the workflow and interactions between PRISMA, the DL framework and the file system.

4.2.1 *Prefetch Order*

As previously stated, PRISMA has to know in advance the order in which the framework will read the training files. Typically, the DL framework uses a list containing the filename of each training file, which defines the order in which the files will be read in each epoch. The list of filenames must be shuffled for each epoch, to avoid overfitting. If this shuffling process happens inside the framework, then PRISMA will be dependent on the workflow of the framework to obtain the read order. For example, if the framework only shuffled the filenames right before each epoch starts, then PRISMA would have to wait for the beginning of the epoch to prefetch the files (for that epoch). This would impair the performance of PRISMA since initially the files that the framework would try to read would not be in memory. Moreover, while waiting to obtain the read order for the next epoch, PRISMA would probably be paused wasting time, instead of prefetching data. To prevent this, and to

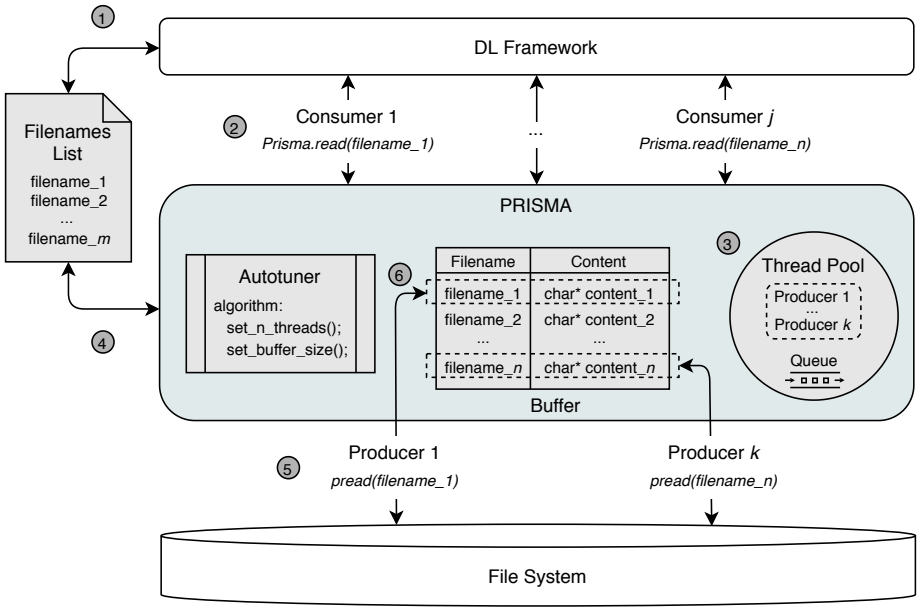


Figure 4.2: PRISMA workflow and interactions.

ensure that PRISMA delivers the best possible I/O performance, the shuffling of the filenames must be performed outside the DL framework. For this to be possible, a list with the shuffled filenames has to be passed to the framework, so it knows the order in which the files must be read.

As shown in Figure 4.2, both PRISMA and the DL framework share a *filenames list*. This list dictates the order in which both the *consumers* and *producers* should read the training files. Before training begins, the DL framework must access the *filenames list* to obtain the list of files in the correct order (event (1)). It is worth mentioning that if the DL framework is using multiple threads/processes to perform I/O, then the files may be read in a slightly different order than the one stipulated in the *filenames list*. This can happen due to the fact that some threads/processes execute the read requests faster than others.

Since all the filenames presented in the *filenames list* are globally shuffled and repeated for each epoch, this method does not affect the accuracy of the training model.

4.2.2 *Initialization*

The first time that the DL framework calls `Prisma.read()`, PRISMA is initialized. The initialization process consists of the following steps:

1. Read configuration file (Section 4.2.4).
2. Create thread pool (Section 4.2.3).
3. Launch *autotuner* thread (Section 4.2.5).
4. Launch profiler thread (Section 4.2.6).
5. Enqueue read operations (Section 4.2.3).

According to Figure 4.2, the initialization process starts after event (2), where *consumer* 1 calls `Prisma.read()` to read the file that is represented by *filename*_1.

4.2.3 *Data Prefetching and Parallel I/O*

PRISMA uses one or more threads to prefetch the files from the backend storage and store them in the buffer. In the initialization process a thread pool is created (event (3)) with one thread, in case `autotune` is enabled, or `n_threads` in case this value was manually set. Then PRISMA goes through the *filenames list* (event (4)) and for each filename in the *filenames list*, a read operation is added to the thread pool queue. The threads in the pool are responsible for executing the enqueued read operations (event (5)) and storing the files in the buffer (event (6)). Given this, at any given time there are at most `n_threads` reading data from the backend storage.

Every time a *consumer* wants to read a file, it simply checks if the file is already stored in the buffer. If it is, then the *consumer* will read it, otherwise the *consumer* will block waiting for the *producers* to prefetch that file. On the other hand, the *producers* will block waiting for free space in the buffer, whenever it is full. As *consumers* read the files, they remove them from the buffer, so that more space is available for other files to be prefetched. To wake up *consumers* that are blocked, whenever a *producer* thread stores a file in the buffer, it notifies the *consumers* that may be waiting for that file. Similarly, whenever a *consumer* thread removes a file from the buffer, it notifies the *producers* that may be blocked waiting for space to be released in the buffer.

Since different threads execute the operations at different rates, it could happen that files that appear later in the *filenames list* are stored first in the buffer. If this was the case, depending on the number of threads and buffer size, the buffer could get full with out of order files. This would cause the *consumer* threads to block waiting for recent files to be prefetched, and the *producers* to block waiting for free space in the buffer, causing a deadlock situation. To prevent this from happening, when the operations are stored in the queue, they are given an identifier that represents the order in which the respective file should be stored in the buffer. With this strategy, the *producer* threads may read the files at its own speed, however they can only store a file in the buffer after all the files with lower identifiers have already been buffered.

4.2.4 *Configuration Parameters*

To adapt PRISMA to different I/O workloads and scenarios, the following configuration parameters can be tuned:

BUFFER_SIZE Maximum number of elements that can be stored in the buffer; if set to `autotune` it will be automatically defined using the *autotuning* mechanism described in Section 4.2.5.

N_THREADS Number of threads in the thread pool (Section 4.2.3); if set to `autotune` it will be automatically defined using the *autotuning* mechanism described in Section 4.2.5.

READ_BLOCK_SIZE Read block size used by the `pread()` system call in bytes; the default value is 64 KB.

MAX_BUFFER_SIZE Maximum buffer size that can be set by the *autotuning* mechanism.

MAX_N_THREADS Maximum number of threads that can be set by the *autotuning* mechanism; the default value is equal to the number of concurrent threads supported by the hardware.

Apart from these parameters, the user can enable or disable the `debug` and `profiling` (Section 4.2.6) modes. When `debug` is enabled a log file is created with information regarding the files that are being read and the decisions that are being made by the *autotuning* mechanism.

When choosing the `buffer_size` (or `max_buffer_size` in case autotune is enabled) it is important to take into account the size of each dataset file, so that we can predict the memory that the buffer will consume. For instance, when training the model with the ILSVRC2012 dataset converted to TFRecords, each file is around 135 MB, however when using the original dataset with raw images each file has an average size of 115 KB (Section 3.2.3). Given this, when using TFRecords, a buffer of size 10 consumes approximately 1.4 GB of memory, while when using raw images, the same buffer size only consumes around 1.2 MB. This is also important in terms of performance, since an extremely small buffer causes threads to block continuously, adding overhead to the training process, whereas a large buffer wastes unnecessary resources.

4.2.5 *Autotuning Mechanism*

When the `buffer_size` and `n_threads` configurations are set to `autotune`, these values are automatically tuned using an *autotuning* algorithm. PRISMA uses a background thread (outside of the thread pool), called *autotuner*, that is responsible for periodically executing the *autotuning* algorithm. Algorithm 1 describes the *autotuning* algorithm used by PRISMA.

The *autotuning* algorithm starts with a minimum value for both parameters, represented by *init_buffer_size* and *init_n_threads* in Algorithm 1. Depending on the I/O workload (i.e., I/O intensity of the training model, dataset dimension, size of each training file), there are initial buffer sizes that are more efficient than others, in the sense that they allow the *autotuning* algorithm to converge faster to the optimal configuration. For instance, in a scenario where the dataset is composed of multiple small files, and the model is I/O-bound, it would be beneficial to start with a higher buffer size. On the other hand, with a compute intensive model, and a dataset with huge files, initiating the process with a large buffer could lead to waste of resources, since compute intensive models do not rely on I/O efficiency, not requiring much data to be prefetched. For this reason, *init_buffer_size* and *init_n_threads* should be set based on the I/O workload.

After being initialized, the buffer size and number of threads are increased based on the buffer usage (*buffer_usage*). The buffer usage represents the percentage of elements (*#elements*) in the buffer, compared to its maximum capacity (*buffer_size*), and can be obtained using Equation 4.

$$buffer\_usage = \frac{\#elements}{buffer\_size} \times 100 \tag{4}$$

Autotuning is only performed until an optimal configuration is found, due to the fact that the workload remains considerably stable throughout the training process. An optimal configuration is considered to be one that allows a relatively high buffer usage to be maintained, preventing *consumers* from blocking while waiting for data. So that the *autotuning* mechanism can be aware

Algorithm 1 PRISMA Autotuning Algorithm

Require:
$init\_buffer\_size > 0$
$init\_n\_threads > 0$
$buffer\_size\_step > 0$
$n\_threads\_step > 0$

1: $buffer\_size \leftarrow init\_buffer\_size$
2: $n\_threads \leftarrow init\_n\_threads$
3: $sleep(init\_interval)$
4: **while** *true* **do**
5: **if** $\#records \geq 10$ **then**
6: $avg\_buffer\_usage \leftarrow get\_avg\_buffer\_usage()$
7: $std\_dev \leftarrow get\_std\_dev()$
8: **if** $avg\_buffer\_usage \geq 80$ **and** $std\_dev \leq 20$ **then** ▷ Optimal condition
9: $disable\_autotuning()$
10: **else if** $avg\_buffer\_usage \leq 30$ **or** $std\_dev \leq 20$ **then**
11: $n\_threads \leftarrow n\_threads + n\_threads\_step$
12: **else**
13: $buffer\_size \leftarrow buffer\_size + buffer\_size\_step$
14: **end if**
15: **end if**
16: $sleep(loop\_interval)$
17: **end while**

of the impact of the configurations it selects, PRISMA collects information about the buffer usage, between each *autotuning* run. Before any decision is made, the average buffer usage and standard deviation are calculated using the information collected since the last *autotuning* execution (lines 6 and 7 of Algorithm 1). Then, depending on the obtained values, either the buffer size or the number of threads is increased. In this context, an optimal configuration is considered to be represented by an average buffer usage of at least 80%, with a standard deviation lower or equal to 20%. As such, this represents the optimal condition of the *autotuning* algorithm, specified in line 8 of Algorithm 1. When the optimal condition is reached, the *autotuning* algorithm is disabled (line 9 of Algorithm 1).

To reach an optimal configuration, the buffer size and number of threads are increased by a *buffer_size_step* (line 13 of Algorithm 1) and a *n_threads_step* (line 11 of Algorithm 1), respectively. Similarly to the initial parameter values, the efficiency of the increasing steps are also related to the I/O workload (i.e., size of each training file and dimension of the dataset). Although increasing the parameters at an exponential rate would allow to find the optimal configuration faster, this would converge to higher values, potentially resulting in overprovisioning. To prevent this, the *autotuning* algorithm increases each parameter linearly.

When the buffer is too small for the workload production and consumption rates, it is continuously reaching its maximum and minimum capacity. This causes *consumers* and *producers* to block several times, consequently adding overhead to the process. Given this, the buffer size is considered to be inadequate when the buffer usage demonstrates high variance values. In this context, a standard

deviation of more than 20% is considered to be high. Another issue is when the production rate cannot match the consumption demand, causing the *consumers* to block waiting for data. This reflects on a low average buffer usage, since the buffer is empty most of the time. To address this issue, the number of *producer* threads should be increased to prefetch data at a higher rate. In this context, a low buffer usage is assumed to be a value below 30%. In summary, line 10 of Algorithm 1 presents the necessary condition for the number of threads to be increased.

The period of time between each *autotuning* cycle, represented by *loop_interval* in Algorithm 1, should be brief enough to ensure that an optimal configuration is quickly found. On the other hand, decisions based on a limited number of values can mislead the algorithm. Once again, the efficiency of the autotuning interval depends on the I/O workload, in the sense that they allow more or less buffer usage records to be collected, allowing the *autotuning* algorithm to make decisions that are more or less reliable. Since the autotuning interval may not be sufficient to collect a significant amount of buffer usage records, the configuration is only tuned if at least 10 buffer usage records have been collected by PRISMA. If the number of collected records from one cycle to another is lower than 10, these are saved and passed on to the next cycles, until the necessary amount of records is reached.

As during the initialization of the framework, the consumption rate is inconsistent, before the *autotuning* process begins, a brief wait of *init_interval* milliseconds is performed to allow the system to stabilize (line 3 of Algorithm 1).

Whilst studying the best strategy for the *autotuning* algorithm, the number of *consumer* waits was also considered. Since an optimal scenario would be one where *consumers* never have to wait for data, the number of *consumer* waits could be an effective metric in which to base the *autotuning* decisions. However, as previously stated, when the DL framework uses multiple *consumer* threads/processes, the order in which the files are requested may not match the order dictated by the *filenames list*. Considering this, a high number of *consumer* waits does not exactly mean that the configuration used is inefficient, but that *consumers* are requesting files in a different order than they are being prefetched. For this reason, the number of *consumer* waits is not taken into account by the *autotuning* algorithm. On the other hand, the buffer usage was the metric that seemed to best reflect the effectiveness of the chosen configuration, therefore was the one used in the *autotuning* algorithm.

It is important to mention that *autotuning* can be performed exclusively for one of the parameters, `buffer_size` or `n_threads`. This means that, for example, if `n_threads` is set to 5 by the user, and `buffer_size` is set to `autotune`, then the *autotuning* mechanism will only be responsible for finding the optimal value for the `buffer_size` configuration.

4.2.6 *Profiling*

In cases where the user decides to manually set the `buffer_size`, the `n_threads`, or even the `read_block_size`, it can be challenging to choose the optimal values. To help the user understand the impact that the selected configuration has on PRISMA performance, PRISMA provides a *profiling* mode. Profiling is performed by a *profiler thread* (outside the thread pool) that collects information during training about the following metrics:

READ DURATION Time it takes to read a single training file in milliseconds.

#PRODUCER WAITS Number of times a *producer* was blocked waiting for free space in the buffer.

#CONSUMER WAITS Number of times a *consumer* was blocked waiting for a specific file to be prefetched.

When PRISMA finishes its execution, the *profiler thread* prints the average values of each of the collected metrics.

4.2.7 *Client-Server*

For a DL framework to use PRISMA, it simply needs to create an instance of PRISMA and then invoke the `read()` method every time it intends to read a file. When the DL framework uses processes instead of threads, each process will have its own instance and variables, due to the fact that processes are isolated (i.e., do not share memory). To overcome this issue, PRISMA also provides a *client-server* architecture, where each client sends the read request to the server, and the server forwards the request to PRISMA. To maintain the parallelism intended by the multi-process scenario, the server spawns a thread for each client. When using the *client-server* architecture, each process should instantiate the client, and invoke the client read method, which has the same signature as the one presented in Listing 4.1. Figure 4.3 presents the interactions between PRISMA and the DL framework with the *client-server* architecture, where the *consumers* invoke the `read()` method directly in a client instance.

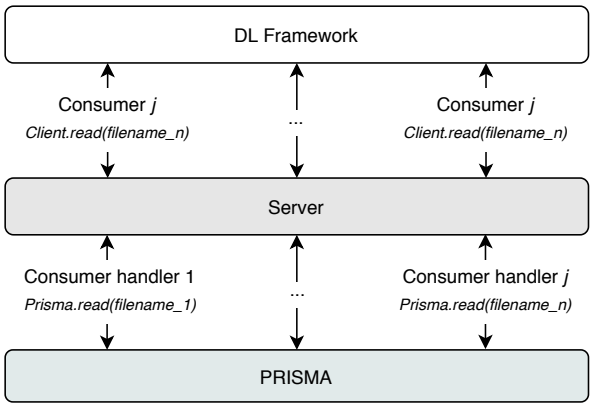


Figure 4.3: PRISMA client-server architecture.

The *client-server* architecture should exclusively be used in multi-process scenarios, since it adds communication overhead caused by including an intermediary between the DL framework and PRISMA. For single-process, single-threaded and multi-threaded applications, the `read()` method should be invoked directly in a PRISMA instance.

4.3 IMPLEMENTATION

We implemented a PRISMA prototype with approximately 1.7 thousand lines of C++ code. The thread pool used by PRISMA was developed using the CTPL library [26]. Additionally, since the DL framework may not read the training files exactly in the order dictated by the *filenames list*, the PRISMA internal buffer has to provide a way of quickly accessing the content associated with a specific filename. Thus, the buffer was implemented using a HashMap that maps the filename to the file content. Because existing structures of the C++ standard are not thread safe, and to allow multiple threads to concurrently read, write and erase different keys of the HashMap, the data structure used for the buffer was Intel *Threading Building Blocks (TBB)* Concurrent HashMap [24]. Intel TBB Concurrent HashMap maps keys to values in a way that allows multiple threads to concurrently access values via `find`, `insert` and `erase` methods, using implicit locks [126].

Additionally, we ensured concurrency control over *producers* and *consumers* accessing the same file and accessing the shared variables, thus avoiding race conditions and inconsistent state. Each HashMap entry was protected with a `mutex`, so that there would not be more than one *producer*/*consumer* trying to write/read the same file.

When it comes to PRISMA client-server implementation, the communication between the clients and the server is performed through UNIX Domain Sockets. Moreover, the configuration parameter `max_n_threads` is set by default to the number of concurrent threads supported by the hardware, using the C++ function `std::thread::hardware_concurrency()`.

The *filenames list* shared between PRISMA and the DL framework is a file composed of $\#files \times \#epochs$ filenames, that must be created before the training starts. In order to do this, we developed a simple Python module, called `shuffle_filenames`, that provides a `shuffle_filenames()` function. This function is responsible for creating the *filenames list* file by performing the following steps:

1. Receives a list with all the training filenames, called `filenames`.
2. Creates a new file called `filenames_list`.
3. Shuffles the `filenames` and writes them to the file.
4. Repeats step 2. for the number of training epochs.

PRISMA configuration parameters can be set using an INI configuration file. For simplicity reasons, and due to the fact that the relationship between the autotuning parameters and the I/O workload is not completely known, we decided to assign specific values to these parameters in the PRISMA prototype implementation. As such, the *init_buffer_size* and *init_n_threads* parameters were set to the minimum values of 10 and 1, respectively. Similarly, *buffer_size_step* was set to 10 and *n_threads_step* to 1. The autotuning interval was set to 100 milliseconds, a value based on the TensorFlow autotune feature, that uses an initial gap of 10 milliseconds between cycles (Section 3.1.4). Finally, the initial autotuning wait was set to 15 seconds to allow the DL framework consumption rate to stabilize.

4.3.1 *Integration with TensorFlow*

Given that one of the case studies of this dissertation consists of evaluating the PRISMA prototype performance when training DL models with TensorFlow, we integrated PRISMA with this framework. For this to be possible, TensorFlow required the following adjustments:

1. Use the `shuffle_filenames()` function that was previously described to shuffle the filenames and create the filenames list.
2. Invoke `Prisma.read()` instead of the default `pread()` system call.

To perform the shuffling process outside of TensorFlow, the dataset preprocessing code [41] of the TensorFlow CTL implementation for ResNet-50 [95] had to be extended. The modifications involved disabling the `shuffle` input pipeline transformation applied to the filenames and using a function called `get_shuffled_filenames()` instead. The function `get_shuffled_filenames()` provides a list with the exact content of the filenames list and writes it to a file. This is achieved by importing the Python module that was developed, and evoking the `shuffle_filenames()` function. Given that `shuffle_filenames()` already repeats the filenames for each epoch, the `repeat` input pipeline transformation was also disabled.

A new TensorFlow file system implementation, called `prisma`, had to be created to use `Prisma.read()` instead of the `pread()` system call. This implementation is practically the same as the POSIX file system, only with the following changes:

```
// Instantiate PRISMA
Prisma prisma = Prisma();

// Invoke PRISMA read() method
// prisma.read() replaces the pread() call
ssize_t r = prisma.read(filename_, dst, requested_read_length, offset);
```

Listing 4.2: Changes made to the TensorFlow POSIX implementation.

4.3.2 *Integration with PyTorch*

Similarly to TensorFlow, another case study of this dissertation consists of evaluating the PRISMA prototype performance when training DL models with PyTorch. Once again, for this to be possible, PRISMA had to be integrated with PyTorch.

Since PRISMA is written in C++, for it to be integrated with the Python interface of PyTorch, a Python binding of PRISMA had to be created using pybind11 [89].

So that PyTorch reads the files in the same order specified in the filenames list, a custom Dataset and Sampler were implemented. PyTorch uses indexes instead of filenames to determine the order in which the files will be read. Given this, first the custom Dataset creates an index dictionary that associates an index with a filename. Then, a function similar to `shuffle_filenames()`, called `shuffle_indexes()`, shuffles the indexes and uses the index dictionary to create the filenames list. This function is also provided by the `shuffle_filenames` Python module, and can be invoked after that same module is imported. The shuffled index list created with `shuffle_indexes()` is used by the custom Sampler to provide the Data Loader with an iterator for the shuffled indexes of a given epoch.

Considering that PyTorch uses processes instead of threads to parallelize I/O, the client-server implementation of PRISMA had to be employed. However, when no worker processes are used,

PRISMA standard implementation can still be applied. Thus, when the `num_workers` argument of the Data Loader is set to 0, a PRISMA instance is created in the custom Dataset and is later used by the main process. When multiple workers are used, a PRISMA client instance is created on each worker process using the `worker_init_fn` argument of the Data Loader. Each worker process uses the `Client.read()` method, instead of the common `Prisma.read()`. As specified in Section 4.2.7, the client instances send the read requests to the server, which forwards them to PRISMA.

To evaluate PyTorch with PRISMA, the experiments were performed using PyTorch ImageNet implementation [40]. According to this implementation, the Sampler iterator with the indexes for each epoch is only created at the beginning of the epoch. Consequently, PyTorch only starts prefetching the samples for an epoch when that epoch begins, contrary to what happens with PRISMA, that prefetches samples independently of the training epoch.

4.4 SUMMARY

PRISMA is a storage middleware that was designed to improve the I/O performance of local DL applications. Contrary to other solutions, PRISMA can be applied to any DL framework and uses as few computational resources as possible. The I/O optimizations employed by PRISMA are data prefetching and parallel I/O. To perform parallel I/O, PRISMA uses multiple threads to read data from the backend storage, called *producers*. At the same time, the DL framework uses multiple *consumer* threads to request data from PRISMA. Data prefetching is conducted by reading the training files before the DL framework requests them, storing the data in an internal buffer.

So that PRISMA can know in advance the order in which the framework will read the files, a *filenames list* is shared between the framework and PRISMA. While PRISMA reads data exactly in the order dictated by the *filenames list*, when using more than one *consumer*, the DL framework may request the files in a slightly different order.

PRISMA starts executing when the DL framework requests data for the first time. Before beginning to prefetch data, PRISMA reads a configuration file that allows to set the buffer size (`buffer_size`) and number of threads (`n_threads`) used. Both of these parameters can either be manually set by the user or automatically tuned by PRISMA, using an *autotuning* mechanism. The *autotuning* mechanism bases its decisions on the percentage of buffer that is being used, between each *autotuning* cycle.

Since several threads are continuously removing and inserting data in the buffer, concurrency control strategies, such as locking and concurrent data structures, had to be adopted.

When the user manually sets the `buffer_size` and `n_threads` configurations, he can use PRISMA *profiling* mode to help him determine if the chosen configuration is being effective or not. Moreover, PRISMA provides a *debug* mode, which creates a log with information about the read operations that are being performed and the decisions made by the *autotuning* algorithm.

There are DL frameworks, such as PyTorch, that adopt a multi-process scenario, instead of using multiple threads to perform I/O. Given that processes are isolated, if each process uses its own instance of PRISMA, it will have an independent buffer, *autotuner* thread, and other variables. To prevent this from happening, a *client-server* architecture is provided by PRISMA, that simply adds a server as an intermediate between the DL framework and the middleware.

Given that the case studies of this dissertation consist of evaluating the impact of PRISMA on TensorFlow and PyTorch, the PRISMA prototype was integrated with both of these DL frameworks.

5

CASE STUDIES AND EXPERIMENTAL EVALUATION

We integrated the PRISMA prototype with two popular DL frameworks, namely TensorFlow and PyTorch. In this chapter, we perform a thorough experimental evaluation of the PRISMA prototype and compare its benefits with TensorFlow and PyTorch, under multiple workloads. The conducted experiments demonstrate the impact that PRISMA introduces in both frameworks, not only in terms of performance but also on the utilization of computational resources.

5.1 TENSORFLOW

We now discuss the experimental evaluation of the TensorFlow framework. We first describe the experimental setup and methodology used, and then depict and analyze the results achieved under different scenarios.

5.1.1 *Methodology and Experimental Setup*

All experiments were conducted on a single compute node of the ABCI supercomputer. Both hardware and software specifications were already identified in Table 1.

To ensure that the entire dataset would not fit in memory, we used cgroups to limit the Python process that executed TensorFlow to 64 GiB of memory. The overall system resources and GPU utilization were observed using dstat and nvidia-smi, respectively.

MODELS To provide a fair and comprehensive evaluation of PRISMA, we followed a methodology similar to the one presented in Chapter 3. All experiments were conducted using the LeNet, AlexNet and ResNet-50 models. Again, these models were selected due to the fact that they present different I/O workloads. While LeNet and AlexNet are I/O intensive, ResNet-50 is a compute-bound model.

To perform these experiments, the LeNet network used was the one that had to be adapted for the preliminary studies (Chapter 3).

DATASET All models were trained using the ILSVRC2012 dataset, which is the same described in Section 3.2.3. Similarly to what happened in the preliminary studies, the dataset was converted to TFRecords instead of using raw images. During the experiments, the dataset was stored in the local NVMe SSD disk of the compute node.

TENSORFLOW INPUT PIPELINE The input pipeline used in this experiment was the same as in Section 3.2.4, which employs *parallel interleave* to fetch TFRecords and *parallel map* to preprocess the training samples. PRISMA was applied to the *TF baseline* version of TensorFlow, so that its data prefetching and parallel I/O features could be compared to the ones provided by *TF optimized*. Since *TF baseline* performs sequential I/O, in the experiments there is only one consumer thread requesting data from PRISMA.

PRISMA was compared to the *TF baseline* and *TF optimized* versions of TensorFlow previously analyzed in Section 3.2. The experiments consisted in training the three models for 10 epochs, with a variable batch size, namely 64, 128, 256. In addition, the training was distributed across the 4 GPUs of the compute node, using `tf.distribute.MirroredStrategy` [71]. Unless otherwise stated, for each experiment were performed 5 runs and the average values of each metric were measured. PRISMA `max_buffer_size` and `max_n_threads` configurations were set to 60 and 80 (value returned by the C++ function `std::thread::hardware_concurrency()`), respectively, during the experiments.

Although validation is performed after each epoch, TensorFlow does not read the validation files right after it finishes reading the training files of each epoch. Given this, as it was not possible to predict when the validation files were going to be read, PRISMA was only used to optimize the reading performance of the training files. Consequently, the filenames list shared by TensorFlow and PRISMA did not contain any validation files. While the *TF optimized* version used parallel I/O to prefetch both training and validation files, when using PRISMA the validation files were read sequentially with no prefetching.

5.1.2 *Results*

In the upcoming sections we provide an analysis regarding the performance and resource usage of TensorFlow and PRISMA. Moreover, we compare the autotuning mechanisms of TensorFlow and PRISMA and present the accuracy achieved for each model.

Training Time

The training time of all models was measured for a batch size of 64, 128, and 256. The `tf.distribute.MirroredStrategy` performs data parallelism, causing the batch size to be distributed between all participating GPUs. This requires all GPUs to synchronize at the end of each iteration, in order to estimate the average gradient (Section 2.5). Increasing the batch size reduces the number of iterations, consequently decreasing the number of times that the GPUs have to synchronize. As such, a larger batch size translates into a lower communication overhead between GPUs, therefore accelerating computation. Given this, when increasing the batch size, it is expected to decrease the execution time [45, 17].

Figures 5.1 to 5.3 present the training time of the LeNet, AlexNet and ResNet-50 models, using TensorFlow and PRISMA. With *TF baseline* the training time remains practically constant with the LeNet and AlexNet models, regardless of the batch size. This is caused by the fact that, for I/O intensive models, the overall training time is limited by the I/O performance provided by the system. As such, since with *TF baseline* I/O is the bottleneck, enhancing the computing performance (i.e., increasing the batch size) does not impact the training time.

According to the results obtained, PRISMA has a positive effect on the overall training time of both I/O intensive models, namely LeNet and AlexNet, for all batch sizes. Similarly to what happens with the *TF optimized* version, PRISMA does not have an impact on the training time of ResNet-50, due to the fact that this is a compute intensive model. Contrary to *TF baseline*, the I/O performance provided by PRISMA and *TF optimized* causes the training performance to improve with a larger batch size. This proves that optimizing I/O, allows the performance of I/O-bound models to scale with the batch size.

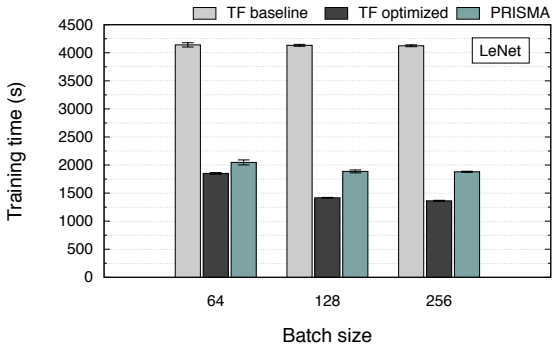


Figure 5.1: Average training time of PRISMA and TensorFlow with LeNet.

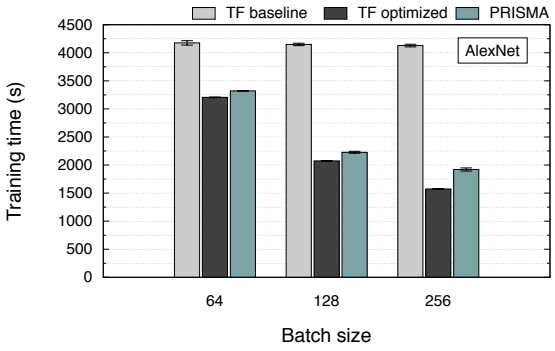


Figure 5.2: Average training time of PRISMA and TensorFlow with AlexNet.

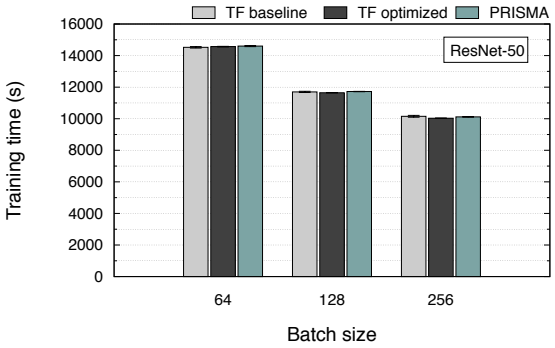


Figure 5.3: Average training time of PRISMA and TensorFlow with ResNet-50.

When compared to *TF baseline*, both PRISMA and *TF optimized* are able to improve the training time by more than 50% and 20%, for LeNet and AlexNet, respectively. This is due to the fact that *TF baseline* performs sequential I/O and does not prefetch data.

Interestingly, with a batch size of 64, PRISMA achieves a performance similar to *TF optimized*. When we increase the batch size, *TF optimized* scales better than PRISMA. This difference can be observed mainly with LeNet, where while PRISMA improves the training time by 54%, *TF optimized* manages to achieve an improvement of 67% when using a batch size of 256. Since the validation files represent approximately 11% of the dataset, the performance discrepancy between PRISMA and *TF optimized* is mainly caused by the fact that PRISMA does not optimize the I/O of the validation files, whereas *TF optimized* does.

Resource Usage

The histogram plots provided in this section present the average usage of a given resource, based on the 5 runs that were performed for each batch size. The time series show the resource usage over time, and represent a single run with a batch size of 256. The time series relative to the resource usage with a batch size of 64 and 128 can be found in Section A.1.1 of the Appendix.

DISK READ THROUGHPUT The first resource usage analysis to be performed concerns disk read throughput. Figures 5.4 to 5.6 represent the average read throughput of all models. *TF baseline* reaches an average read throughput of approximately 315 MiB/s with the I/O intensive models (i.e., LeNet and AlexNet), regardless of the batch size. With ResNet-50, *TF baseline* reaches higher throughput values with a larger batch size, achieving a maximum of approximately 130 MiB/s. Additionally, with *TF baseline* the read throughput does not scale with the batch size, when training the I/O intensive models. As previously described, this is due to the fact that with *TF baseline* I/O is the bottleneck, therefore improving the computing performance does not enhance the overall training time.

TF optimized reaches higher throughput values with a larger batch size, with both I/O-bound and compute-bound models. Moreover, *TF optimized* reaches a maximum read throughput of approximately 975 MiB/s, when training LeNet, and 850 MiB/s when training AlexNet. With ResNet-50 the read throughput is significantly lower, reaching a maximum of around 140 MiB/s.

With PRISMA, the read throughput also scales with the batch size, reaching maximum values of approximately 700 MiB/s for both LeNet and AlexNet, and around 135 MiB/s for ResNet-50. Therefore, PRISMA reaches a considerably higher read throughput than *TF baseline*.

Figure 5.7 depicts the disk read throughout over time for all models. According to the time series, *TF baseline* and *TF optimized* achieve more consistent values (i.e., values with less variance) than PRISMA. Although PRISMA intersects the read operations of the training files, it has no influence on the validation files I/O. Actually, PRISMA is not even aware when the validation files are read. Since PRISMA runs on top of *TF baseline*, the validation files are sequentially read using *sequential interleave*. For this reason, at the end of each epoch, when the validation files start being fetched, this causes the read throughput to drop. Furthermore, this interferes with PRISMA performance, since temporarily, when the validation files are being read, PRISMA internal buffer reaches its maximum capacity, what causes the producer threads to sit idle waiting for free space in the buffer. All these factors contribute to the fact that the read throughput of PRISMA varies more than *TF baseline* and *TF optimized*. The variance of PRISMA read throughput over time reflects on the average results, which prevents PRISMA from reaching a read throughput as high as *TF optimized*.

GPU USAGE Apart from the disk read throughput, the GPU utilization was also assessed. Figures 5.8 to 5.10 depict the average GPU usage of all models. The *TF baseline* version reaches a higher GPU usage with smaller batch sizes, when training the I/O intensive models (i.e., LeNet and AlexNet). On the contrary, with the compute intensive model (i.e., ResNet-50), the GPU usage scales with the batch size. Given this, *TF baseline* reaches a maximum GPU usage of around 25%, with LeNet, and 60% with AlexNet. As expected, the higher GPU usage is achieved with ResNet-50, being approximately 90%.

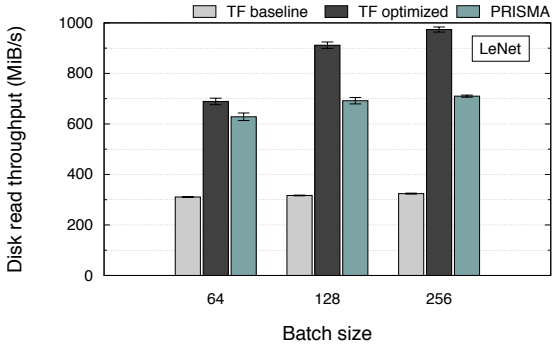


Figure 5.4: Average disk read throughput for TensorFlow and PRISMA setups with LeNet.

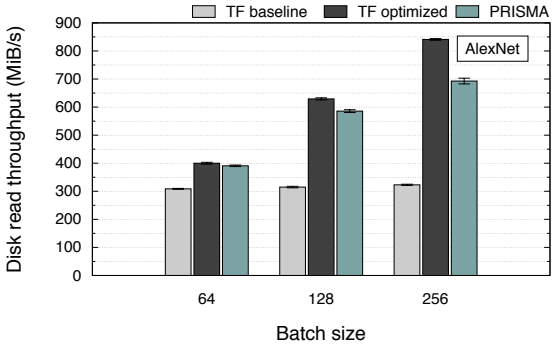


Figure 5.5: Average disk read throughput for TensorFlow and PRISMA setups with AlexNet.

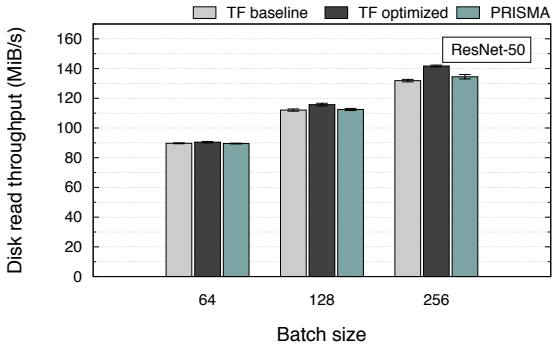


Figure 5.6: Average disk read throughput for TensorFlow and PRISMA setups with ResNet-50.

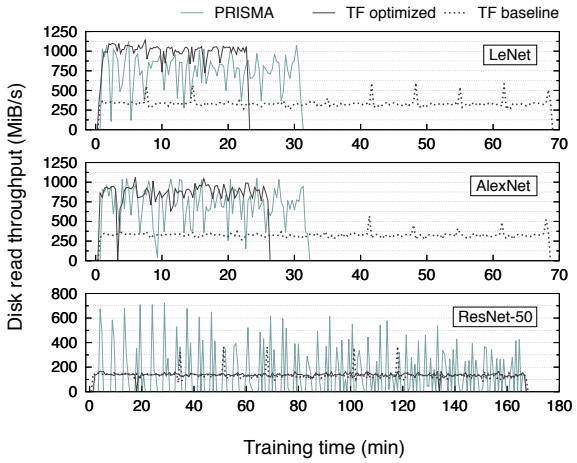


Figure 5.7: TensorFlow and PRISMA disk read throughput over time with a batch size of 256.

The average GPU usage of *TF optimized* increases with a larger batch size in all scenarios, except when training LeNet with a batch size of 256. That said, *TF optimized* achieves a maximum GPU usage of around 55% with LeNet, and 80% with AlexNet. With ResNet-50 the maximum GPU usage rises to approximately 90%.

When training the LeNet model with PRISMA, the GPU usage decreases when increasing the batch size. With AlexNet, the same happens when increasing the batch size from 128 to 256. However, when increasing the batch size from 64 to 128, the AlexNet GPU usage increases. Similarly, with ResNet-50, the GPU usage increases with a larger batch size. Given this, PRISMA achieves a maximum GPU usage of 45% with LeNet, 70% with AlexNet, and 90% with ResNet-50. According to the results, PRISMA achieves an advantage of 20% of GPU usage over *TF baseline*.

With a larger batch size, the GPU usage is expected to increase, however if there is an I/O bottleneck, it can have the opposite effect. Specifically, a larger batch size means the GPU is going to train with more data in each iteration, therefore requiring a higher read throughput. If the throughput is not high enough, the GPU may end up wasting most of the time waiting for batches to consume,

causing the GPU to be underutilized. This is why, in certain scenarios, GPU usage does not increase with the batch size.

The time series depicted in Figure 5.11 represents the GPU usage throughout time for all models. The fluctuations on PRISMA disk read throughput have an effect on the GPU usage. This occurs due to the fact that when read throughput is lower, the GPUs have to wait more time to receive the training batches, causing more idle time and, consequently, impacting GPU usage.

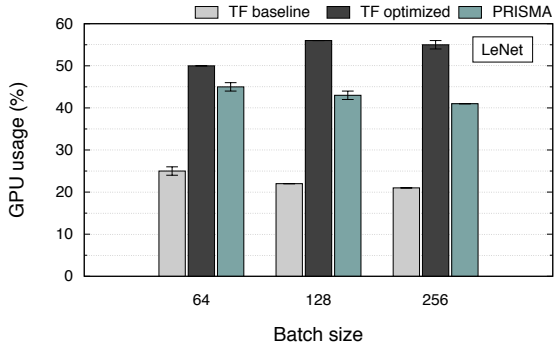


Figure 5.8: Average GPU usage for TensorFlow and PRISMA setups with LeNet.

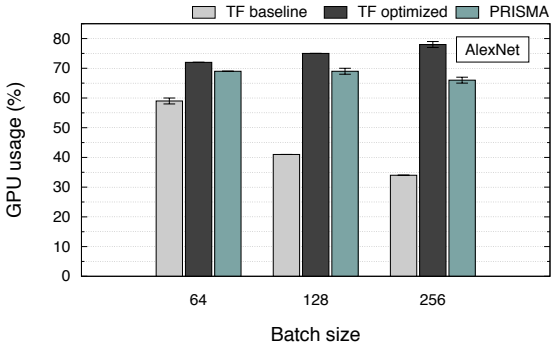


Figure 5.9: Average GPU usage for TensorFlow and PRISMA setups with AlexNet.

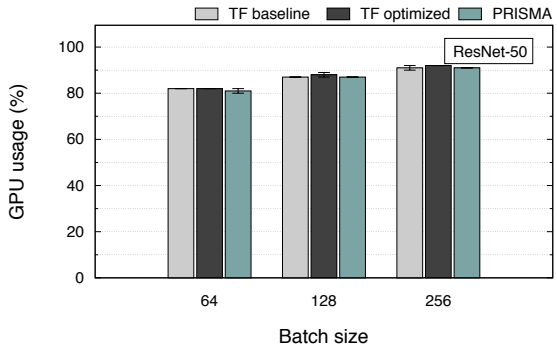


Figure 5.10: Average GPU usage for TensorFlow and PRISMA setups with ResNet-50.

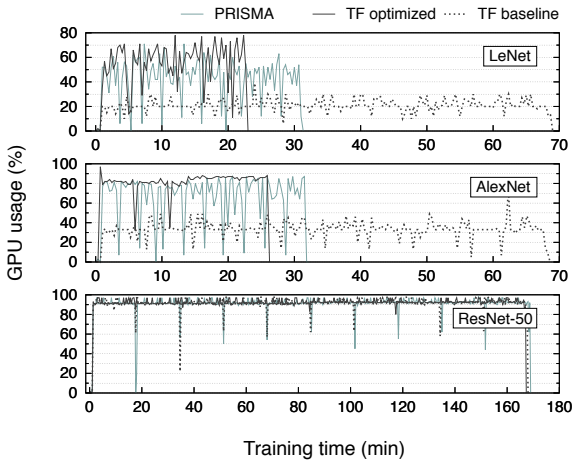


Figure 5.11: TensorFlow and PRISMA GPU usage over time with a batch size of 256.

MEMORY USAGE We now discuss the impact of each setup on memory usage. The average memory usage of all models is presented in Figures 5.12 to 5.14. Regardless of the model or batch size used, the average memory usage remains practically constant. *TF baseline* uses a maximum of approximately 20 GiB of memory. *TF optimized* and PRISMA consume approximately the same amount of memory, reaching a maximum of around 22 GiB. This means that the autotuning features of TensorFlow and PRISMA are making similar decisions in terms of buffer size. Based on the results obtained, in the worst case scenario, using parallel I/O and data prefetching increases the memory usage by 5 GiB.

Figure 5.15 depicts the memory usage over time for all models. According to the time series, both PRISMA and *TF optimized* appear to have similar behaviors regarding memory usage. While *TF baseline* performs a low and consistent use of the available memory throughout time, both PRISMA and *TF optimized* present a higher memory usage with slight fluctuations, caused by the number of elements that are being stored in the prefetching buffer at any given time.

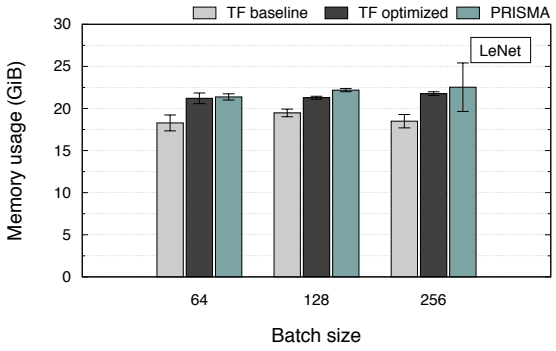


Figure 5.12: Average memory usage for TensorFlow and PRISMA setups with the LeNet model.

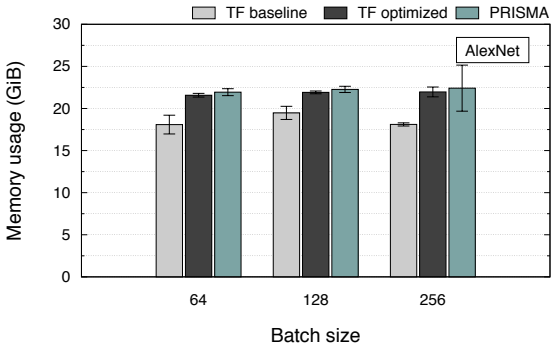


Figure 5.13: Average memory usage for TensorFlow and PRISMA setups with AlexNet.

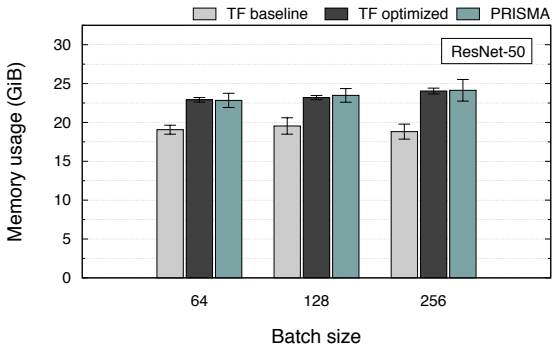


Figure 5.14: Average memory usage for TensorFlow and PRISMA setups with ResNet-50.

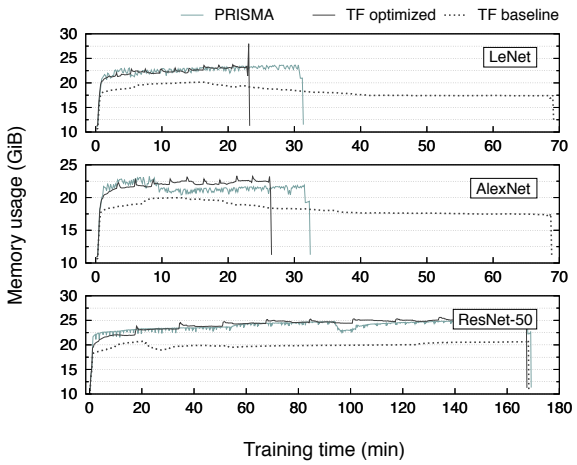


Figure 5.15: TensorFlow and PRISMA memory usage over time with a batch size of 256.

CPU USAGE Finally, CPU usage was also analyzed. Figures 5.16 to 5.18 depict the average CPU usage of all models. It is important to mention that the CPU usage presented corresponds to the sum of the user-level (application) and the system-level (kernel) usage. Increasing the batch size does not seem to have an impact on the CPU usage of *TF baseline*, due to the single I/O thread used for reading data from storage. With both LeNet and AlexNet, *TF baseline* uses approximately 15% of CPU. With ResNet-50, the CPU usage is around 6%.

TF optimized increases the CPU usage with the batch size, when training the I/O-bound models. The maximum CPU usage achieved by *TF optimized* is approximately 47% with LeNet and 42% with AlexNet. With the ResNet-50 model, *TF optimized* uses around 7% of CPU.

With the I/O intensive models, PRISMA CPU usage scales with the batch size. The maximum CPU usage reached by PRISMA is around 32% with the LeNet and AlexNet models. On the other hand, with ResNet-50, PRISMA achieves a maximum CPU usage of 7%. When training the I/O-bound models, *TF baseline* and PRISMA are less intensive than *TF optimized* in terms of CPU. This is due to the number of threads that *TF optimized* uses to read data, as described in Section 5.1.2. While *TF baseline* performs I/O using a single thread, and PRISMA uses a maximum of 4 concurrent threads, *TF optimized* reaches 30 concurrent threads during training. Consequently, compared to *TF baseline*, PRISMA and *TF optimized* may cause a 17% and 32% increase in CPU usage, respectively. Therefore, when it comes to CPU utilization, *TF optimized* is approximately 15% more expensive than PRISMA. It is important to mention that the increase in CPU usage is also caused by the fact that solutions that are more efficient in terms of I/O are faster at reading data, keeping the CPU busy more often.

With ResNet-50, the I/O threads are idle most of the time (Section 5.1.2). Similarly, the threads responsible for performing the remaining tasks of the input pipeline must also be frequently inactive. As a result, regardless of the setup, ResNet-50 is the model with the lower CPU usage.

Since LeNet is an extremely simple network, it demands low computing power, causing an intensive I/O workload [101]. Thus, increasing the batch size beyond a certain point fails to improve compute performance, which may not only be causing the GPU and CPU usage to remain constant, but can also be the reason why *TF optimized* and PRISMA cannot improve the training time when using LeNet with a batch size of 256 (Figure 5.1).

The time series depicted in Figure 5.19 represents the CPU usage over time for all models. The abrupt drops caused by the fact that PRISMA does not optimize the I/O performance of validation files also reflect on PRISMA CPU usage. Therefore, throughout the overall execution PRISMA CPU utilization varies more than *TF baseline* and *TF optimized*.

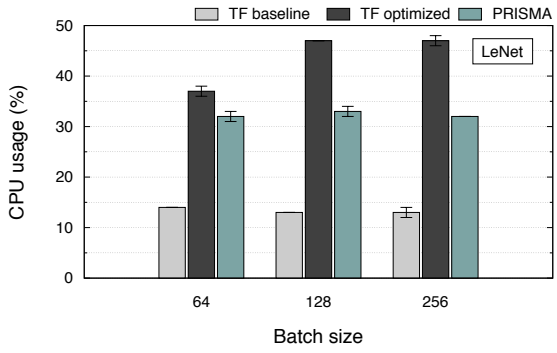


Figure 5.16: Average CPU usage for TensorFlow and PRISMA setups with LeNet.

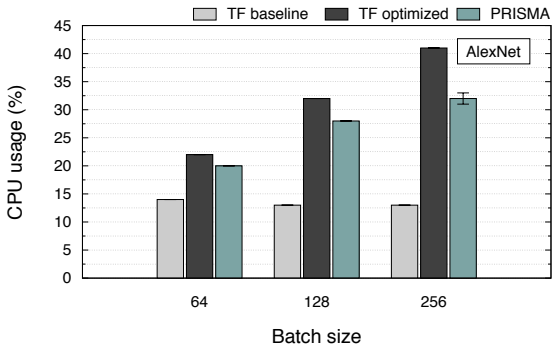


Figure 5.17: Average CPU usage for TensorFlow and PRISMA setups with AlexNet.

PRISMA Autotuning

We now discuss the impact of the PRISMA autotuning algorithm, when using the TensorFlow DL framework. Figure 5.20 depicts the selected buffer size of the autotuning algorithm. For all models

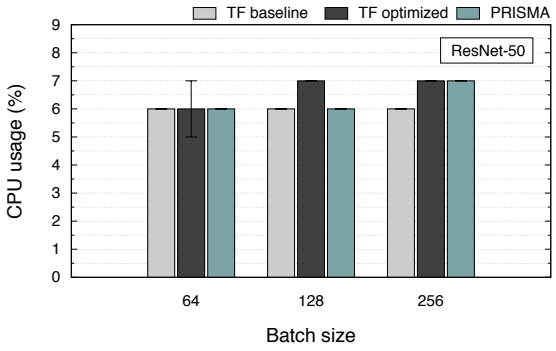


Figure 5.18: Average CPU usage for TensorFlow and PRISMA setups with ResNet-50.

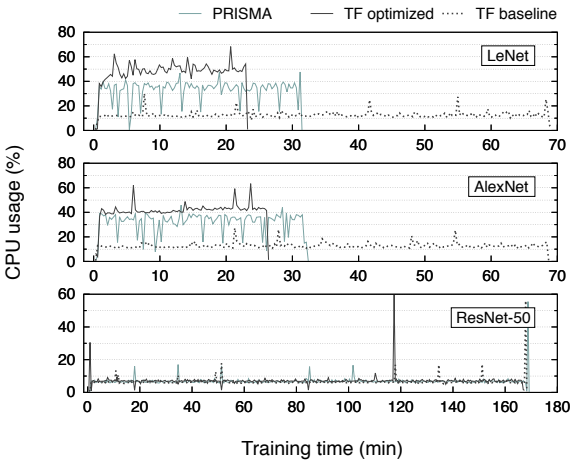


Figure 5.19: TensorFlow and PRISMA CPU usage over time with a batch size of 256.

and batch sizes, the algorithm selects a buffer size of 20 elements, which means that prefetching a maximum of 20 ILSVRC2012 TFRecords is enough to keep the TensorFlow consumer thread busy. A buffer size of 20 elements consumes a maximum of 2.7 GB of memory, considering that each ILSVRC2012 TFRecord has approximately 135 MB (Section 3.2.3).

The number of threads selected by the PRISMA autotuning algorithm is represented on Figure 5.21. Regardless of the model and batch size, PRISMA uses more producer threads for the LeNet model than for AlexNet and ResNet-50. Similarly, AlexNet uses more producer threads than ResNet-50. Therefore, the number of threads chosen by the autotuning algorithm scales with the I/O requirements of the model.

When the batch size increases, the throughput demands are higher, and thus the number of threads selected by the autotuning algorithm should not decrease with a larger batch size, as occurs with LeNet. Due to the simplicity of the LeNet model, increasing the batch size beyond 128 does not seem to have benefits (Figure 5.1). This reflects on the number of threads selected by the autotuning algorithm, causing it to decrease with a batch size of 256.

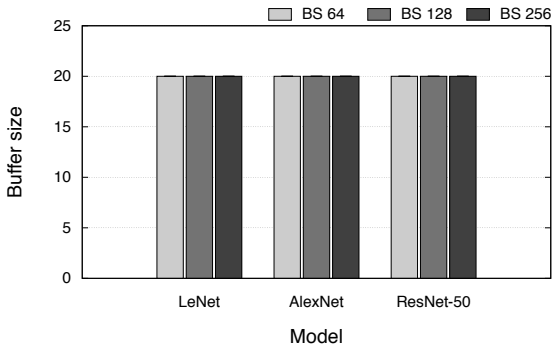


Figure 5.20: Buffer size selected by the PRISMA autotuning mechanism with TensorFlow.

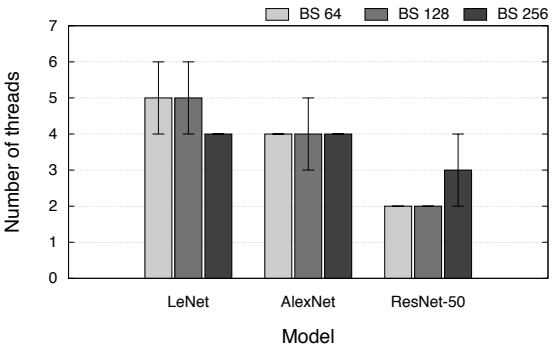


Figure 5.21: Number of threads selected by the PRISMA autotuning mechanism with TensorFlow.

To determine if the autotuning mechanism can achieve better results than configuring the buffer size and number of threads manually, two different PRISMA configurations (*Minimum* and *Maximum*) were compared to the autotuning mechanism. The *Minimum* configuration consists of manually setting the buffer size and number of threads to its minimum values, 1 and 10, respectively. In contrast, the *Maximum* configuration consists of manually setting the buffer size to 60 (i.e., approximately 8.1 GB of prefetched data) and the number of threads to 30 (i.e., the maximum number of threads used by *TF optimized*). Figure 5.22 presents the average training time of LeNet, the most I/O intensive model, with each PRISMA configuration, while varying the batch size between 64, 128 and 256. The results were obtained by measuring the average values of 5 runs.

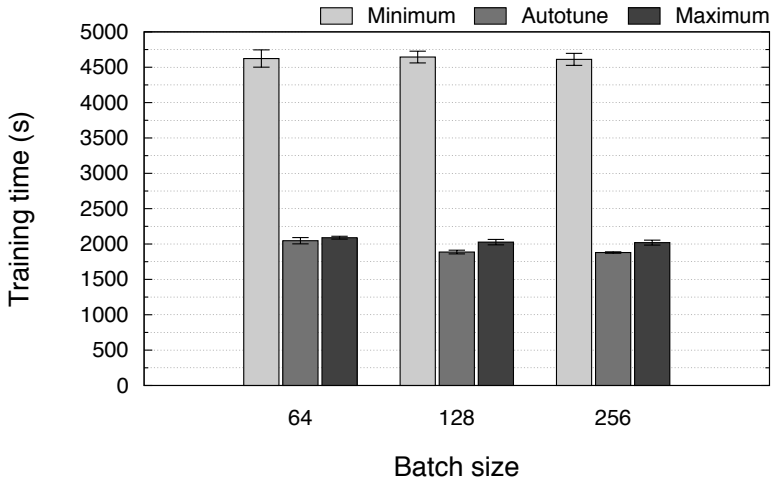


Figure 5.22: Training time of the PRISMA autotuning mechanism compared to manual settings with TensorFlow.

According to the results obtained, the *Minimum* configuration takes approximately 4600 seconds to finish the 10 training epochs, regardless of the batch size. In contrast, the *Maximum* configuration takes a minimum of 2000 seconds. Moreover, the autotuning mechanism achieves a minimum training time of approximately 1850 seconds, outperforming both the *Minimum* and *Maximum* configurations. This proves that the autotuning mechanism can, in fact, provide successful configurations.

Concurrent Threads

The number of concurrent threads used by PRISMA to prefetch data was evaluated, using the same strategy described in Section 3.2.5. In this case instead of using the timestamps relative to TensorFlow `pread()` system call, the number of concurrent active threads was obtained based on the timestamps of PRISMA `pread()` calls.

Figure 5.23 depicts a CDF with the percentage of time spent by N threads actively reading data in simultaneous from the backend storage, for both TensorFlow and PRISMA autotune features. The results are represented for each training model, namely LeNet, AlexNet, and ResNet-50.

According to Figure 5.23, while PRISMA uses a maximum of 4 concurrent threads (or 3 in the case of ResNet-50), TensorFlow reaches approximately 7 times more threads. It is important to mention

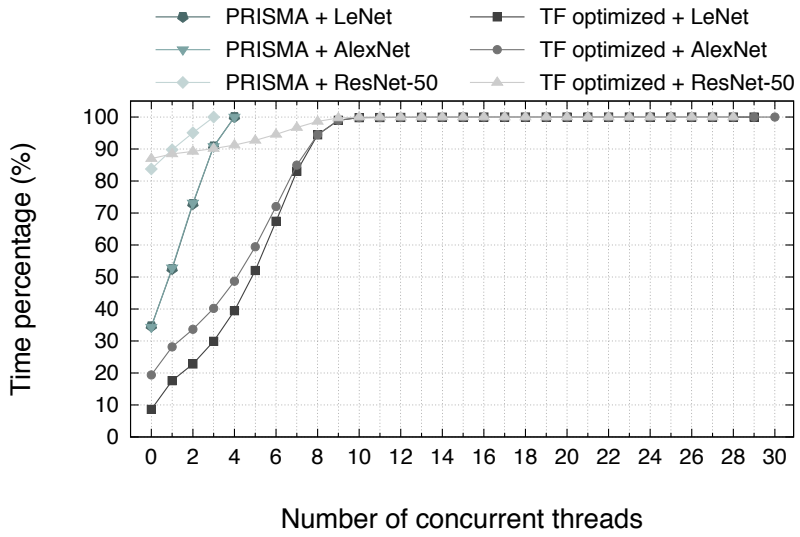


Figure 5.23: Time percentage of each number of TensorFlow and PRISMA concurrent threads.

that, in terms of training time, *TF optimized* improves the baseline scenario by a maximum of 67%, while PRISMA falls short behind improving the time by 54%. These values are obtained for the LeNet model and with AlexNet the difference between the two solutions is even less noticeable. As for ResNet-50, there is no advantage in using *TF optimized* instead of PRISMA, proving the extra 25 threads used by *TF optimized* to be completely useless. Although *TF optimized* rarely reaches 30 concurrent threads, this resources are unnecessarily allocated. In contrast, PRISMA autotuning mechanism only allocates the threads that it selects, providing more resources for other applications that may be running on the same compute node. In a scenario where the dataset is accessed by several TensorFlow instances, TensorFlow autotune feature will saturate the file system more easily than PRISMA, since it uses more threads to read from storage at the same time [70, 135].

When using the ResNet-50 model, approximately 85% of the time there are no PRISMA producer threads reading data. This means that 85% of the time the producer threads are idle waiting for free space in the buffer, which once again proves that ResNet-50 spends most of the time performing computation. *TF optimized* shows a similar behavior for ResNet-50, however it still reaches the excessive value of 28 concurrent threads. With LeNet and AlexNet, PRISMA producer threads appear to be idle approximately 35% of the time, a higher value than *TF optimized*, which reveals an idle time percentage of approximately 20% and 10% with LeNet and AlexNet, respectively. This difference may be caused by the fact that PRISMA runs on top of *TF baseline*, meaning that there is only one consumer thread requesting data from PRISMA. Due to this, PRISMA production rate is much higher than the consumption rate, causing the buffer to be full most of the time and, consequently, causing the producers to block while waiting for free space in the buffer.

Figure 5.23 shows that most of the time ResNet-50 does not perform I/O, i.e., has 0 active I/O threads. To complement this, Figure 5.24 presents PRISMA buffer usage throughout time for each model, using a batch size of 256 (the same used to obtain the number of concurrent threads). The results prove that, in fact, AlexNet and LeNet make similar use of the buffer, hence they have identical idle time percentages. ResNet-50 shows a consistently higher buffer usage when compared to the

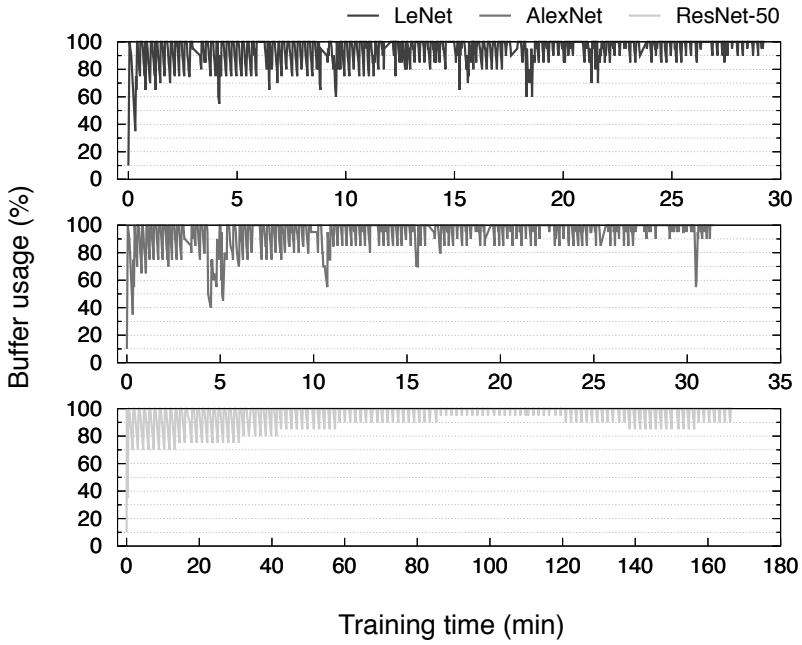


Figure 5.24: PRISMA buffer usage over time.

I/O intensive models, repeatedly reaching 100% utilization. This justifies the approximately 85% of idle time reached by ResNet-50 because the longer the buffer is full, the longer the threads will be idle (i.e., inactive) waiting for space in the buffer. It is worth noting that the values presented in Figure 5.24 are relative to a single run, while those in Figure 5.23 represent the average of 5 runs.

Model Accuracy

Given that when using PRISMA, the training filenames are globally shuffled before creating the filenames list shared between the DL framework and PRISMA, the accuracy of the model is not affected. To prove this, the accuracy of the ResNet-50 model was measured using *TF baseline*, *TF optimized* and PRISMA. Table 2 presents the top-1 accuracy of each setup after running 10 epochs, for a batch size of 64, 128, and 256. The results were obtained by measuring the average accuracy of 5 different runs.

Table 2: Top-1 accuracy of the ResNet-50 model.

Batch Size	**TF baseline**	**TF optimized**	**PRISMA**
64	$47.4 \pm 0.9\%$	$47.2 \pm 0.8\%$	$47.6 \pm 0.5\%$
128	$46.4 \pm 0.5\%$	$45.2 \pm 2.6\%$	$45.8 \pm 1.9\%$
256	$43.0 \pm 1.2\%$	$44.4 \pm 2.1\%$	$44.4 \pm 1.5\%$

As expected, PRISMA achieves approximately the same accuracy as *TF optimized* and *TF baseline*.

5.2 PYTORCH

We now discuss the experimental evaluation conducted with the PyTorch framework. We first describe the experimental setup and methodology used, and then depict and analyze the results achieved under different scenarios.

5.2.1 *Methodology and Experimental Setup*

All experiments were conducted on a single compute node of the ABCI supercomputer. Both hardware and software specifications were already identified in Table 1. PyTorch 1.7.0 was the version used to perform the experiments.

To ensure that the entire dataset would not fit in memory, we used cgroups to limit the Python process that executed PyTorch to 64 GiB of memory. The overall system resources and GPU utilization were observed using dstat and nvidia-smi, respectively.

MODELS Since ResNet-50 does not take advantage of the proposed I/O optimizations, PyTorch was evaluated only with the I/O intensive models, namely LeNet and AlexNet. Once again, to perform this experiments, the LeNet network used was the one that had to be adapted for the preliminary studies (Chapter 3).

DATASET The models were trained with the ILSVRC2012 dataset, however this time it was not converted to TFRecords, but to PyTorch tensors [92]. PyTorch Data Loader has the possibility of using worker processes to parallelize data loading and to prefetch data [27]. Although with TensorFlow input pipeline the samples are read and preprocessed simultaneously by different threads, with PyTorch the worker processes that fetch data are the same ones that perform the preprocessing. Therefore, while a worker is busy preprocessing data, it cannot perform I/O, thus causing I/O performance to be dependent on data preprocessing. In other words, no matter how efficient the I/O is, it will always be limited by the preprocessing time of each sample. Given this, if PRISMA was compared to PyTorch using raw images, PRISMA would not improve the overall training time, since the improvement in I/O performance would be concealed by the preprocessing time. To address this, the images were stored in PyTorch tensors after being preprocessed. With this strategy, the worker processes only need to load the tensors and no longer need to apply any transformations (e.g., cropping, resizing). Although this is more fair than comparing PRISMA and PyTorch using raw images, the I/O performance is still dependent on the tensor loading time.

Each tensor has an average size of 300 KB, opposed to the 115 KB of each raw image. So that the total dataset size would not exceed the size of the original ILSVRC2012 dataset (144 GB), but at the same time not allowing the entire dataset to fit into the 64 GiB of memory restricted by cgroups, only a quarter of the original dataset was converted to tensors. That said, the experiments were performed with a 94 GB dataset, stored in the local NVMe SSD disk of the compute node. In case the entire dataset was converted to tensors, the dataset would have a total size of approximately 370 GB, exceeding the size of the original dataset.

PRISMA was compared to PyTorch while training LeNet and AlexNet for 10 epochs. ResNet-50 was not evaluated due to the fact of being a compute-bound model. The experiments were performed with a batch size of 256, since this represents the most intensive I/O workload. To train with the 4 GPUs available on the compute node, `torch.nn.DataParallel` [28] was used. Considering that this strategy divides the batch across all GPUs, each GPU trained with a batch size of 64. Unless otherwise stated, for each experiment were performed 5 runs and the average values of each metric were measured. PRISMA `max_buffer_size` and `max_n_threads` configurations were set to 60 and 80 (value returned by `std::thread::hardware_concurrency()`), respectively, during the experiments.

As previously stated, PyTorch uses worker processes to parallelize I/O and prefetch data. However, this framework does not provide an autotuning mechanism to select the optimal value for the number of workers. Given this, PyTorch was evaluated using a variable number of workers, namely 0, 2, 4, 8, and 16 workers. Since the workers are also used for loading tensors, PRISMA was evaluated with more than one worker to understand the impact of the data loading performance on the overall training time. When the number of worker processes is set to 0, it is the main process that does all the work related to loading data, so this represents the baseline scenario of PyTorch. According to the PyTorch ImageNet implementation [40] used to perform the experiments, the default number of workers is 4.

PyTorch also offers the possibility of defining the number of samples that each worker prefetches, using the `prefetch_factor` argument of the PyTorch Data Loader [27]. In the experiments the `prefetch_factor` default value was used, meaning that a total of $2 \times \#workers$ samples will be prefetched by PyTorch.

Similarly to the TensorFlow case study, in these experiments PRISMA was used exclusively to prefetch training files and not validation files.

5.2.2 *Results*

Training Time

The training time of PyTorch was measured using only the main process (i.e., 0 workers) and using 2, 4, 8, 16 and 32 workers. Given that PyTorch reaches its peak performance with 16 workers, this was the maximum number of workers with which PRISMA was evaluated.

Figures 5.25 and 5.26 show the training time of both LeNet and AlexNet, respectively, using PyTorch and PRISMA. According to the obtained results, PRISMA outperforms PyTorch with 0, 2 and 4 workers, however PyTorch achieves lower training times with 8 and 16 workers. PyTorch reaches a minimum training time of approximately 750 seconds, and a maximum of approximately 4250 seconds, for both LeNet and AlexNet. On the other hand, PRISMA achieves a minimum training time of around 1100 seconds, and a maximum of approximately 1500 seconds, for both models.

Since the performance of PRISMA improves with a higher number of workers, this proves that even with PyTorch tensors, the tensor loading time has an influence on the I/O performance. Nevertheless, with a higher number of workers, the overhead caused by the concurrency control mechanisms used by PRISMA ends up overcoming the benefit of parallelizing the loading of tensors, degrading the performance of PRISMA. This overhead arises from the continuous acquisition and release of locks, and the wait that each process has to perform while it is unable to acquire the lock. Given

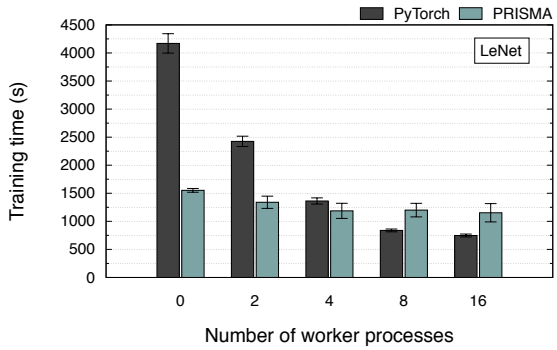


Figure 5.25: Average training time of PyTorch and PRISMA with LeNet.

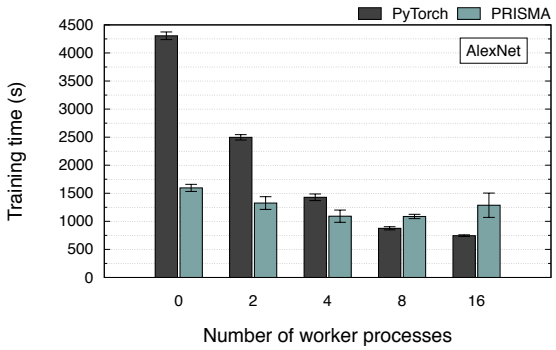


Figure 5.26: Average training time of PyTorch and PRISMA with AlexNet.

this, PRISMA improves the training performance of the baseline scenario by a maximum of 72% for LeNet with 16 workers and 75% for AlexNet with 8 workers, while PyTorch improves the training performance by a maximum of 82% for LeNet and 83% for AlexNet, both with 16 workers. Apart from this, PRISMA outperforms PyTorch with 2 and 4 workers due to the fact that it prefetches samples independently of the training epoch, and PyTorch only starts prefetching elements for an epoch when that epoch begins.

Resource Usage

The figures provided in this section depict the resource usage of PyTorch and PRISMA, when training LeNet and AlexNet with multiple workers. The histogram plots present the average usage of a given resource, based on the 5 runs that were performed for a variable number of workers. The time series show the resource usage over time, and represent a single run with the LeNet model for 0, 4 and 16 workers, since 0 workers represents the baseline scenario, 4 workers is the default value, and 16 workers is the optimal configuration for PyTorch. The time series relative to the resource usage of LeNet with 2 and 8 workers, as well as for AlexNet with 0, 2, 4, 8, and 16 workers can be found in Section A.1.2 of the Appendix.

DISK READ THROUGHPUT The first resource usage analysis to be performed concerns disk read throughput. Figures 5.27 and 5.28 represent the average read throughput of the two models, for multiple workers. With PyTorch, the read throughput scales with the number of worker processes. That said, PyTorch reaches a maximum read throughput of approximately 1200 MiB/s, for both models. PRISMA read throughput increases with the number of workers, achieving a maximum read throughput of approximately 800 MiB/s, for the two models. Moreover, PRISMA reaches a higher read throughput than PyTorch for a lower number of workers (e.g., 0, 2, 4), however PyTorch achieves better results with 8 and 16 workers. This is due to the fact that with a higher number of worker processes, the overhead caused by the concurrency control mechanisms impairs PRISMA performance.

Figure 5.29 depicts the disk read throughout over time of the two models. When it comes to read throughput, PyTorch demonstrates more consistent (i.e., less variable) values than PRISMA. PyTorch only initiates the prefetching process for an epoch, when that same epoch starts. Given this, in the

period of time between the end of one epoch and the beginning of the next, there is a drop in the framework consumption rate, causing the buffer to reach its maximum capacity more often. This means that the producer threads do not have to fetch data as often, momentarily affecting the read throughput. The variability is more accentuated with 0 workers, since with 4 workers the decrease in the consumption rate is much faster and subtle.

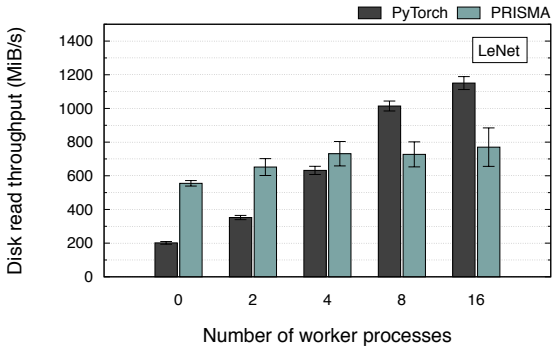


Figure 5.27: Average disk read throughput for PyTorch and PRISMA setups with LeNet.

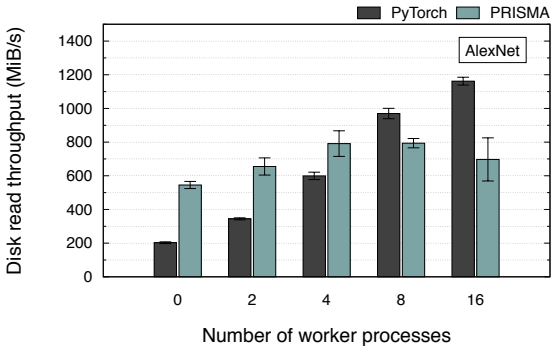


Figure 5.28: Average disk read throughput for PyTorch and PRISMA setups with AlexNet.

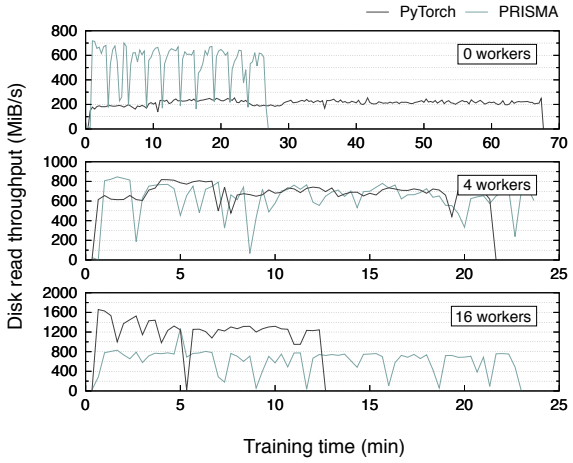


Figure 5.29: PyTorch and PRISMA disk read throughput over time with LeNet for 0, 4, and 16 workers.

GPU USAGE Apart from the disk read throughput, the GPU utilization was also assessed. Figures 5.30 and 5.31 depict the average GPU usage of the two models, for multiple workers. Given that read throughput directly impacts the GPU usage, PRISMA and PyTorch GPU usage show an identical behavior to the read throughput. PyTorch reaches a maximum GPU usage of approximately 30% for LeNet and 25% for AlexNet. On the other hand, PRISMA reached a maximum GPU usage of around 20% for LeNet and 15% for AlexNet. In general, PRISMA demonstrates to have a higher GPU usage than PyTorch with 0, 2 and 4 workers.

The time series depicted in Figure 5.32 represents the GPU usage over time of the two models. Throughout the overall execution, PyTorch and PRISMA GPU usage demonstrate similar variations.

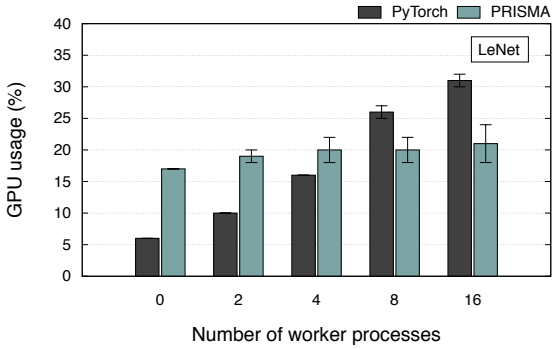


Figure 5.30: Average GPU usage for PyTorch and PRISMA setups with LeNet.

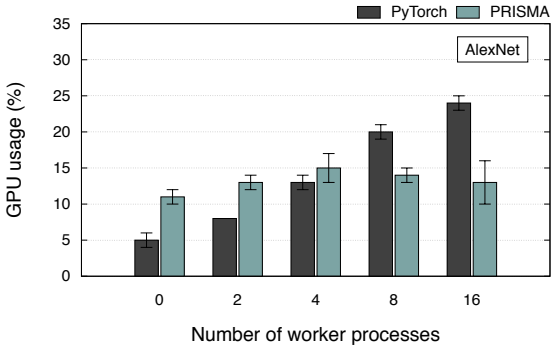


Figure 5.31: Average GPU usage for PyTorch and PRISMA setups with AlexNet.

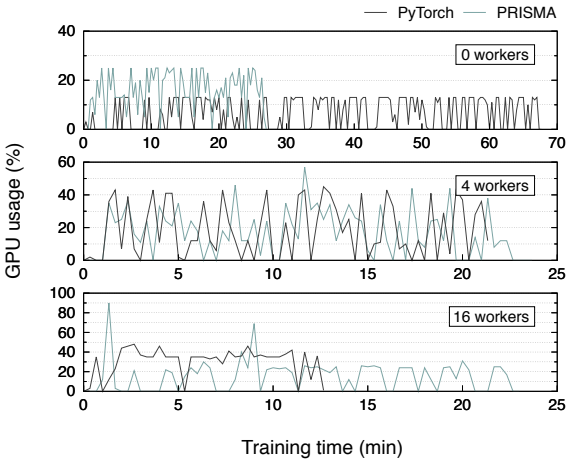


Figure 5.32: PyTorch and PRISMA GPU usage over time with LeNet for 0, 4, and 16 workers.

MEMORY USAGE We now discuss the impact of each setup on memory usage. The average memory usage of the two models, for multiple workers, is presented in Figures 5.33 and 5.34. The average memory consumption of both PyTorch and PRISMA increases with the number of workers. PyTorch reaches a maximum memory utilization of around 17.5 GiB, and a minimum of 15 GiB, for both LeNet and AlexNet. In contrast, PRISMA achieves a maximum of nearly 19 GiB, and a minimum of 17 GiB, for both models. Since PRISMA runs on top of the PyTorch version with which is being compared, the difference of memory between PyTorch and PRISMA represents the consumption of memory made by PRISMA. Considering this, PRISMA adds approximately 2 GiB of memory usage to the PyTorch implementation.

Figure 5.35 depicts the memory usage over time of the two models. During training, both PRISMA and PyTorch show similar memory usage variations. While with TensorFlow the buffer usage over

time was reflected in small memory consumption fluctuations (Figure 5.15), in the case of PyTorch this is not noticeable, not only because the buffer is smaller (Section 5.2.2), but also due to the fact that the training files take up much less space. Whereas in the case of TensorFlow the full buffer consumes 2.7 GB of memory, with PyTorch the full buffer is only equivalent to 1.2 MB, hence it is not possible to observe fluctuations in the memory usage time series.

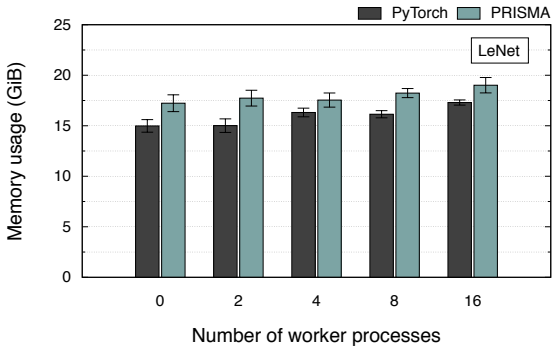


Figure 5.33: Average memory usage for PyTorch and PRISMA setups with LeNet.

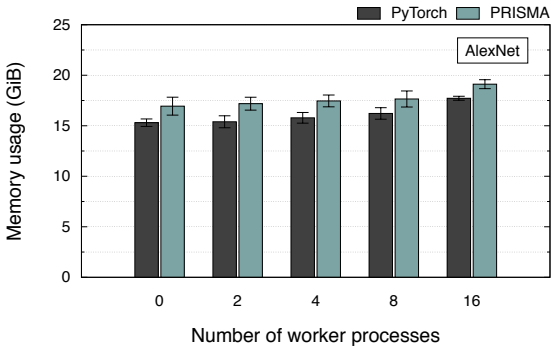


Figure 5.34: Average memory usage for PyTorch and PRISMA setups with AlexNet.

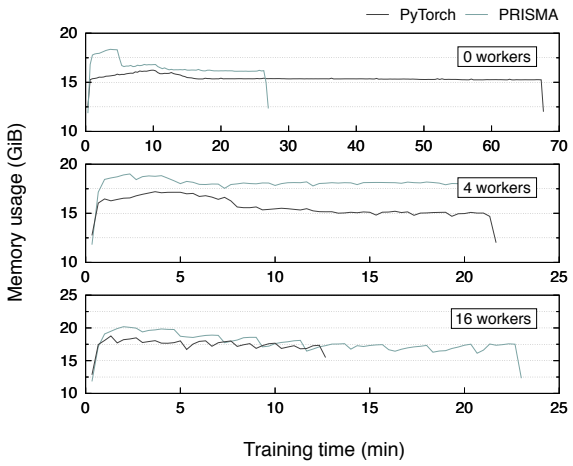


Figure 5.35: PyTorch and PRISMA memory usage over time with LeNet for 0, 4, and 16 workers.

CPU USAGE Finally, CPU usage was also analyzed. Figures 5.36 and 5.37 depict the average CPU usage of both models, for multiple workers. It is important to mention that the CPU usage presented corresponds to the sum of the user-level (application) and the system-level (kernel) usage. PyTorch CPU usage varies from 2% to 6%, with both LeNet and AlexNet. With PRISMA the maximum CPU usage achieved was approximately 17%, and the minimum 4%. Except for 0 workers, PyTorch and PRISMA have a very similar CPU usage.

Typically, the better performance the setup has, the more CPU it will use, due to the fact that the data has to be processed by the CPU before being transferred to the GPU. With 0 workers PyTorch

uses only one process to perform data loading. Moreover, with this setup PRISMA client-server implementation is not used, therefore PyTorch main process communicates directly with PRISMA and does not have to wait for the server to intermediate the read operations. Consequently, both PyTorch main process and PRISMA consumer threads have less wait time, causing the CPU to be higher than with other setups.

The time series depicted in Figure 5.38 represents the CPU usage over time of the two models. Throughout the overall execution, the CPU usage is similar to the read throughput, since when the throughput is lower it means that less threads are actively reading data, therefore less CPU is consumed.

Figure 5.36: Average CPU usage for PyTorch and PRISMA setups with LeNet.

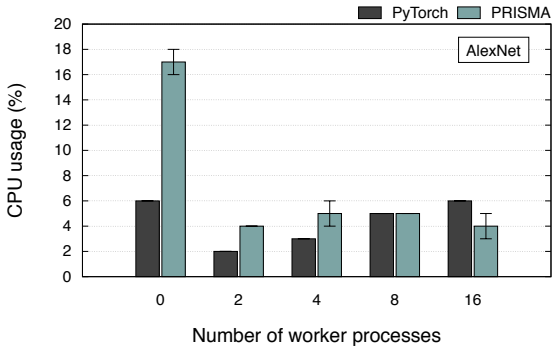


Figure 5.37: Average CPU usage for PyTorch and PRISMA setups with AlexNet.

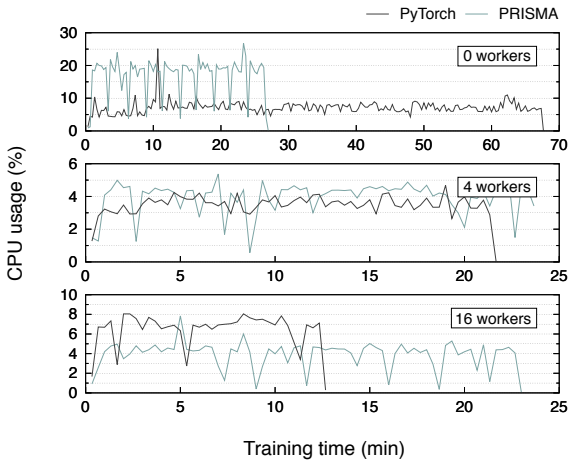


Figure 5.38: PyTorch and PRISMA CPU usage over time with LeNet for 0, 4, and 16 workers.

PRISMA Autotuning

To prove the validity of PRISMA autotuning mechanism when using PyTorch, we measured the selected buffer size and number of threads, which are depicted in Figures 5.39 and 5.40, respectively.

The autotuning algorithm was designed based on the TensorFlow case study, i.e., considering the internal mechanisms of TensorFlow and taking into account that the training samples consisted of ILSVRC2012 TFRecords. When analyzing the obtained results, with a higher number of workers, the autotuning decisions exhibit more variation, than with 0 workers. Since the training files used with PyTorch are approximately 1200 times smaller than the ones used with TensorFlow, the file production and consumption rates are much higher with PyTorch. Moreover, while TensorFlow uses only one consumer thread, PyTorch uses multiple workers. Both of these factors are impacting the autotuning algorithm decisions, causing it to occasionally select a higher buffer size. Thus, the autotuning algorithm should be fine-tuned to converge to more consistent buffer sizes in scenarios where consumption and production rates are higher. Nevertheless, according to Figures 5.25 and 5.26, the variance in the selected buffer size does not impact the training performance between runs.

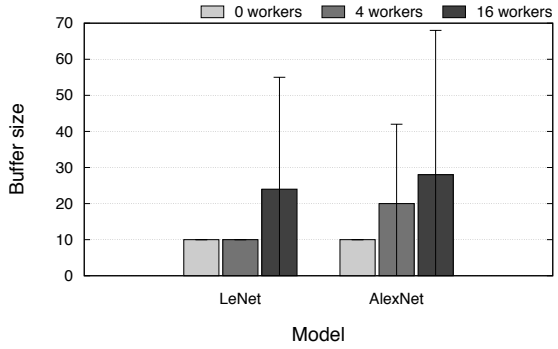


Figure 5.39: Buffer size selected by the PRISMA autotuning mechanism with PyTorch.

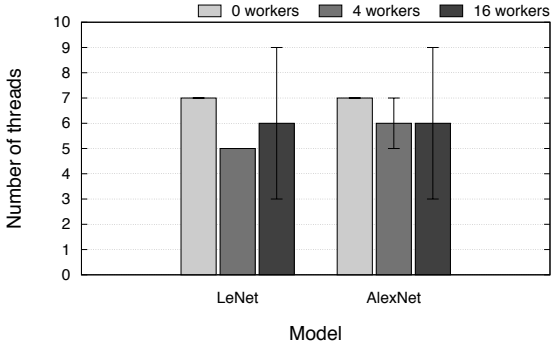


Figure 5.40: Number of threads selected by the PRISMA autotuning mechanism with PyTorch.

Similarly to TensorFlow (Section 5.1.2), PRISMA autotuning mechanism was compared to a *Minimum* and *Maximum* configurations when training LeNet. Since the autotuning algorithm used with PyTorch was the same designed for TensorFlow, we decided to use the the same *Minimum* and *Maximum* configurations adopted in Section 5.1.2. As such, the *Minimum* configuration consists of manually setting the buffer size to 10 and the number of threads to 1, while the *Maximum* configuration comprises a buffer size of 60 elements and 30 threads. The results were obtained by measuring the average values of 5 runs.

Figure 5.41 presents the training time of each manual setting compared to the autotuning mechanism, with the LeNet model. According to the results obtained, although PRISMA performs significantly better than the *Minimum* configuration, its training time is worse than the *Maximum* configuration. Once again, since the autotuning algorithm was designed based on the TensorFlow case study, it does not provide the optimal performance in this scenario. Despite that the *Maximum* configuration outperforms PRISMA autotuning algorithm by up to 24%, the autotuning algorithm uses approximately 6 threads, while the *Maximum* configuration uses 30. Additionally, the autotuning algorithm has a ramp-up period until reaching the optimal configuration, whereas the *Maximum* configuration already starts the training process with the parameters set to the maximum values.

It is worth mentioning that with the *Maximum* configuration, PRISMA reaches a performance improvement of 79% with 16 workers, when compared to the baseline scenario. This value is extremely close to the 82% achieved by PyTorch, when training LeNet (Section 5.2.2).

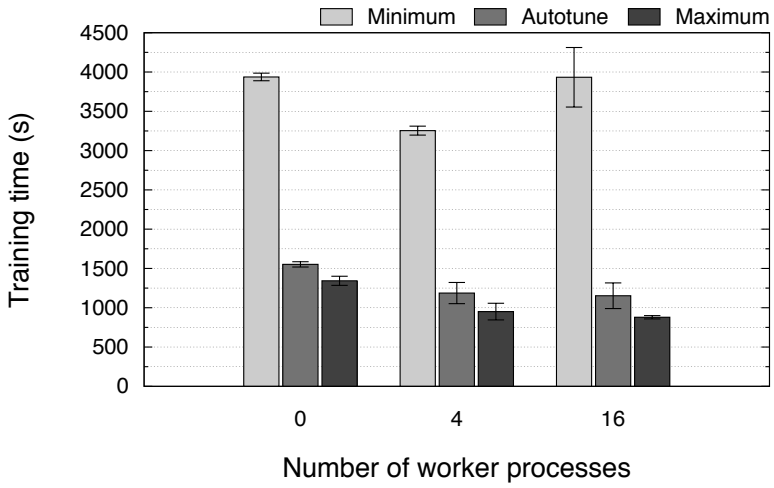


Figure 5.41: Training time of the PRISMA autotuning mechanism compared to manual settings with PyTorch.

Given that PRISMA is a framework-agnostic middleware, to prove that it can be applied to more than one framework and to demonstrate that different scenarios can benefit from this solution, we decided to integrate PRISMA with PyTorch. However, to achieve optimal performance with PyTorch, PRISMA autotuning algorithm would have to be adjusted, more specifically adapting the initial buffer size and increasing step. Still, with the autotuning algorithm used, PRISMA has a major advantage over PyTorch for a lower number a workers. More specifically, for 4 workers, which represents the default value that would be typically selected by the majority of the users, PRISMA can significantly outperform PyTorch.

5.3 DISCUSSION

Adopting a framework-agnostic solution like PRISMA has the advantage of being easily extended to different scenarios, without having to study the details and complexity of a DL framework. In contrast, the I/O optimizations currently implemented in DL frameworks are internal to the system and cannot be reused with other frameworks. TensorFlow and PyTorch case studies prove that PRISMA can be successfully applied to different DL frameworks.

Compared to TensorFlow baseline scenario, PRISMA can improve the performance of I/O intensive models by 54%. The I/O optimized version of TensorFlow achieves a higher performance, improving the baseline training time by 67%. However, while both PRISMA and *TF optimized* have the same memory cost, *TF optimized* consumes more 15% of CPU and allocates more threads than PRISMA. A scenario where this difference would be noticeable is when executing multiple TensorFlow instances (or other applications) on the same node, as is the case with the ABCI supercomputer. This is due to the fact that (i) PRISMA uses fewer concurrent threads, so it will not saturate the storage as easily; and (ii) PRISMA consumes less CPU, providing more resources for the remaining TensorFlow applications.

In addition, the accuracy of the training model is approximately the same when using *TF baseline*, *TF optimized*, and PRISMA, as demonstrated with the ResNet-50 results, which proves that PRISMA does not have an impact on the accuracy of the model.

When comparing with the baseline scenario (i.e., PyTorch with 0 workers), PyTorch I/O optimizations can improve the training time by approximately 80% for both LeNet and AlexNet. On the other hand, PRISMA improves the baseline training time by approximately 70% for both models. When up to 4 workers are used, PRISMA significantly outperforms PyTorch. Considering this, for scenarios where it is advantageous to use less workers, PRISMA is a more favorable option than PyTorch, although it has an additional cost of 2 GiB of memory. In the event that PyTorch I/O performance was not dependent on the preprocessing, PRISMA could be applied to the baseline scenario of PyTorch and possibly outperform PyTorch with a higher number of workers, since there would be no overhead caused by PRISMA concurrency control mechanisms. In addition, PRISMA client-server implementation would not need to be used with PyTorch baseline scenario, removing yet another layer of overhead.

Apart from this, when manually setting a *Maximum* configuration with 30 producer threads and a buffer size of 60, PRISMA accomplishes even lower training times, reaching values extremely close to PyTorch for a larger number of workers (e.g., 8, 16). Since PRISMA autotuning mechanism was designed based on the TensorFlow case study, it cannot outperform the *Maximum* configuration with PyTorch. Therefore, to achieve the best performance possible with PyTorch, PRISMA autotuning algorithm would have to be fine-tuned.

Compared to PyTorch, when it comes to choosing the optimal configuration, PRISMA has the upper hand. PRISMA autotuning mechanism automatically selects the optimal combination of buffer size and number of threads, preventing the user from wasting time looking for an efficient configuration. On the contrary, PyTorch configurations have to be manually set.

The results of the PyTorch case study (Section 5.2.2) indicate that there is still a margin of performance optimization for PRISMA, which involves optimizing concurrent access to PRISMA shared data structures. Regardless of the DL framework used, PRISMA could achieve better performance if the validation files were also prefetched, since these represent approximately 11% of the ILSVRC2012 dataset. According to the results presented in Section 5.2.2, PRISMA autotuning mechanism does not converge to the optimal configuration when using a high number of workers. Although this issue is partially caused by the concurrency control overhead, a larger initial buffer size would also be beneficial to improve the autotuning decisions. As such, if the PRISMA autotuning algorithm could predict an efficient initial buffer size, this would prevent the autotuning from converging to suboptimal configurations. The initial buffer size of each scenario could be based on the dimension of the dataset files, since this directly impacts the file production and consumption rates.

Currently each PRISMA instance is isolated, therefore the autotuning decisions made by one PRISMA instance are completely independent of other instances that are running on the same compute node. Thus, considering the previous ABCI use case (i.e., multiple TensorFlow instances operating concurrently), rather than enforcing local optimizations within each instance, it would be interesting that PRISMA could have global visibility, allowing all instances to be coordinated. To achieve this, PRISMA could implement the *Software-Defined Storage (SDS)* principles [65]. A possible approach for this would be having PRISMA I/O mechanisms as the data plane, making it responsible

for enforcing and implementing the storage optimizations over data; and having the control plane, which comprises global visibility of the infrastructure, holding the PRISMA autotuning algorithm and adjusting each individual PRISMA instance according to user-defined objectives.

6

CONCLUSION

I/O has proven to be one of the largest bottlenecks in DL training scenarios where the dataset does not fit in memory [37, 45, 129, 78, 88]. Thus, multiple DL frameworks have internal I/O optimizations that are specific to the framework itself and cannot be easily decoupled and applied to other frameworks. A number of external optimizations has also been proposed, such as data loading pipelining [105, 3, 137], I/O parallelization [88, 130, 55], data echoing [18], among others [79, 54, 57, 135, 138, 128]. However, all of these hold certain drawbacks, such as impacting the accuracy of the training models, being single-purposed and designed to a specific framework. Furthermore, all the proposed solutions either have to be manually tuned, or comprise greedy algorithms for resource provisioning.

To prove that the performance of local DL applications could benefit from I/O optimization, TensorFlow Dataset API was thoroughly studied and evaluated. The results of this preliminary study demonstrated that, in a single node setup, applying prefetching and I/O parallelization could improve TensorFlow performance by 67%. Thus, to address the issues previously mentioned, this dissertation proposes PRISMA, a framework-agnostic prefetching middleware for accelerating local DL workloads.

PRISMA is a framework-agnostic storage middleware that delivers efficient data prefetching of training files, and provides an autotuning mechanism for automatically determining the optimal configuration regarding the number of I/O threads and the internal buffer size, while allocating as few resources as possible. With this features, PRISMA is able to improve the I/O performance of different DL frameworks.

PRISMA was employed over TensorFlow and PyTorch, which required a simple custom integration to be implemented for each framework (more specifically, at the interaction between the framework and the backend storage). Compared to TensorFlow baseline version, PRISMA achieves a performance improvement of 54%. Although the TensorFlow I/O optimized version can improve the baseline training time by 67%, it consumes more 15% of CPU than PRISMA. Regarding the autotuning mechanisms of both TensorFlow and PRISMA, while PRISMA uses an average of 4 threads, TensorFlow ends up allocating near 30 threads for the same training scenario, although it only executes between 4 and 8 concurrent threads during training. As such, in a use case similar to the one conducted on ABCI, which consists of running multiple TensorFlow applications on the same compute node, PRISMA could have an advantage over TensorFlow in terms of I/O performance and resource usage.

When it comes to PyTorch, PRISMA performs better for a lower number of worker processes. Therefore, in a scenario where less workers are required, PRISMA would be more beneficial since it

reaches lower training times, at the cost of 2 GiB of memory. In a scenario with 16 workers, when using a manual configuration of 30 threads and a buffer size of 60 elements, PRISMA can improve the performance of the baseline scenario by 79%, a value extremely close to the 82% achieved by PyTorch. On the other hand, with the autotuning mechanism, PRISMA outperforms the baseline training time by 72%, for the same scenario, indicating that there is still room for improving the autotuning algorithm. In regards to choosing the optimal configuration, PRISMA has the advantage, since it provides an autotuning mechanism for automatically selecting the best combination of parameters.

To conclude, PRISMA is the most efficient solution in the majority of the scenarios that were studied, and performs similarly to the I/O optimized setups of both TensorFlow and PyTorch. Moreover, the case studies demonstrate that there is still a margin of performance enhancement for PRISMA, proving that with a few future optimizations, PRISMA can outperform both TensorFlow and PyTorch internal I/O optimizations.

6.1 Future Work

To improve the I/O performance of PRISMA, there are a few optimizations that could be performed. First of all, apart from prefetching the training files, PRISMA should be able of prefetching the validation files as well, since these usually represent a significant portion of the overall dataset.

According to the experiments performed, the overhead caused by the concurrency control mechanisms seem to impact PRISMA performance. Given this, optimizing the concurrent access to PRISMA shared data structures is another option that could be explored in the future.

When it comes to PRISMA autotuning mechanism, a possible enhancement consists of automatically identifying the optimal initial buffer size (and increasing step) for the I/O workload in question. This feature could be based on the size of each training sample and on the dimension of the dataset, since these factors directly impact the production and consumption rates.

As previously described, a common use case of HPC infrastructures, which is also conducted on ABCI, consists of executing multiple DL applications on the same compute node. Apart from sharing the compute resources of the node, the applications all have access to the same local device. To ensure that the provisioning process is as efficient as possible, one application should be aware of all the others running on the same compute node. At the moment, PRISMA instances are completely isolated, therefore the autotuning decisions made by one PRISMA instance are completely independent of the others. So that PRISMA could be successfully applied to the ABCI use case, the autotuning algorithm should be able to have a global view of all PRISMA instances running on the compute node. Moreover, the decisions made by the algorithm should take into account the I/O workload of each DL application, to allow provisioning to be as fair as possible. To achieve this, PRISMA could implement the SDS principles [65]. A possible approach for this would be having PRISMA I/O mechanisms as the data plane, making it responsible for enforcing and implementing the storage optimizations over data; and having the control plane, which comprises global visibility of the infrastructure, holding the PRISMA autotuning algorithm and adjusting each individual PRISMA instance according to user-defined objectives.

BIBLIOGRAPHY

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[3] A. Aizman, G. Maltby, and T. Breuel. High performance i/o for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 5965–5967. IEEE, 2019.

[4] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, I. Ahmad, R. H. Arpaci-Dusseau, F. Chen, Y. Chen, Y. Chen, Y. Cheng, V. Chidambaram, D. Da Silva, A. Demke-Brown, P. Desnoyers, J. Flinn, X. He, S. Jiang, G. Kuenning, M. Li, C. Maltzahn, E. L. Miller, K. Mohror, R. Rangaswami, N. Reddy, D. Rosenthal, A. S. Tosun, N. Talagala, P. Varman, S. Vazhkudai, A. Waldani, X. Zhang, Y. Zhang, and M. Zheng. Data storage research vision 2025: Report on nsf visioning workshop held may 30–june 1, 2018. Technical report, USA, 2018.

[5] T. Bayes. Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s. *Philosophical transactions of the Royal Society of London*, 53(53):370–418, 1763.

[6] BeeGFS. BeeGFS. `https://www.beegfs.io/content/`. Accessed November 25, 2019.

[7] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):65, 2019.

[8] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[9] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20), Banff, AL, Canada*, 2020.

[10] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[11] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9. IEEE, 2015.

[12] R. Bonnin. Using tensorflow to predict product weight and dimensions. https://blog.tensorflow.org/2019/09/using-tensorflow-to-predict-product.html, Sep 2019. Accessed December 5, 2020.

[13] A. Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019.

[14] R. Caruana, S. Lawrence, and C. L. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001.

[15] B. Cece. Ai and aerospace. https://www.airbus.com/newsroom/news/en/2016/12/Artificial-Intelligence.html, Dec 2016. Accessed November 18, 2019.

[16] cgroups. cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed October 23, 2020.

[17] S. W. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure. Characterizing deep-learning i/o workloads in tensorflow. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 54–63. IEEE, 2018.

[18] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

[19] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, page 80. ACM, 2019.

[20] H. Chu. Mdb: A memory-mapped database and backend for openldap. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, page 35. Citeseer, 2011.

[21] K. Chung. Generating recommendations at amazon scale with apache spark and amazon dsstne. https://aws.amazon.com/pt/blogs/big-data/generating-recommendations-at-amazon-scale-with-apache-spark-and-amazon-dsstne/, Jul 2016. Accessed November 18, 2019.

[22] D. Cireşan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[23] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[24] concurrent_hash_map. concurrent_hash_map. https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/concurrent_hash_map.html. Accessed October 28, 2020.

[25] D. Crevier. *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, 1993. ISBN 9780465029976. URL https://books.google.pt/books?id=QJNQAAAAMAAJ.

[26] CTPL. CTPL. https://github.com/vit-vit/CTPL, Accessed November 28, 2020.

[27] DataLoader. torch.utils.data. https://pytorch.org/docs/stable/data.html. Accessed October 20, 2020.

[28] DataParallel. DataParallel. https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html, Accessed November 10, 2020.

[29] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231, 2012.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[31] dstat. dstat(1) - linux man page. https://linux.die.net/man/1/dstat. Accessed December 29, 2019.

[32] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

[33] C. Francois. *Deep learning with Python*. Manning Publications Company, 2017.

[34] S. Ghosh and D. L. Reilly. Credit card fraud detection with a neural-network. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 3, pages 621–630. IEEE, 1994.

[35] GPUDirect RDMA. Developing a linux kernel module using gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html. Accessed October 16, 2020.

[36] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[37] J. Han, L. Xu, M. M. Rafique, A. R. Butt, and S.-H. Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[38] 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*, pages 770–778, 2016.

[39] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[40] ImageNet example. ImageNet training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet, Accessed November 10, 2020.

[41] Imagenet preprocessing. imagenet_preprocessing.py. https://github.com/tensorflow/models/blob/master/official/vision/image_classification/resnet/imagenet_preprocessing.py, Accessed November 5, 2020.

[42] interleave_dataset_op.cc. interleave_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/v2.2.0/tensorflow/core/kernels/data/interleave_dataset_op.cc. Accessed October 20, 2020.

[43] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[44] A. Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[45] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda. Performance characterization of dnn training using tensorflow and pytorch on modern clusters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[46] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[47] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[48] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. Strads: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 5. ACM, 2016.

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

[50] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[51] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[52] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[54] A. V. Kumar and M. Sivathanu. Quiver: An informed storage cache for deep learning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 283–296, 2020.

[55] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, et al. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 51. IEEE Press, 2018.

[56] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, T. Duerig, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982*, 2018.

[57] F. P. Lanaras. Reducing data path from storage to gpus for deep learning, 2018.

[58] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed November 19, 2019.

[59] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[60] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[61] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[62] S.-H. Lim, S. R. Young, and R. M. Patton. An analysis of image storage systems for scalable training of deep neural networks. 1 2016. URL https://www.osti.gov/biblio/1335300.

[63] Y. Liu, K. Gadepalli, M. Norouzi, G. E. Dahl, T. Kohlberger, A. Boyko, S. Venugopalan, A. Timofeev, P. Q. Nelson, G. S. Corrado, et al. Detecting cancer metastases on gigapixel pathology images. *arXiv preprint arXiv:1703.02442*, 2017.

[64] N. Léonard and C. M. Halasz. Twitter meets tensorflow. https://blog.twitter.com/engineering/en_us/topics/insights/2018/twittertensorflow.html, Jun 2018. Accessed November 18, 2019.

[65] R. Macedo, J. Paulo, J. Pereira, and A. Bessani. A survey and classification of software-defined storage systems. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

[66] map_dataset_op.cc. map_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/v2.2.0/tensorflow/core/kernels/data/map_dataset_op.cc. Accessed October 20, 2020.

[67] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[68] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 571–582. IEEE, 2017.

[69] Microsoft. Azure blob storage. https://azure.microsoft.com/en-in/services/storage/blobs/#overview. Accessed December 10, 2020.

[70] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding manycore scalability of file systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 71–85, 2016.

[71] MirroredStrategy. tf.distribute.MirroredStrategy. https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy, Accessed November 10, 2020.

[72] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., USA, 1 edition, 1997. ISBN 0070428077.

[73] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[74] model.cc. model.cc. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/model.cc. Accessed October 21, 2020.

[75] model_dataset_op.cc. model_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/model_dataset_op.cc. Accessed October 21, 2020.

[76] M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA:, 2015.

[77] nvidia-smi. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface. Accessed December 29, 2019.

[78] L. Oden, C. Schiffer, H. Spitzer, T. Dickscheid, and D. Pleiter. Io challenges for human brain atlasing using deep learning methods-an in-depth analysis. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 291–298. IEEE, 2019.

[79] H. Ohtsuji, E. Hayashi, N. Fukumoto, E. Yoshida, T. Okamoto, T. Kuramoto, and O. Tatebe. Mitigating the impact of tail latency of storage systems on scalable deep learning applications. *Parallel Data Systems Workshop*, 2019.

[80] N. R. C. U. C. on Innovations in Computing and C. L. from History. *Funding a revolution: government support for computing research*. National Academy Press, 1999. ISBN 9780309062787. URL https://books.google.pt/books?id=4o1QAAAAMAAJ.

[81] parallel_interleave_dataset_op.cc. parallel_interleave_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/v2.2.0/tensorflow/core/kernels/data/parallel_interleave_dataset_op.cc. Accessed October 20, 2020.

[82] parallel_map_dataset_op.cc. parallel_map_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/v2.2.0/tensorflow/core/kernels/data/parallel_map_dataset_op.cc. Accessed October 20, 2020.

[83] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[84] C. Perlich, B. Dalessandro, T. Raeder, O. Stitelman, and F. Provost. Machine learning for targeted display advertising: Transfer learning in action. *Machine learning*, 95(1):103–127, 2014.

[85] J. Polzin. Intelligent scanning using deep learning for mri. https://blog.tensorflow.org/2019/03/intelligent-scanning-using-deep-learning.html, Mar 2019. Accessed December 5, 2020.

[86] prefetch_autotuner.cc. prefetch_autotuner.cc. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/prefetch_autotuner.cc. Accessed October 20, 2020.

[87] prefetch_dataset_op.cc. prefetch_dataset_op.cc. https://github.com/tensorflow/tensorflow/blob/r2.1/tensorflow/core/kernels/data/prefetch_dataset_op.cc. Accessed October 20, 2020.

[88] S. Pumma, M. Si, W.-C. Feng, and P. Balaji. Scalable deep learning via i/o analysis and optimization. *ACM Trans. Parallel Comput*, 1(1), 2019.

[89] pybind11. pybind11. https://pybind11.readthedocs.io/en/stable/index.html, Accessed November 10, 2020.

[90] PyTorch DataLoader. torch.utils.data. https://pytorch.org/docs/stable/data.html, Accessed October 22, 2020.

[91] PyTorch Elastic. PyTorch Elastic. https://pytorch.org/elastic/0.1.0rc2/index.html, Accessed November 24, 2020.

[92] PyTorch tensor. torch.Tensor. https://pytorch.org/docs/stable/tensors.html, Accessed November 10, 2020.

[93] D. A. S. Rao and G. Verweij. Sizing the prize: What's the real value of AI for your business and how can you capitalise? *PwC Publication, PwC*, 2017.

[94] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[95] ResNet-50 CTL. Custom training loop (CTL) implementation for ResNet-50. https://github.com/tensorflow/models/tree/master/official/vision/image_classification/resnet, Accessed October 23, 2020.

[96] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[97] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[98] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[99] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

[100] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop*, volume 62, pages 98–105. Madison, Wisconsin, 1998.

[101] S. Sarkar. A scalable artificial intelligence data pipeline for accelerating time to insight. Storage Developer Conference, 2019. URL https://www.snia.org/sites/default/files/SDC/2019/presentations/Machine_Learning/Sarkar_Sanhita_A_Scalable_Artificial_Intelligence_Data_Pipeline_for_Accelerating_Time_to_Insight.pdf.

[102] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*, 2017.

[103] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, page 19–es, USA, 2002. USENIX Association.

[104] T. J. Sejnowski. *The deep learning revolution*. MIT Press, 2018.

[105] K. Serizawa and O. Tatebe. Accelerating machine learning i/o by overlapping data staging and mini-batch generations. In *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 31–34. ACM, 2019.

[106] Y. Shoham, R. Perrault, E. Brynjolfsson, J. Clark, J. Manyika, J. C. Niebles, T. Lyons, J. Etchemendy, and Z. Bauer. The ai index 2018 annual report. *AI Index Steering Committee, Human-Centered AI Initiative, Stanford University. Available at http://cdn. aiindex. org/2018/AI% 20Index*, 202018, 2018.

[107] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[108] N. Suda and D. Loh. Machine learning on arm cortex-m microcontrollers. *Arm Ltd.: Cambridge, UK*, 2019.

[109] S. Sur, M. J. Koop, L. Chai, and D. K. Panda. Performance analysis and evaluation of mellanox connectx infiniband architecture with multi-core platforms. In *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*, pages 125–134. IEEE, 2007.

[110] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[111] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[112] Tensorflow input pipeline. tf.data: Build tensorflow input pipelines. https://www.tensorflow.org/guide/data. Accessed October 20, 2020.

[113] tf.data.Dataset. tf.data.dataset. https://www.tensorflow.org/api_docs/python/tf/data/Dataset. Accessed August 31, 2020.

[114] tf.data.Dataset.batch. tf.data.dataset.batch. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#batch. Accessed December 2, 2020.

[115] tf.data.Dataset.interleave. tf.data.dataset.interleave. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#interleave. Accessed October 20, 2020.

[116] tf.data.Dataset.map. tf.data.dataset.map. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#map. Accessed October 20, 2020.

[117] tf.data.Dataset.prefetch. tf.data.dataset.prefetch. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#prefetch. Accessed October 20, 2020.

[118] tf.data.Dataset.repeat. tf.data.dataset.repeat. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#repeat. Accessed December 2, 2020.

[119] tf.data.Dataset.shuffle. tf.data.dataset.shuffle. https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shuffle. Accessed December 2, 2020.

[120] TFRecord. TFRecord and tf.Example. https://www.tensorflow.org/tutorials/load_data/tfrecord, Accessed November 25, 2019.

[121] The HDF Group. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5. Accessed December 21, 2019.

[122] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[123] S. Tokui, K. Oono, S. Hido, and J. Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[124] A. Trask, D. Gilmore, and M. Russell. Modeling order in neural word embeddings at scale. *arXiv preprint arXiv:1506.02338*, 2015.

[125] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.

[126] M. Voss, R. Asenjo, and J. Reinders. *Pro TBB: C++ parallel programming with threading building blocks*. Apress, 2019.

[127] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. *Oak Ridge National Laboratory, National Center for Computational Sciences, Tech. Rep*, 2009.

[128] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[129] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. Characterizing deep learning training workloads on alibaba-pai. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 189–202. IEEE, 2019.

[130] C.-C. Yang and G. Cong. Accelerating data loading in deep neural network training. *arXiv preprint arXiv:1910.01196*, 2019.

[131] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[132] B. Zamanlooy and M. Mirhassani. Efficient vlsi implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):39–48, 2013.

[133] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, et al. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, 2013.

[134] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[135] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney. Fanstore: Enabling efficient and scalable i/o for distributed deep learning. *arXiv preprint arXiv:1809.10799*, 2018.

[136] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100. IEEE, 2018.

[137] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156. IEEE, 2018.

[138] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury. Efficient user-level storage disaggregation for deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[139] Y. Zhuang, A. Thiagarajan, and T. Sweeney. Ranking tweets with tensorflow. https://blog.tensorflow.org/2019/03/ranking-tweets-with-tensorflow.html, Mar 2019. Accessed December 5, 2020.

A

APPENDIX

A.1 RESOURCE USAGE

A.1.1 *TensorFlow*

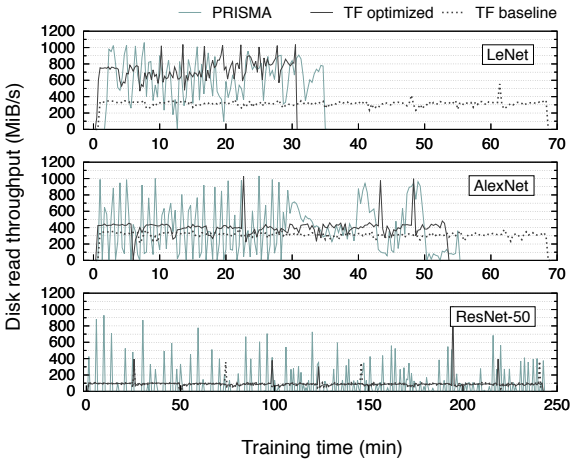


Figure A.1: TensorFlow and PRISMA disk read throughput over time with a batch size of 64.

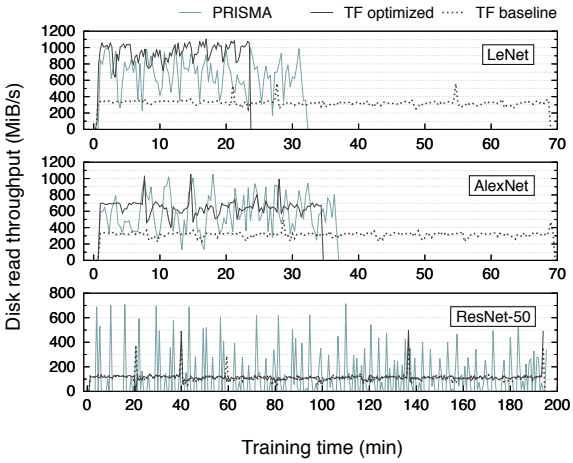


Figure A.2: TensorFlow and PRISMA disk read throughput over time with a batch size of 128.

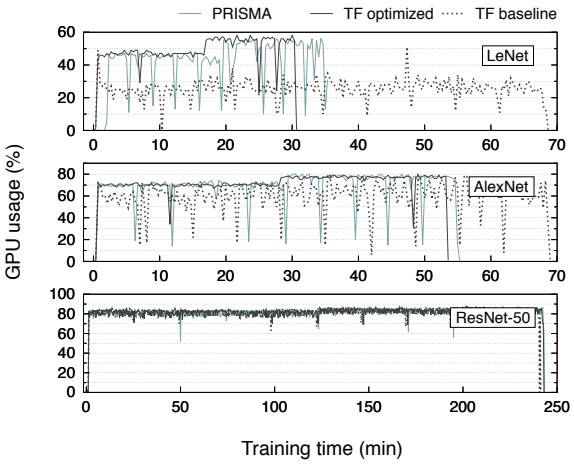


Figure A.3: TensorFlow and PRISMA GPU usage over time with a batch size of 64.

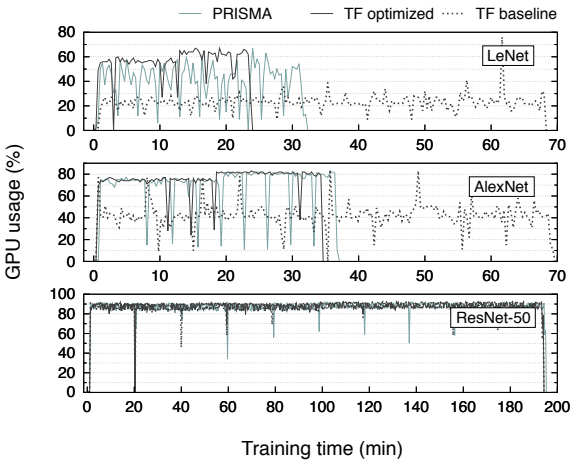


Figure A.4: TensorFlow and PRISMA GPU usage over time with a batch size of 128.

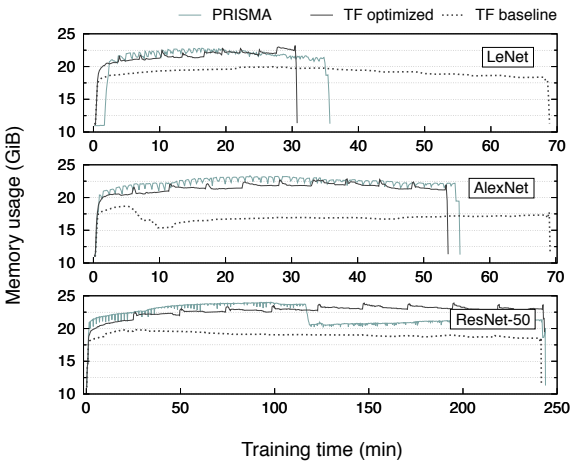


Figure A.5: TensorFlow and PRISMA memory usage over time with a batch size of 64.

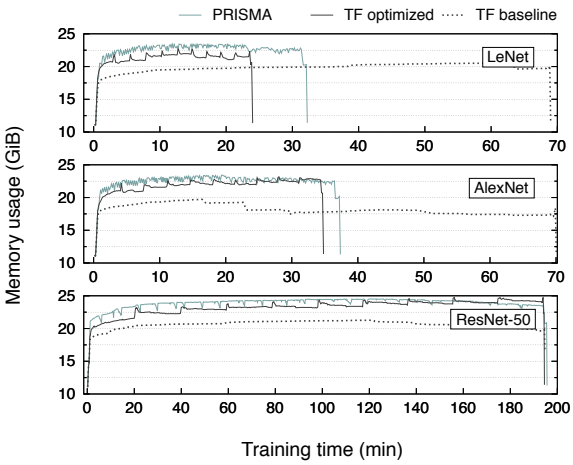


Figure A.6: TensorFlow and PRISMA memory usage over time with a batch size of 128.

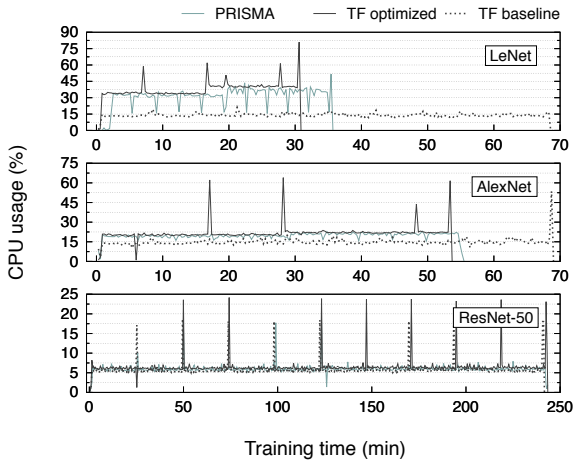


Figure A.7: TensorFlow and PRISMA CPU usage over time with a batch size of 64.

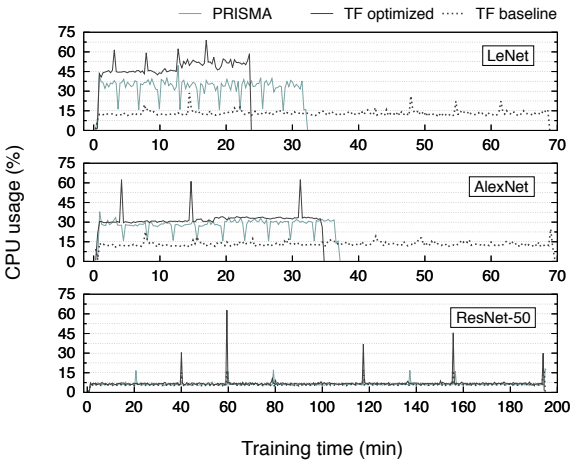


Figure A.8: TensorFlow and PRISMA CPU usage over time with a batch size of 128.

A.1.2 *PyTorch*

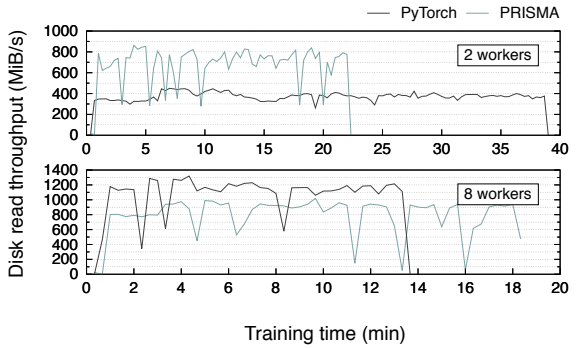


Figure A.9: PyTorch and PRISMA disk read throughput over time with LeNet for 2 and 8 workers.

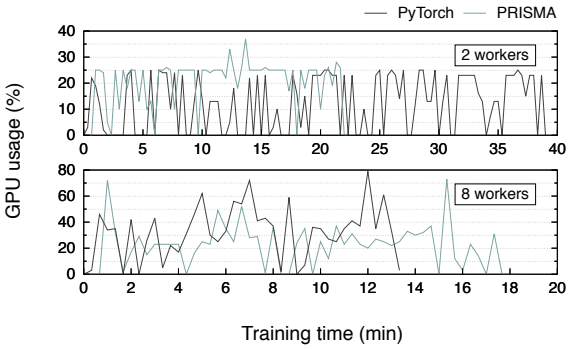


Figure A.10: PyTorch and PRISMA GPU usage over time with LeNet for 2 and 8 workers.

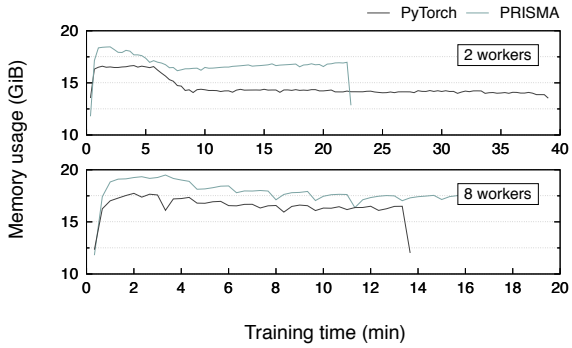


Figure A.11: PyTorch and PRISMA memory usage over time with LeNet for 2 and 8 workers.

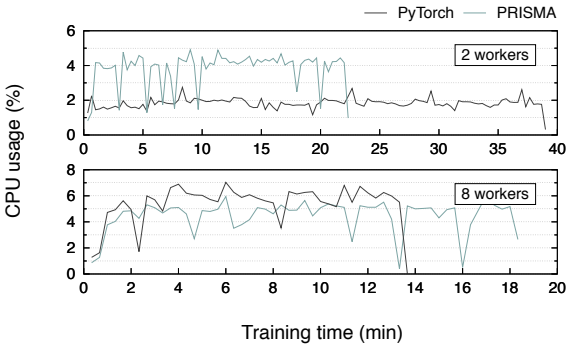


Figure A.12: PyTorch and PRISMA CPU usage over time with LeNet for 2 and 8 workers.

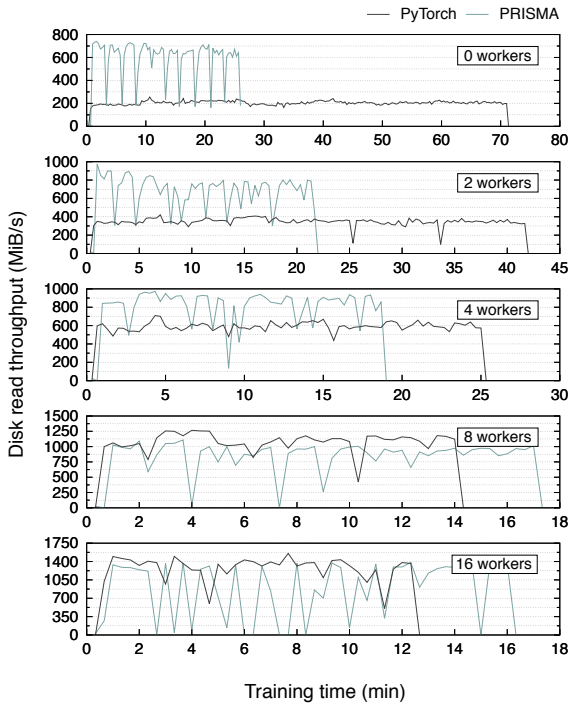


Figure A.13: PyTorch and PRISMA disk read throughput over time with AlexNet.

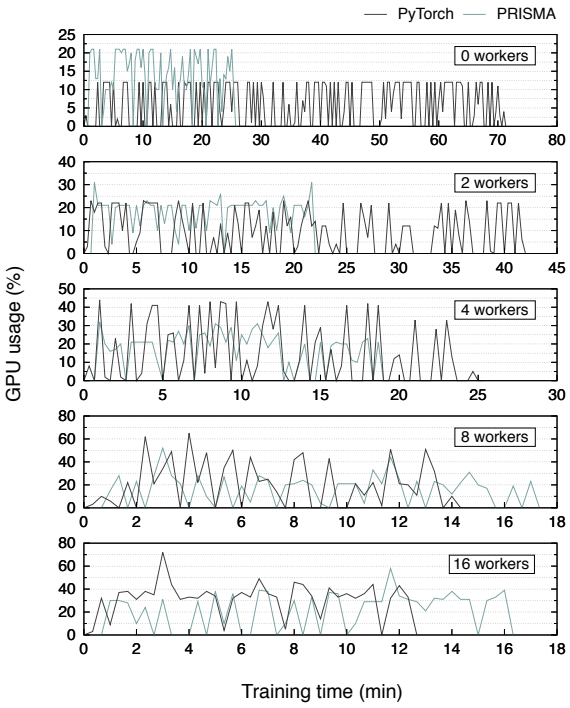


Figure A.14: PyTorch and PRISMA GPU usage over time with AlexNet.

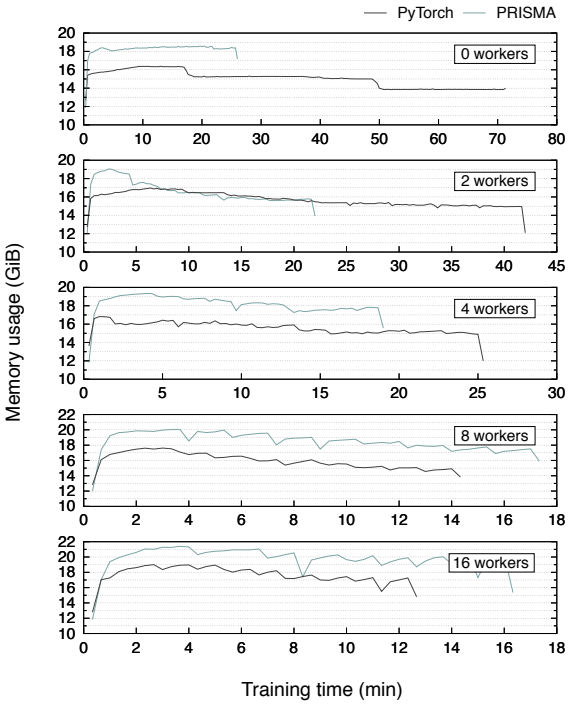


Figure A.15: PyTorch and PRISMA memory usage over time with AlexNet.

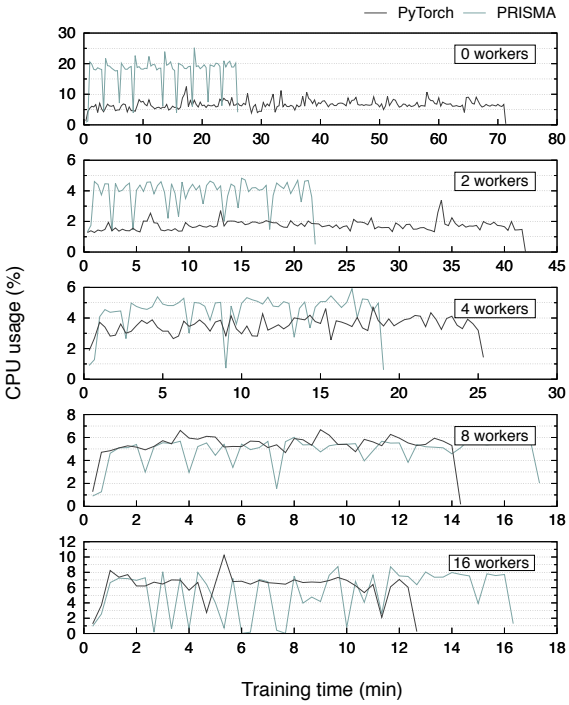


Figure A.16: PyTorch and PRISMA CPU usage over time with AlexNet.

This work is financed by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT), within project UIDB/50014/2020.