Politecnico di Torino

Politecnico di Torino

M.Sc in ICT for Smart Societies - LM 27

A.Y. 2024/2025

July 2025

Design and implementation of a deployment tool for modular DNN inference using ZeroMQ-based GPU-aware communication

Supervisors:

Prof. Carla Fabiana Chiasserini

Dott. Corrado Puligheddu

Candidate:

Dario Antonio Ruta

Summary

Deep Neural Networks (DNNs) are the fundamental structure adopted to provide smart service in a wide range of AI applications. However, DNN-based tasks have high computing requirements, posing huge challenges on their deployment on small and resource-constrained devices such as mobile phones or IoT devices. To address this issue, some solutions consider model compression techniques to limit the computational burden on the device as well as the model memory footprint. Other strategies involve partial or full task offloading towards more powerful computing platforms placed at the edge of new-generation mobile networks (5G-MEC), ensuring low latency and near-zero computing cost for the mobile device. In such context, mobile devices can consider DNN tasks as on-demand services. However, for the fact that MEC platforms are more resourceful than mobile devices, MEC radio and computing resources are limited. Therefore, it is of paramount importance to manage and optimize them to maximize the task exectution rate. In this context, promising results emerge from sharing parts (blocks of layers) of DNN models among similar offloaded tasks. However, coping with parallel model execution in a scenario with high dynamism and strict latency requirements during offloading poses some challenges to be solved.

This thesis work presents BlockFlow, a high-performance deployment and management tool for modular and dynamic DNN inference. It incorporates TensorMQ, a novel contribution in GPU-aware communication for inter-block tensor forwarding at inference time based on ZeroMQ library. A detailed system design and technical motivation of the adopted choices for the practical implementation are widely discussed. BlockFlow provides a high degree of flexibility and adaptability across different computing scenarios such as single-node-single-GPU, single-node-multi-GPU, multi-node, and it can be adopted when offering DNN as service to the users. The performance of the inference pipeline has been tested under different operating conditions and for the most common object detection and object classification DNN architectures. TensorMQ addresses the ping-pong problem between CPU and GPU in single-node-single-GPU and single-node-multi-GPU setups for modular DNN architectures presenting a zero-copy solution, thus reducing the communication

overhead, increasing the pipeline throughput and avoiding redundant data movements while still maintaining a high degree of system modularity and dynamicity. Experimental results in single-GPU setup, considering the popular Resnet50 model, demonstrate how TensorMQ outperform standard ZeroMQ, achieving up to 2.89x faster inference time, 8.30x reduction in pipeline communication overhead in ResNet50 architecture split into 4 blocks. In multi-GPU setups, it allows for up to 5.26x communication latency gains.

"A mia madre"

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Carla Fabiana Chiasserini, for giving me the opportunity to work on one of my most cherished topics. My sincere thanks also go to Dr. Corrado Puligheddu for his valuable advice and for incredible generosity with his time.
I would also like to thank all the people I have met along the way and with whom I have shared unforgettable experiences, both inside and outside the classrooms of the Politecnico. I could not name everyone, but a special mention goes to Tanucci, Nanni, Francesco, Ettore, to whom I wish all the best.
If we are reading this document today, it is undoubtedly thanks to my Papà, whom I cannot thank enough for allowing me to become who I am today.
Finally, I would like to thank Martina for being an integral part of my life.

Table of Contents

List of Tables

List of Figures

Acronyms

SoTA
State of The Art

ASIC
Application-Specific Integrated Circuits

GPU
Graphical Processing Unit

CV
Computer Vision

DNN
Deep Neural Network

CNN
Convolutional Neural Network

LLM
Large Language model

ETSI
European Telecommunication Standard Institute

MEC
Mobile Edge Computing

DOT
DNN for scalable Offloading of Tasks

IPC

Inter-Process Communication

RTT

Round-Trip-Time

PDF

Probability density function

ePDF

Empirical Probability density function

PDI

Path Distribution Index

Chapter 1

Introduction

In recent years, the number of digital services based on Artificial Intelligence (AI) is growing exponentially, bringing a digital revolution of our society. In this vein, the main form of technological progress can be attributed to the large-scale deployment of services based on Deep Neural Networks (DNNs) jointly with Deep Learning (DL) algorithms.
DNNs are the essential paradigm that enabled smart use-cases in a variety of industries such as smart city, precision agriculture, smart health, autonomous-driving and many others.

One of the major factors contributing to this digital revolution has undoubtedly been the increased availability of data to be processed, available computational resources and specialized hardware like Application-Specific Integrated Circuits (ASICs) and Graphical Processing Units (GPUs), effectively enabling the creation of increasingly intelligent and, at the same time, large and complex models.
Indeed, over the past few years model sizes have grown dramatically. This rate of scaling is outpacing Moore's Law, creating a significant gap between the supply and demand for computing power. This poses a serious challenge in the deployment of DNNs in devices with small computational capabilities like IoT and Edge devices. This scenario gets even more complicated if Edge devices are battery powered.
One solution to tackle this problem lies on algorithmic procedures to reduce the computational burden and the memory footprint of the model (e.g., using pruning and quantization techniques). However, model compression techniques usually lead to a reduction in model performance (e.g., accuracy) [1]. Moreover, often the compressed model fails to provide an adequate trade-off between power consumed and required performance. Conversely, the edge device in some cases can take advantage of high-speed wide-band connectivity to offload the full computation or a part of it to a cloud server with more computational power. Nonetheless, applications with strict latency constraints fail to benefit from this solution as they experience large delays due to the network signal propagation to remote hosts

and back. To cope with these issues, the European Telecommunication Standard Institute (ETSI) introduced the concept of Multi-Access Edge Computing (MEC) [2], bringing cloud computing capabilities to the network edge thereby reducing the latency and enhance the overall performance of user applications [3].

In a scenario where AI tasks based on DNNs can be seen as services, the end user can enjoy the above benefits by offloading computer vision tasks to the edge. However it is important to ensure that certain performance levels are guaranteed while respecting systemic constraints.

1.1 Offloading Computer Vision Tasks at the Edge

Among the huge number of DNN based tasks, Computer Vision (CV) tasks may be a great candidate to be offloaded.

Offloading computer vision tasks at the edge-cloud is a promising approach able to reduce latency required for computation due to higher computational capabilities and reduce energy consumption due to the shorter path the data has to travel through the network to reach the remote-cloud. In particular, those who benefit most are small devices that are unable to handle the required calculations due to strict constraints on their hardware.

Offloading computer vision tasks to a Multi-Access Edge computing platform, significantly outperforms traditional remote cloud offloading schemes (more on Multi-Access Edge Computing paradigm can be found in sec. 2.4). In [4], the authors implemented an end-to-end solution, demonstrating the goodness of the 5G-MEC architecture in the context of offloading of computer vision tasks from mobile devices. Specifically, they focused on offloading emotion recognition tasks through CNNs running at the edge. They proved that 5G-MEC architecture improves the throughput of more than 250% and reduces the response time of 71.3% compared to the remote-cloud-based offloading. For small packet sizes, even to a cloud service located in the same country, the RTT could be halved.[5]. The reduction factor is due to a smaller network propagation delays (i.e. fewer number of hops to be traversed) and a more efficient radio communication under 5G-NR with respect to previous RAN generation.

Offloading to the edge-cloud implies the transmission of information over a wireless channel, thus the number of bits transmitted over the channel is a key factor that needs to be taken into account in latency constrained applications. Often, edge devices experience signal quality degradation and transmission bit-rate variability due to their relative movement with respect to the closest Base Station (BS) they are connected to.

For that reason, a variation of the Edge Computing paradigm gained attention

in recent times where the edge device does not offload the full task but only a part of it. This approach refers to split-computing, where the main actors involved (i.e edge-device, edge-cloud, cloud) are in charge of only a part of the computation. In the context of DNN based tasks like CV tasks this approach refers to the *split-DNN*.

Several works take into consideration *split-DNN* as a potential solution to jointly optimize the radio and computing resources and the total energy consumption of the inference pipeline over a multi-tiered interconnected mobile-edge-cloud system to serve multiple DNN based tasks [6]. However, the optimization problem stated above is hard to be solved also due to the *Data Amplification Effect* [7], where quite often in CNNs intermediate feature maps are bigger in size with respect to the input images.

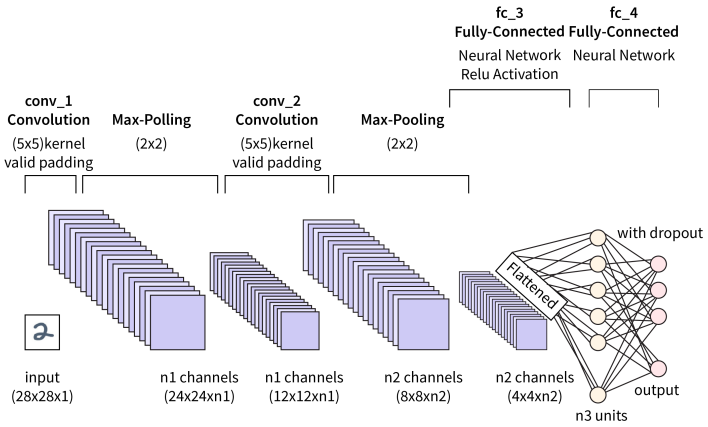

Figure 1.1: Simple CNN structure for Handwritten Digit Recognition

For example, even considering a trivial CNN as the one reported fig 1.1, one can notice that given as input an image of size $[H, W, C] = [28,28,1]$, after the first convolutional layer with 32 filters, kernel size of $[H, W] = [5,5]$, stride $= 1$ and padding $= 0$, the output feature map is of shape $[H, W, C] = [24,24,32]$, increasing the size by almost 32x with respect to the input tensor. [1]

Solutions to these problems are linked again to model compression techniques or the use of encoders-decoders between the entities involved to shrink the amount of bits sent over the channel.

However, in a scenario where the same base model is shared to perform multiple tasks, there is a risk of irreparably compromising accuracy performance in sensitive tasks to them. A possible approach is to enrich the system with a degree of flexibility, thus making it flexible and adaptable to the required context. The

[1] more on CNNs functional is reported in sec. 2.1.2

work at [8] perfectly marries the problem of data amplification effect. Indeed, the authors introduced *Slimmable Encoders* to cope with limited channel capacity in IoT wireless networks.

The *Data Amplification Effect* must be taken into account even in simpler contexts such as traditional Offloading to the Edge Server. There could be cases in which, the split-DNN architecture lies on a single host. In these scenarios, the inference is done by means of a DNN block's pipeline. Hence, it is of paramount importance to ensure efficient communication between the parts of the DNN structure.

Valuable research works such as OffloaDNN [9], focus on this use-case providing a framework able to increase the computing platform efficiency by minimizing the system resources (both radio and computational) in 5G-MEC infrastructure. The main idea is to share dynamically one or more DNNs blocks between one of more than one admitted task at the edge. As a result, the forward pass of a DNN is no longer a static sequence of layers belonging to the same entity but becomes a logical execution path, where different blocks are involved in the inference process depending on the active tasks.

1.2 Thesis Motivations & Objectives

As discussed, OffloaDNN provides meaningful insights in the context of offloading CV tasks at the edge, demonstrating and validating the effectiveness of the proposed solution under different scenarios.

1.2.1 Motivations

When adopting a modular DNN architecture it is essential to consider not only the pure computation time—represented by the cumulative time spent in each block—but also additional overhead factors such as communication latency, inter-block data transfers, and serialization/deserialization operations. Furthermore, computing time itself can vary significantly depending on the current load of each block, introducing complications on accurate latency estimation. Indeed, latency depends on the processing time of each block in the pipeline and on the waiting time spent before being processed by the block.
Mathematically speaking, the average total time T_b spent by a generic tensor t in a generic block b of the pipeline depends on the block load ρ and on the device d hosting the block (e.g, GPU or CPU) and it is:

$$T_b(\rho, d) = T_w(\rho) + T_c(d) + T_{tx} + O_b \tag{1.1}$$

where T_w is the waiting time of the generic tensor t spent waiting to be served and it depends by the block load ρ, T_c is the computing time and it depends on the device d that is hosting the block while T_{tx} is the inter-block transmission time. The term O_b is the overhead component of each inference and it is related to computing operations like serialization and deserialization etc. The load ρ is function of the tensor arrival rate at the block b and its processing time and it is:

$$\rho(\lambda, d) = \frac{\lambda}{\mu(d)}. \tag{1.2}$$

where μ is the average service rate of the generic block b computed as the reciprocal of the computing time.

In Eq.1.1, the inter-block transmission time T_{tx} has quite often been overlooked in the context of branchy and modularly deployed DNNs in the same computing platform. Indeed, an efficient transmission procedure between blocks is of paramount importance since it can directly impact the total computing time, the computing effort and the energy consumption of the pipeline.

Clearly, due to the DNN computation itself and by the presence of hardware accelerators, the development of efficient communication strategies between blocks poses some challenges to overcome, otherwise there could be the serious risk that the advantages in terms of memory and resource reduction would be nullified by prohibitive latency times and excessive overhead.

Moreover, the nature of a distributed application imposes a careful management and orchestration of the active entities involved in the architectural schema.

1.2.2 Objectives

This thesis work has the following objectives:

Objective 1: *Analyze the intra-host modular-DNN paradigm under the scenario of minimizing computational resources given by the possibility to share blocks between different tasks;*

Objective 2: *Research possible solutions related to efficient communication between DNN's blocks instantiated in a computing platform in order to achieve a good trade-off between the advantages offered by the modular architecture itself while avoiding prohibitive overhead;*

Objective 3: *Design and realize a tool able to manage modular-DNN inference architecture applying the main concepts found as outcome of objective 2;*

Objective 4: *Make the tool integrable to an optimization engine that might command the deployment of DNN modules and their connection for each offloaded task;*

Objective 5: *Validate its reliability and performance with both single-GPU and multi-GPU setup.*

1.3 Thesis Contributions

This study presents **BlockFlow**, an *open-source* tool that aims to manage split-DNN architectures intra-node and inter-node providing strong flexibility on DNNs AI services.

The main features of **BlockFlow** are:

- **TensorMQ**: a near-zero communication overhead tensor exchanging paradigm between DNN blocks in presence of hardware acceleration devices[2],
- complete life-cycle management of the Blocks instantiated in the machine,
- real-time system KPI monitoring during inference.

BlockFlow can be widely used in all applications that require a certain degree of flexibility, taking advantage of the possibility of having multiple shared DNN structures.

1.4 Thesis structure

The presented thesis work is articulated into 5 main chapters each of them with specific goals:

Chapter 1: This is the introductory chapter, which serves to lay the foundations for the understanding of the context on which the thesis is based. It presents offloading to the edge procedures with a particular focus for DNN based tasks, discusses its advantages, but highlights some limitations. It analyzes related works on the Literature and finally, it presents the objectives and main contributions of this thesis.

[2]Nvidia GPUs compatible with CUDA library

Chapter 2: It it serves as a theoretical background (sec 2.1 presenting a general overview of the related concepts used in the thesis. Hence, it provide a broader overview of OffloaDNN.

Chapter 3: It aims to present in detail the work carried out for the practical implementation of the tool, discussing the technical and implementation aspects.

Chapter 4: It presents the performance results obtained by the tool and discusses its strengths.

Chapter 5: It concludes the work by briefly summarizing the main answers to the research questions posed and the objectives set. Finally, it presents a brief overview of how the research may be continued.

Chapter 2

Background

2.1 AI & Deep Neural Networks

The term Artificial Intelligence (AI) refers to a set of methods and techniques used to enable machines to solve problems through the use of computer science. According to that definition, AI spans the concepts of Machine Learning (ML), Deep Learning (DL) and finally Neural Networks (NNs), as: neural networks are a type of deep learning models which are subfield of machine learning which, in turn, is a subset of AI (Figure 2.1).

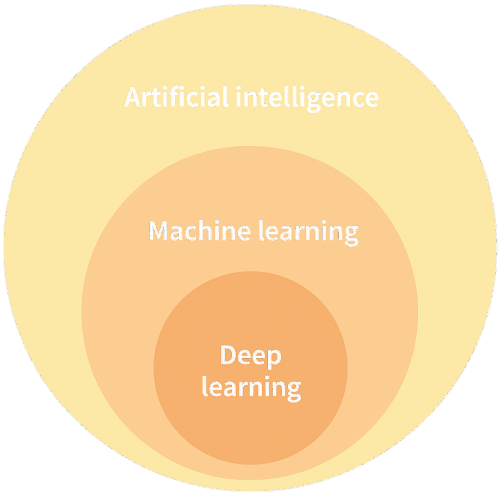

Figure 2.1: AI, ML, DL relationship.

Machine Learning (ML) and Deep Learning (DL) refer to methodologies designed to train models capable of performing specific tasks effectively, thereby enabling

machines to exhibit intelligent behavior.

These methodologies are grounded in the assumption that, for a given phenomenon, there may exist an underlying mathematical relationship between a set of variables (features) and the observed outcome. In other words, given a collection of input data (features) and corresponding output values (labels or targets), a machine learning model aims to approximate the functional relationship that maps inputs to outputs. The process of developing a machine learning model typically involves three fundamental stages: **model selection**, **feature extraction**, and **decision making for prediction**. Since ML models are mathematical in nature, they rely on parameters whose values influence the model's performance. These parameters—either fully or partially—are learned through a **data-driven training process**, in which the model adjusts its internal configuration to best capture the patterns present in the data.
This introduction allow us to formulate a difference between classical machine learning and deep learning:

- **Machine Learning**: the model selection process encompasses the use of a statistical model to be selected among many available, the feature extraction process is handcrafted and the decision making is data-driven
- **Deep Learning**: the model selection process encompasses the use of Neural Networks, the feature extraction process is data-driven and the decision making process is data-driven.

It is important to notice that choosing the most appropriate model and selecting the right set of features can be complex for some problems.
The idea behind NNs is that if a NN is sufficiently powerful and well trained, it will be able to compute the best possible feature extraction which leads to good performance in the decision making stage.

2.1.1 Deep Neural Networks

Neural networks are mathematical structures designed to mimic the human brain, whose main component is the neuron.
In the AI context, the neuron is the base component of a neural network that performs a weighted sum of several inputs and eventually the weighted sum passes through a non-linear function. The neuron's mathematical formulation is:

$$y = \phi\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

or in matrix formulation:

$$y = \phi(w^T x + b) \tag{2.2}$$

where y is the output (scalar) of the neuron, w is the set of weights, x is the input feature vector and b is the bias. The ϕ stands for the non-linearity eventually used after the weighted sum and it is chosen arbitrarily. The most common non−linear functions are: ReLU, Sigmoid, Leaky ReLU etc. each of them with a specific behavior and use case.

Neurons arranged in parallel create a NN layer and layers placed in a sequential way create a neural network structure.

Deep Neural Networks refers to a sequential composition of multiple layers, each of them composed by several neurons, able to work in a complex latent space; the first layer is also called *input layer*, the last one is known as *output layer* while the ones in the middle are called *hidden layers*. In feed-forward Neural networks, each layer takes as input the output of the previous layer. A layer whose neurons are connected with all the neurons of the next layer is called *fully connected layer* or *linear layer* whose mathematical formulation is:

$$y = \phi(Wx + b) \tag{2.3}$$

where $y : (F_{output}; 1)$ is the output vector of the layer, $x : (F_{input}; 1)$ is the input vector, $W : (F_{output;F_{input}})$ is the weight matrix and $b(F_{out}; 1)$ is the bias vector. The output vector of a NN y_{net} given an input vector x is given by the composition of the function f of each layer (eq.2.4 and fig. 2.2).

$$y = (f_L \circ f_{L-1} \circ \ldots \circ f_1)(x) \tag{2.4}$$

As discussed in the previous section, the final goal of each machine learning model is to fit its parameters over a set of data samples and train them to learn the statistical dependency between the input provided and the output desired. In other words, given a set of observations we try to estimate a mathematical function that is able to generalize the observed phenomenon and thus, make prediction when new data are provided. Since Deep Neural networks perform feature extraction by themselves, they are universal function approximator. In practice, they are able to provide good approximation of functions in N dimensions and the accuracy of the approximation depends on: the amount and the quality of data provided, the number of parameters of the NN itself and their relative connections between them and finally the training strategy. The training process is an iterative procedure devoted to retrieve the set of optimal parameters of the NN and it is based on the backpropagation of the error that the neural network makes at each *forward pass*. The error made by the NN is measured by an error function known as *loss function*, that is chosen arbitrarily among several available based on the task the NN is going to be trained for. Afterward, the gradient of the loss function is backpropagated and finally the weights are updated according the chosen optimization logic (e.g, gradient descent, Adaptive moment estimation).

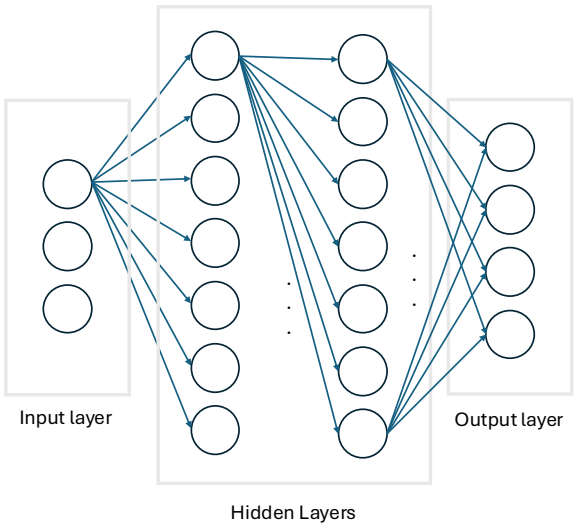

Figure 2.2: Simple Feed-Forward neural network with fully connected layers.

2.1.2 Convolutional Neural Networks

Convolutional Neural networks (CNN) belong to a subset of neural networks that incorporate the convolution operation into their layers. These structures are particularly useful for all those problems whose data presents locality properties such as images, music, timeseries.
In a convolutional layer, each neuron performs convolution over a subset of output values coming from the previous layer's neurons. More generally, the idea behind convolutional layers is based on the fact that each neuron is in charge of extracting some knowledge in a specific spatial area of the data (i.e. set of adjacent pixels in an image, number of consecutive samples in a signal) through multiple filtering stages. This provides several advantages such as:

- the network can work with input data of variable size;
- reduced number of parameters due to less connections between neurons.

The portion of space over which a neuron performs convolution is called *local receptive field* and it depends by the filter size. The output of a convolutional layer

presents one or more than one feature maps that are latent representation of the input features. The number of feature maps in output of a convolutional layer depends on the number of filters used. Mathematically speaking, a generic neuron performs the following operation over a feature map:

$$y_n = w * x + b \tag{2.5}$$

where y_n is the output of the neuron, w is the filter or kernel, x is the corresponding values of the feature map in the same position of the filter and b is the bias term. A visual representation of convolution operation over a feature map is reported in Figure 2.3.

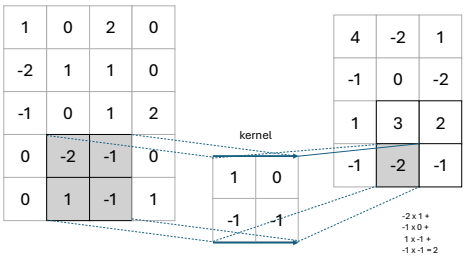

Figure 2.3: Convolution operation with kernel size [2,2].

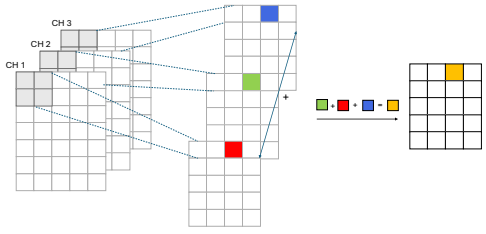

Figure 2.4: Generic Feature map creation procedure.

The image 2.4 illustrates the main steps carried out within a convolutional layer to create a single feature map. Given an input of a generic shape with 3 channels, the filter is slid over the first input channel, creating a temporary feature map. The same operation is repeated on all input channels. Finally, the temporary feature maps are summed element-wise. The parameter that determines how many steps the filter slides is called stride. Jointly with the padding parameter and the kernel size, it determines the shape of the feature map $[h, w]$ as output from the layer. The shape can be computed by as:

$$\text{Output\_size} = \frac{\text{Input\_size} - \text{Kernel\_size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

A set of as many filters as the number of input channels once trained allows to detect one single feature. Subsequently, for F features to be detected a number F of filters needs to be used. In case of F filters, the same set of operations is repeated for each of them. The dimensions of the feature map will therefore be $[N, h, w]$.

In a convolutional layer i the number of learnable parameters is

$$num\_params_i = KxKxCxF \tag{2.6}$$

where KxK is the kernel size, C is the number of input channels and F is the number of filters used.

CNNs used for CV tasks are structures that employ a high number of convolutional layers, each with an adequate number of filters based on the number of features to be extracted. Figure 2.5 shows the detailed architecture of VGG-19 network for object classification.

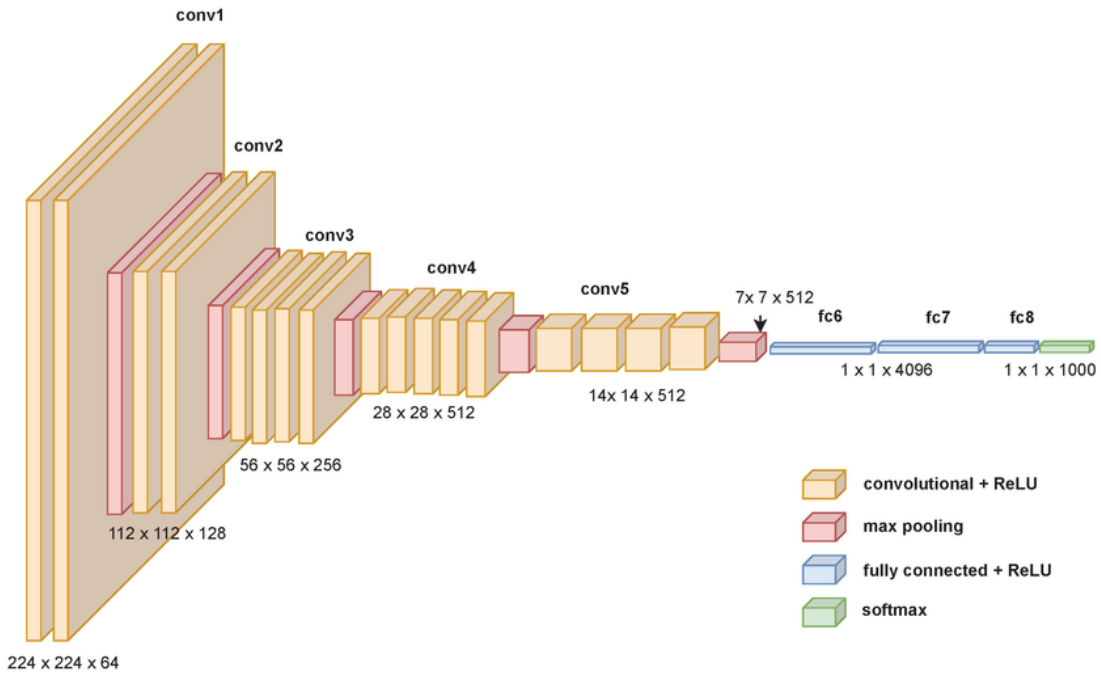

Figure 2.5: VGG19 Architecture [10].

From a practical point of view, most CNNs reduce the spatial size of feature maps and increase their depth (or number of channels) with stride parameter larger than 1 or with pooling operations. As consequence, even in the case of small kernels (e.g., 3×3), the combined effect of multiple layers enlarge the field of view of the final layers over a large portion of the input image. During training, the filters of the initial layers tend to detect low-level features like angles, contours, textures, while deeper layers detect high-level features such as parts of objects, complex semantic structures. For an object classification problem, for example, the backbone of the neural network consists of a set of convolutional layers aimed at feature extraction; on top of the backbone is placed a classification head (linear

layer) preceded by a flattening of the feature maps produced by the backbone. Training of deep CNNs could suffer from the well-known phenomena of vanishing gradient and/or exploding gradient. In the first case, traditional activation functions such as sigmoid, ReLU produce derivatives in the interval (0,1), so the product of gradients across many layers tends to decrease exponentially with the depth of the network. Similarly, if the derivatives become large, gradient explosion can occur, with unstable weight updates. To mitigate these effects, modern CNN architectures introduce structural components such as residual blocks used in ResNet.

Resnet Architectures

ResNets were presented in 2015 for the first time in [11] as Convolutional Networks aiming to tackle the vanishing gradient problem, improve weights stability during training and allow to the creation of deeper architecture thus increasing accuracy performance on image processing tasks. Residual blocks are based on the introduction of skip connection to allow the convolutional layer to only learn a correction instead of the full transformation, leading to a reduced number of parameters used, faster convergence due to the fact that gradients are better conditioned.

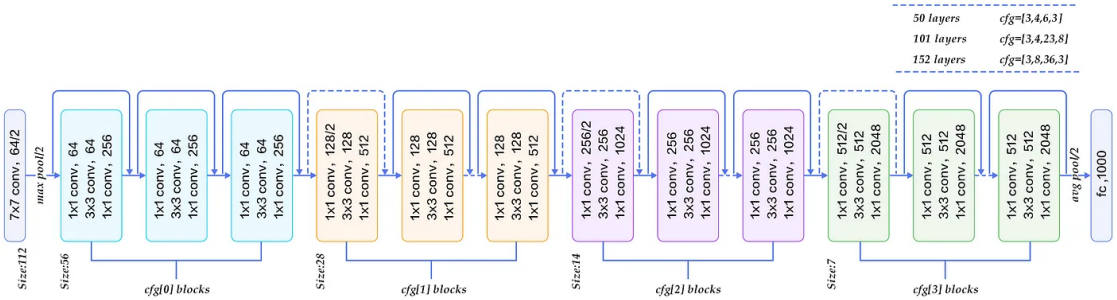

Figure 2.6: ResNet50 Architecture.

The architecture of ResNet50 has **4 stages** as shown in Figure 2.6. Moving from one stage to another, the channel width is doubled and the size of the input is reduced to half. It is available in different versions with different depths: 18,34,50,101,152. The architecture of the stages is the same among all versions but the amount of residual blocks per layer changes. Table 2.1 reports a complete overview about different ResNet structures.

2.2 Deep Learning at scale

Deployment of large models, particularly Large Language Models (LLMs) on a single device could be problematic due to the lack of available memory or computing

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	3×3, 64 3×3, 64 ×2	3×3, 64 3×3, 64 ×3	1×1, 64 3×3, 64 1×1, 256 ×3	1×1, 64 3×3, 64 1×1, 256 ×3	1×1, 64 3×3, 64 1×1, 256 ×3
conv3_x	28×28	3×3, 128 3×3, 128 ×2	3×3, 128 3×3, 128 ×4	1×1, 128 3×3, 128 1×1, 512 ×4	1×1, 128 3×3, 128 1×1, 512 ×4	1×1, 128 3×3, 128 1×1, 512 ×8
conv4_x	14×14	3×3, 256 3×3, 256 ×2	3×3, 256 3×3, 256 ×6	1×1, 256 3×3, 256 1×1, 1024 ×6	1×1, 256 3×3, 256 1×1, 1024 ×23	1×1, 256 3×3, 256 1×1, 1024 ×36
conv5_x	7×7	3×3, 512 3×3, 512 ×2	3×3, 512 3×3, 512 ×3	1×1, 512 3×3, 512 1×1, 2048 ×3	1×1, 512 3×3, 512 1×1, 2048 ×3	1×1, 512 3×3, 512 1×1, 2048 ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 2.1: ResNet Architecture Comparison.

resources. Moreover, in case it could be possible, due to the need of always-up-to-date, well-performing models, the repetitive nature of the training procedure would mean that the entire training process for these structures would be very slow.

Training and deployment of deep learning models at scale present computational and architectural challenges and it necessitates distributed training and/or deployment techniques such as Data and Model Parallelism (Figure 2.7).

Data Parallelism is the simplest form of parallelism. It is based on the replication of the full model over multiple GPUs and dividing the training dataset into independent batches to be delivered to each model copy. Each device performs the forward and backward passes in parallel on its data shard, then they synchronize by aggregating the gradients and finally they update the shared model parameters. The main advantage is simplicity and the communication load between GPUs which is relatively low. On the other hand, there is the strict limitation that the model can fit on a single GPU.

Conversely, **model parallelism** divides the model across multiple devices. In this strategy, each GPU hosts part of the model weights rather than a complete copy. In general, this can be done in two complementary ways:

- **Pipeline Parallelism**: the model is divided *vertically* into sequences of layers.

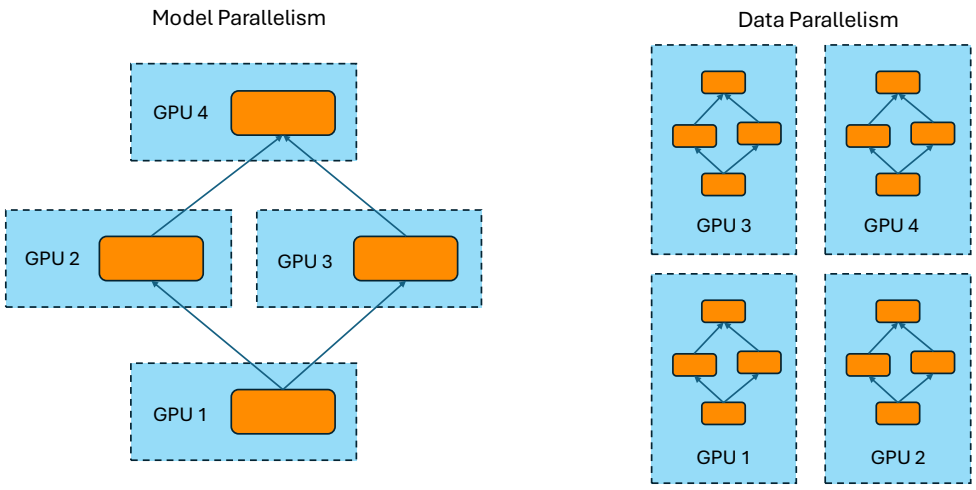

Figure 2.7: Model (left) and Data (right) parallelism conceptual schema.

For example, on 4 GPUs, the first quarter of the layers could reside on GPU 1, the second quarter on GPU 2, and so on. During the forward pass, intermediate tensors are passed from one GPU to the next according to the model flow; during the backward pass, the derivatives are propagated backward along the same path. The model is therefore "pipelined" between GPUs: while one GPU calculates the initial layers for a given batch, another can calculate the subsequent layers of the same batch or a different batch. However, splitting the model into a pipeline introduces temporal inefficiencies that can reduce the total utilization of the device and thus, the system performance too; in particular, this happens when a device remains in an idle state since it is waiting for the previous block to send the intermediate tensor. In addition, overall latency can increase because each batch must go through multiple sequential stages. Efficiency is maximized when the computation times of the various segments are balanced and when optimized intermediate tensor transmission strategies are used, especially for serialization/deserialization and transfer between device memory and host memory.

- **Tensor Parallelism**: This method divides individual operations and tensors within layers. In tensor parallelism, a matrix or tensor is partitioned horizontally across multiple GPUs, each of which computes a portion of the operation simultaneously. For example, in a linear layer $C = A \times B$, the matrix B can be divided by columns into blocks and assigning each block to different GPUs, calculating $(A \times B\_i$ in parallel. Finally, the partial results are combined to obtain the complete result.

2.3 Inter-process Communication (IPC)

Inter-Process Communication (IPC) refers to a set of methods and protocols that enable data exchange between separate processes running on a computer system.

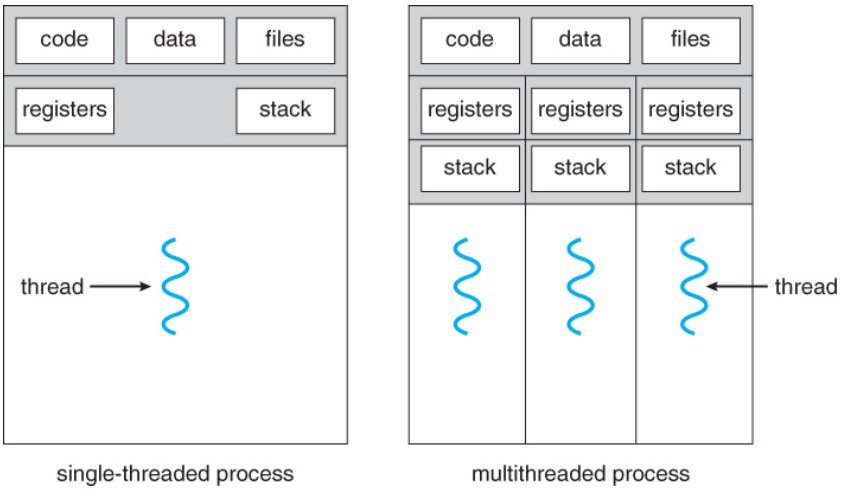

Figure 2.8: Single-threaded and multithreaded processes [12].

As the Figure 2.8 reports, each spawned process has its own address space in memory, opened files and assigned system resources, allowing it to execute instructions independently with respect to other processes. This separation ensures isolation and security: an error in one process does not directly compromise other processes, and a malfunctioning process can be terminated without causing the entire system to crash.
However, this leads to an increased overhead on the spawn of a new process with respect to the launch of a new thread.

IPC techniques can be grouped in two basic models: shared memory and message passing. Communication by means of shared memory is carried out by letting multiple processes access a common memory area and read/write over this space.

On the other hand, message passing techniques let processes communicate by sending messages through communication channels provided by the operating system.
The most common message passing techniques for IPC are:

- *Pipes* which are simple communication channels between two processes. There are two types of pipes: anonymous pipes and named pipes. Anonymous pipes allow one-way information exchange only for parent-child processes and follow the lifetime of the process that created them. In contrast, named pipes can also be used to communicate between unrelated processes and have two-way communication.

- *Sockets*, communication endpoints that can operate both locally (Unix domain sockets) and on the network (TCP/UDP). They allow for bidirectional data exchange between processes on different hosts or on the same host, using various protocols (stream-oriented such as TCP, or datagram such as UDP).

- *Queues* are structures managed by the operating system where processes can send or retrieve messages. The kernel is responsible for storing the messages in order; senders and receivers do not need to be directly connected, reducing the coupling between processes.

2.4 Multi-access Edge Computing (MEC)

Multi-access Edge Computing (MEC) is a technology standardized by the European Telecommunication Standard Institute (ETSI) [2] in 2015 devoted to bring an IT service environment and cloud-computing capabilities at the edge of the mobile network, within the Radio Access Network (RAN) and in close proximity to mobile subscribers. As a consequence, a reduced latency, improved customer experience and a more efficient exploitation of network resources is ensured.

However, it is important to highlight the difference between Edge Computing and Multi-access Edge-Computing.
While Edge-Computing stands for a computing paradigm in which computation is performed close to the data source thus improving latency and bandwidth saving, Multi-access edge computing is a telecommunication concept; a standardized architecture to enable edge computing within mobile networks (e.g, 5G).
Since its standardization and its large-scale deployment, 5G improved end-user connectivity providing faster, more reliable and more efficient communication enabling thus economic growth and competitiveness in the creation of new use-cases in different domains such as smart cities, telemedicine, 4.0 industry etc. In addition to a better use of radio resources and the possibility of having multiple networks

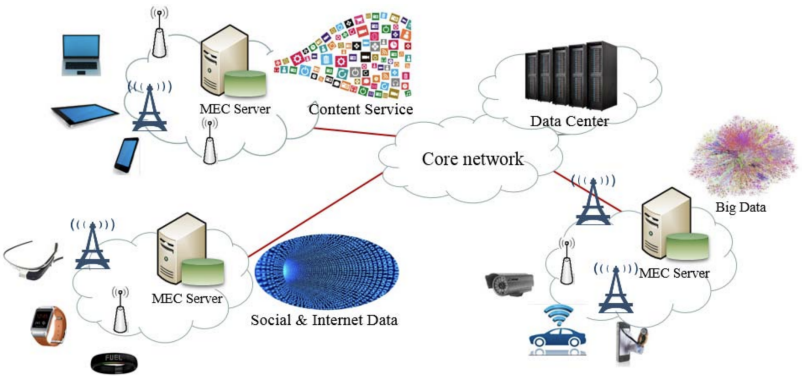

Figure 2.9: MEC overview.

with different requirements coexisting within the same network, 5G leverage more programmable approaches to software networking (SDN) and use virtualization technology within the telecommunications infrastructure (NFV), functions and applications.
More in detail, NFV stands for Network function Virtualization meaning that tasks that were previously carried out exclusively by specialized hardware devices such as routers, firewalls etc. are now virtualized and executed as software components. Indeed, jointly with NFV and SDN, MEC is considered one of the key emerging technology of 5G mobile networks [2].
MEC is based on a virtualized platform, showing an approach that is complementary to NFV. While NFV prioritizes network functions, the MEC framework facilitates the execution of applications at the edge of the network. Each MEC server consists of a hosting infrastructure and an application platform as it is shown in fig. 2.10.

The hosting infrastructure includes the hardware resources (such as the computation, memory, and networking resources) as well as a virtualization Layer. The MEC application platform includes a MEC virtualization manager together with an Infrastructure as a Service (IaaS) controller, and provides multiple MEC application platform services. On top of the MEC application platform, the MEC applications are deployed and executed within virtual machines, which are managed by their related application management systems and agnostic to the MEC server/platform and other MEC applications.
A possible set of MEC applications includes intelligent video surveillance, dynamic traffic management, and real-time environmental monitoring. In the automotive sector, it enables V2X communications and it can support autonomous driving. Moreover, MEC allows local execution of AI models as application (e.g., CNN for computer vision) enabling also devices with limited computing power to perform tasks that they would otherwise be unable to perform.

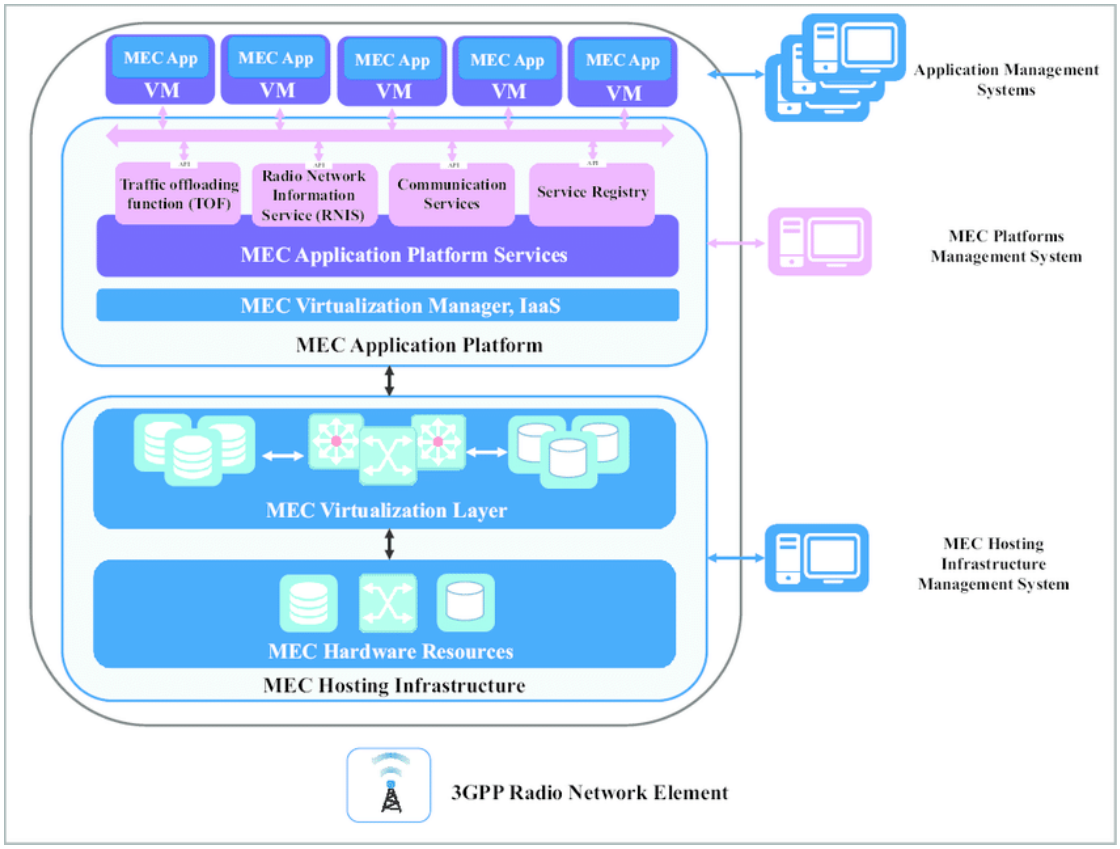

Figure 2.10: MEC platform Schema [13].

2.5 OffloaDNN [9]

In a 5G-MEC, end devices such as mobile phones and IoT devices can offload computationally intensive tasks, such as those related to computer vision, to nearby edge servers. This approach allows end devices to overcome hardware limitations and achieve lower latencies than the central cloud. However, it is important to consider that edge server resources are also limited. Although they offer greater computational power than end devices, edge servers must simultaneously handle several tasks from different devices.
In OffloaDNN [9], the authors formulated a weighted-tree-based heuristic to solve the DNN for scalable Offloading of Tasks (DOT problem). It provides a novel contribution on the jointly optimization of the:

- Edge-Cloud computing and radio resources,
- Offloaded task admission rate,

- Optimal structure of the DNN used to execute the offloaded task.

OffloaDNN brings a new perspective considering DNN structures serving different Computer Vision tasks as set of Blocks, each of them made by more than one original layer of the DNN. However, the main contribution lies in considering the possible structural correlations between the various DNNs required for the execution of different CV tasks.
Indeed, during training, hidden layers of a DNN get specialized into the extraction of different feature levels. Specifically, in CNNs, feature extraction process sees early layers devoted to the construction of a visual basis by learning how to detect low-level features such as corners, edges, colors etc. Final layers instead, uses the extracted information from earlier layers to perform high-level decisions such as object recognition, object classification. This hierarchical feature representation forms the foundation of transfer learning, a technique designed to accelerate the training of DNNs by leveraging pre-trained models. Specifically, the early layers of a pre-trained NN, which capture general low-level features, are retained (i.e., their weights are frozen), while only the later layers are fine-tuned on the target task. This approach not only reduces training time but also improves performance, especially when labeled data for the new task is limited [14].
OffloaDNN exploits the structural redundancy among different DNNs serving different tasks to reduce the Edge Server memory usage and increase the task admission rate to the Edge Server.
The main intuition is to **leverage DNN layers that can be shared at the edge among DNNs serving different tasks**.

In real-case scenarios where the edge device asks for a CV task offloading with specific requirements such as accuracy or maximum admissible round-trip-time, OffloaDNN should provide the most-valuable tradeoff that minimizes the edge server resources while meeting the constraints imposed by the end-device. In these cases, fine-tuning of task specific layers must be executed.
The second intuition is **to choose which layers to share and which to fine-tune based on the task needs**.

Lastly, if the structure is accurate enough, OffloaDNN allows for **structured pruning** to one or more than one block in order to reduce the model memory footprint as well as the processing time.

For a more complete understanding, the main innovations proposed are depicted fig 2.11 considering an object classification problem. In the left sub-picture, it shows how the sharing of the first 2 blocks would result advantageous, in terms of the total blocks deployed (6 total blocks vs 8 total blocks). The center sub-picture shows how to actually improve the accuracy performance for the task *model train detection* by fine-tuning only target specific layers. The latter (right), shows how memory footprint and latency of the structures can be further optimized relying

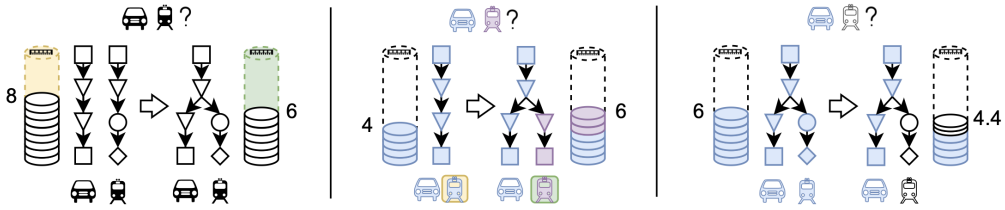

Figure 2.11: OffloaDNN's innovations. (fig.1, p.2 at [9])

on pruning.[1]

OffloaDNN architecture

OffloaDNN can be seen as an application running on the Edge Computing Platform, able to intelligently optimize the system resources, maximize the offloaded task admission rate while matching the task related constraints such as minimum accuracy and maximum round-trip-time.

For the sake of simplicity, we skip the mathematical formulation of the DOT problem and its related heuristic (DOT solver), that the author presented in sec. IV of [9], but rather focus on the architecture and the workflow reported in fig. 2.12. The full way of functioning of OffloaDNN can be summarized into 7 main steps:

Step 1: *Task admission request;*

Step 2: *DNN availability, network status and computing status check;*

Step 3: *DOT solution;*

Step 4: *Radio and computing resource allocation & DNN block selection;*

Step 5: *Activation of the selected DNN blocks;*

Step 6: *Task admission rate notification;*

Step 7: *Task result collection (end-device side);*

To offload tasks, mobile devices first send a task admission request to the OffloaDNN controller **(step 1)**. Afterwards, the controller retrieve from the Virtual

[1]Single-Shot pruning. More on model compression techniques at **??**.

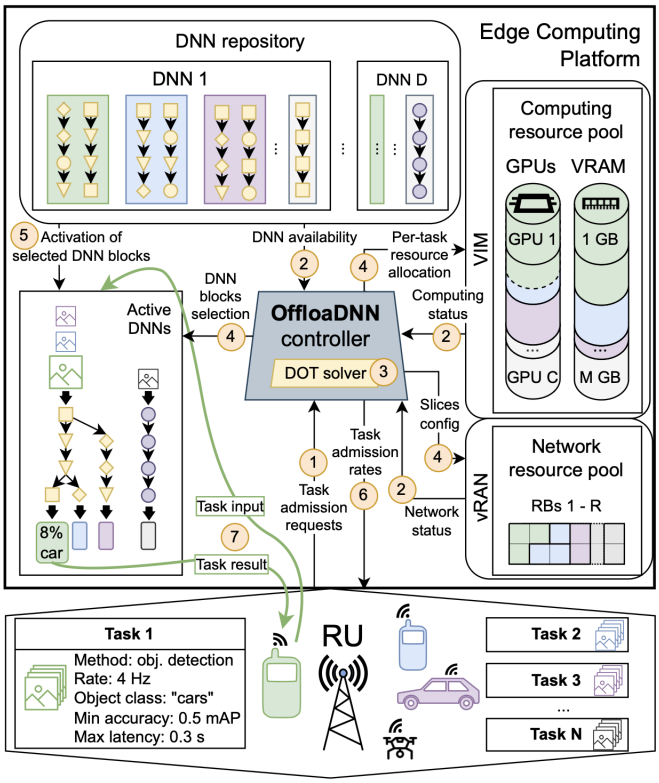

Figure 2.12: OffloaDNN architecture and workflow (fig.4, p.4 at [9]).

Infrastructure Manager (VIM) and the vRAN information about the system status such as the available DNN blocks from the DNN repository and their resource requirements, as well as the current computing resource available and radio resource capacities **(step 2)** and then runs the DOT problem solver **(step 3)**. Once the DOT solver finds a solution, the controller proceeds to allocate the necessary radio and computing resources **(step 4)** and deploys the selected DNN blocks for the tasks that are about to be admitted **(step 5)**. After that, the controller informs the mobile device about the admitted task rates at which their tasks have been accepted **(step 6)**. Finally, the devices can start sending input data and receiving the processed results **(step 7)**.

OffloaDNN results

OffloaDNN aims to tackle the joint optimization of radio and system resources by exploiting the structural correlation about 2 or more DNN structures serving offloaded CV task at the edge. It exhibits significant gains in terms of memory resource savings when compared to previous State-of-the-Art (SoTA) optimizer

such as SEM-O-RAN [15]. In particular, it reduces the computational burden of 77.3% and the average memory occupancy of 82.5% as well as the radio resources of 4.4% while increasing the offloaded tasks of 26.9%.

Chapter 3

BlockFlow

This Chapter presents **BlockFlow**, an open-source tool that aims to deploy, manage and serve modular inference pipeline intra and inter-host.
BlockFlow specifically targets application services based on DNNs, ensuring a reduced end-to-end latency between the hosting machine and the client. Moreover, it presents **TensorMQ**, a novel contribution for a low-overhead tensor transmission in modular DNN architectures (Sec. 3.4).

In the following sections are presented the System Requirements (Sec. 3.1) that **BlockFlow** has to fulfill, followed by the main Technical Challenges to be solved (Sec.3.2).
Sec. 3.3 reports the tools used for the **BlockFlow** realization, explaining the reasons why they were chosen while sec.3.5 provides a careful and detailed explanation about the system design and the workflow.

The source code is available at `https://github.com/darioruta/BlockFlow`

3.1 System Requirements

In a context where an application needs to provide service to remote users through the network (e.g, Offloading through 5G), **BlockFlow** must be in charge of deploying and maintaining the required software infrastructure. More specifically, in a context where an application needs to fulfill DNN task based, it is fundamental that the presented tool pursues the following requirements:

Requirement 1: receive as input the set of DNN structures to be deployed from an optimization engine (e.g., OffloaDNN) as well as the admitted tasks to be served and their optimal paths f;

Requirement 2: selectively deploy as processes the blocks needed to respect 1-to-1 the optimal solution about DNNs structures coming from the optimization engine;

Requirement 3: create logical forward passes that respect 1-to-1 the optimal paths for the admitted tasks by establishing an efficient data transmission channel between the various active blocks;

Requirement 4: selectively terminate the unneeded blocks to free-up system resources;

Requirement 5: monitor in real-time the system performance.

3.2 Technical Challenges

The realization of the tool with the requirements reported above (Sec. 3.1) includes several technical issues to be solved.
A high level idea of the thesis contribution is depicted in Figure 3.1 where in both cases, each block (right) or more generally each DNN (left) is deployed as a process.

In stark contrast to multi-threading, multiprocessing applications require Inter-process communication (IPC) techniques to let processes exchange data, usually coming with additional overheads in terms of communication latency, complexity and efficiency.
Since DNNs benefit from hardware acceleration to speed up the calculation during training and inference stages, in case of pipelined models, IPC latency overheads become even more pronounced. Without the necessary precautions, the transmission of intermediate tensors between blocks in the pipeline is CPU-bounded and could be highly inefficient both in terms of total inference latency and energy consumption. Figure 3.2 illustrates the *ping-pong* effect between RAM and VRAM during a

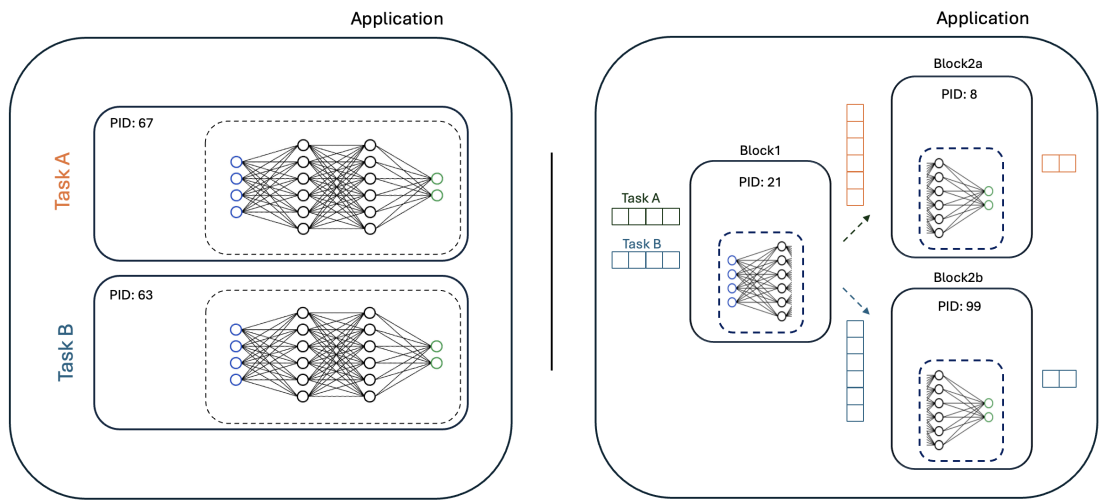

Figure 3.1: (left) Traditional DNN deployment for each admitted task, (right) modular DNN architecture with shared block.

generic tensor forwarding for 2 distinct processes operating on GPU. Red arrows show the data flow across the pipeline.

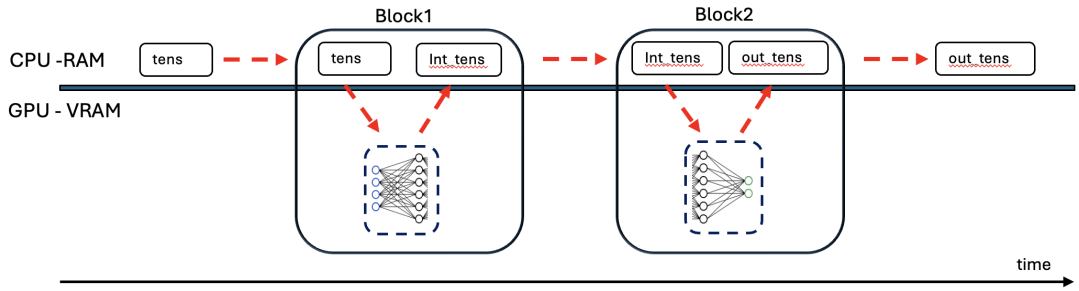

Figure 3.2: Ping-pong effect between RAM and VRAM.

The main challenges to solve are related to the creation of:

- a **light-weight tensor and GPU-aware IPC** to cope with the ping-pong problem improving communication efficiency and thus reducing latency;
- a **modular architecture** with an high degree of flexibility.

Although some existing solutions help to scale Deep Learning application they do not offer a good degree of flexibility especially during inference stages. For example, PyTorch Pipeline [16] only accepts *nn.Sequential* as input model and thus

the network topology becomes static: as soon as a small update in the network topology is needed the full pipeline must be turned-off and a secondary service should rewrite a new one implying service discontinuity for the clients.
As a result, use-cases with high modularity requirements must rely on general purpose communication frameworks which are known to be inefficient for tensor based communication data-flow generating the *ping-pong* effect in presence of hardware accelerators. In this direction, works like *Pytorch RPC* [17] takes into account a communication aware messaging strategy specifically targeting the DL applications. However, *Pytorch RPC* only provides high level APIs specifically targeted to the implementation of training applications in distributed environments. Moreover, it does not perfectly suit the goal of having a fully modular and branchy inference pipeline that can vary according to the system updated of our system optimization engine (i.e the distributed model training pipeline is created at the beginning and once interpreted, the script lives as it is until the end).

3.3 BlockFlow used tools

In this section, the main software tools used to develop **BlockFlow** are listed. For each tool, its main features are presented as well as the role it plays within the system.

3.3.1 ZeroMQ

ZeroMQ [18] is a high-performance messaging library designed for concurrent or distributed applications.
ZeroMQ enables the transmission of messages between thread or processes, in the same or even across different machines, by leveraging in-process communication mechanism (e.g, shared files) or TCP. The term "zero" is derived from the fact that communication between two entities occurs without the use of message brokers. ZeroMQ's philosophy emphasizes simplicity of the library itself; it benefits from low latency, zero licensing costs and simple administration. The added value of this library is that, through high-level calls it is possible to create different topologies between the various entities involved in a simple and efficient way and with different communication patterns, making inter-process communication as simple as inter-thread communication.
ZeroMQ provides different communication patterns:

- REQ/REP: Request/Response paradigm,
- PUSH/PULL: Producer/Consumer paradigm,
- PUB/SUB: Publisher/Subscriber paradigm,

- ROUTER/DEALER: Asynchronous Master/Slave paradigm with built-in routing capabilities.

Moreover, ZeroMQ is a multi-programming-language library, supported in `C`, `C++`,`C#`, `Java`, `Python`, `Ruby`, `Dart`, `GO` etc.

Among different general-purpose communication frameworks like gRPC, ZeroMQ was chosen for its lightness, efficiency and flexibility to be suitable for different communication contexts, supporting different communication paradigms[19].
Several works involve the use of ZeroMQ to ensure a high-performance and light-weight communication in parallel and/or distributed applications. For example, in [20], the authors presented a novel methodology to accelerate the processing of video live stream from IP cameras offloaded to remote devices. The work enforces the use of distributed and multiprocessing architecture by using ZeroMQ as a lightweight protocol to let the remote devices communicate in pipeline. Moreover, ZeroMQ lies as high performance networking library used for the implementation of RDA3 at CERN [21]. Compared with previous versions of RDA, RDA3 scales much better and can handle high data loads and even bursts of requests.
TensorSocket [22] uses ZeroMQ as lightweight communication library in the context of multiprocessing based DNN application. However, the use-case presented in the work, takes into account the training stage. The main contribution is the reduction of the CPU usage during training by sharing the same data loader to multiple training processes (like different models or configurations) and thus reusing the same input pipeline. It uses ZeroMQ to perform Inter Process Communication.

ZeroMQ is integrated into BlockFlow to facilitate IPC. Specifically, it is employed to transmit both control and tensor-related data between distinct blocks that execute a logical forward pass.

3.3.2 PyTorch

PyTorch is an open source deep learning framework used to create, train and develop deep learning models. PyTorch offers a vast library of predefined modules such as tensors, network layers, optimizers and supports a wide range of use cases—from computer vision to natural language processing, from generative models to reinforcement learning. Furthermore, the official repository `torchvision.models` gives the possibility to download the most common neural network structures already pretrained. Among its advantages, the added value of PyTorch is its close connection to Python, allowing developers to write intuitive and readable code and simplify the prototyping and experimentation process.
PyTorch models can also be run in non-Python environments, helping to fill the gap between research prototypes and production implementation.
Indeed, TorchScript [23] allows for high model portability across different platforms. More in detail, it enables the export of trained and developed PyTorch models

from a Python environment towards environments where Python is not available, such as C++.
TorchScript facilitates three fundamental functionalities in this domain:

- optimization for production environments,
- optimized model serialization,
- Increased performance through computational graph optimization.

The creation of a TorchScript model file may be achieved through two different methods: **Tracing** (`torch.jit.trace`) and **Scripting** (`torch.jit.script`).
A Traced TorchScript model registers the operations done during a generic forward pass with an input example. Consequently, possible model dynamicity (network with more branches to be taken based on some input specific magnitudes) is lost since the export procedure depends on the input provided at that time. Conversely, a Scripted model performs a static code compilation and thus it can handle model dynamicity. This is particularly useful for those NNs structures like Recurrent Neural Networks or Branchy−Nets.

PyTorch is used in BlockFlow to export, run and deploy the Blocks.

3.3.3 CherryPi

CherryPi [24] is a minimalist web framework written in Python that allows the development of web applications, enabling server-side code to be written in a similar way to a normal Python program. CherryPi incorporates a thread-pool HTTP/1.1 server, handles HTTP requests in a multi-threaded manner and provides built-in tools such as flexible configuration, session management, caching and authentication. CherryPi is particularly well suited for:

- Rest API implementation, due to the ease with which URIs and various HTTP methods are handled;
- light−weight web applications realization;

In this work, CherryPi is used to create a REST API that will serve as an entry point for system updates.

3.3.4 Pickle

Pickle is a Python module that implements binary protocols for the serialization and deserialization of Python object structures. Serialization (pickling), is the process of converting an object in memory into a stream of bytes that can be saved

to a file or transmitted over a network. Conversely, the deserialization (unpickling) involves the reconstruction of the original object from the byte stream. This serialization/deserialization procedure is done every time information messages are delivered through ZeroMQ.

The pickle library supports multiple versions of the protocol, numerically identified from 0 to 5 (the larger the prot. version the newer is the serialization algorithm). This thesis work used the protocol number 5.

3.3.5 Psutil

Psutil [25] is a cross-platform Python library for accessing information about running processes and system resource usage.
In practice, Psutil offers the possibility to retrieve values such as CPU usage, memory usage, both at the system level and in a more fine-grained way for each individual process by providing the PID.

In this case, Psutil is used to create a resource monitoring system through the use of an entity dedicated to this purpose.

3.3.6 Python NVIDIA Management Library

PyNVML [26] (Python NVIDIA Management Library) is a Python module that allows to monitor NVIDIA GPUs directly from Python scripts.
In practice, it is a Python wrapper for the `NVIDIA Management Library`. PyNVML allows to analyze the status of all GPUs equipped on a computing platform and monitor metrics such as GPU load, memory usage, temperatures, driver status, clock levels, etc.
This has made it possible to integrate the collection of GPU's performance metrics within the application code itself for a detailed and fine-grained analysis of each component of the system (i.e., blocks, dispatchers).

PyNVML is used with the purpose to carefully assess how the various system components occupy the GPU computing resources.

3.3.7 Nvidia nvprof

Nvidia `nvprof` [27] is a command-line tool from the CUDA Toolkit that collects execution data for applications running on NVIDIA GPUs. It allows analysis of CUDA kernel performance, memory transfers between host (CPU) and device (GPU), and CUDA API calls. It offers the possibility to log or visualize the data collected during testing.

In this work, `nvprof` was of crucial importance in evaluating memory transfers between the various blocks of the pipeline for each single inference.

3.3.8 Taskset & Numactl

Both of the tools presented in this section were used to ensure the reproducibility of experiments and optimize the use of computational resources in multi-core and multi-socket architectures. In practice, these tools were used to control process affinity and ensure that the tests launched actually operated in the target cores in order to provide clear explanations for the output results.

`Taskset` is a command available in Linux operating systems, used to set or retrieve the CPU affinity of a process. Using `Taskset`, it is possible to specify the set of cores on which the launched process can run.

Likewise, NUMA control (`numactl`) is a Linux command line tool that allows to run processes with a specific scheduling and memory allocation policy for NUMA (Non-Uniform Memory Access) architectures. In a more simplified way with respect to Taskset, `numactl` allows to directly bind a running program in a target NUMA node by setting the parameter `cpunodebind`. Moreover, for multiprocessing appplications like BlockFlow, it forces the spawn of new processes over the same NUMA node automatically.

3.4 TensorMQ: tensor-aware inter-block communication via CUDA-IPC and ZeroMQ

This section presents **TensorMQ**, the main contribution of this thesis work in line with **Objective 2** reported in sec 1.2.2 and in view of what has been said in the previous section (3.2).
It uses ZeroMQ to deliver information-based communication messages between blocks with low overheads and good adaptability to different inference contexts.
For a generic tensor t, TensorMQ exploits CUDA IPC jointly with ZeroMQ to deliver tensors between blocks. More specifically, considering 2 blocks (e.g., *block1*, *block2*) and a generic tensor t_{b1} coming out from block1, TensorMQ solves the ping-pong problem by:

- generating an *IPC handle* (i.e GPU memory region address where the tensor lives) through the `cudaIpcGetMemHandle` [28] enriched with some additional tensor specific meta-data
- forwarding information rich message to the next block (tensor-handle & meta-data).

Upon receiving the message, *block2*, maps a new empty storage torch object and reconstruct the tensor using `cudaIpcOpenMemHandle`.
As consequence, once the tensor enters the system, it never leaves the GPU, significantly increasing overall pipeline latency performance.
Figure 3.3 clearly shows the workflow. More detailed information about the handle creation and tensor reconstruction procedures as well as the handle structure can be found in sec.3.5.

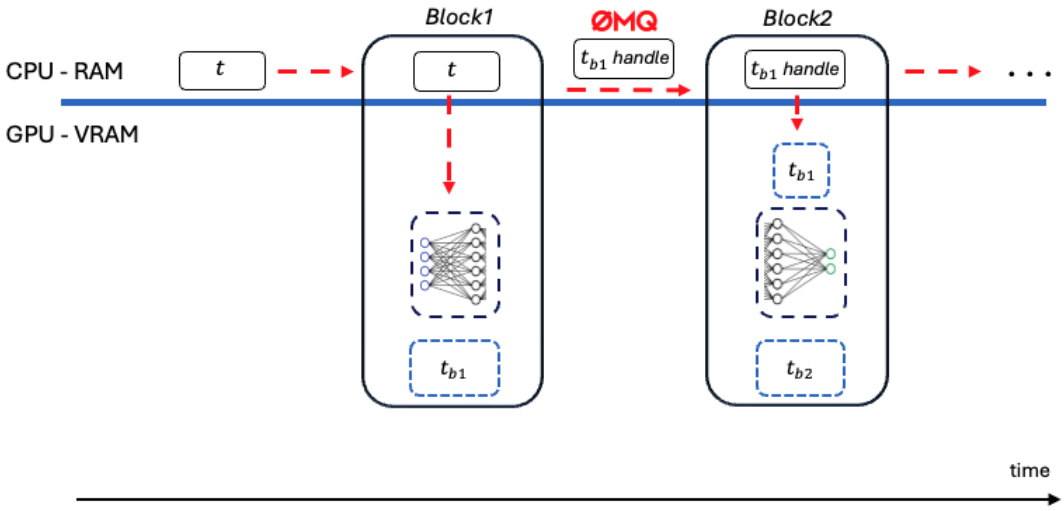

Figure 3.3: IPC handle deliver through TensorMQ.

3.5 Architectural Design

According to what presented in the system requirements in sec 3.1, BlockFlow has a modular architecture able to satisfy the system's needs. In this section the design schema is presented.

Figure 3.5 provides a visualization of the components in the system architecture. The main characters are:

- REST API,
- Manager,
- Controller,
- Block (generic),
- Dispatcher.

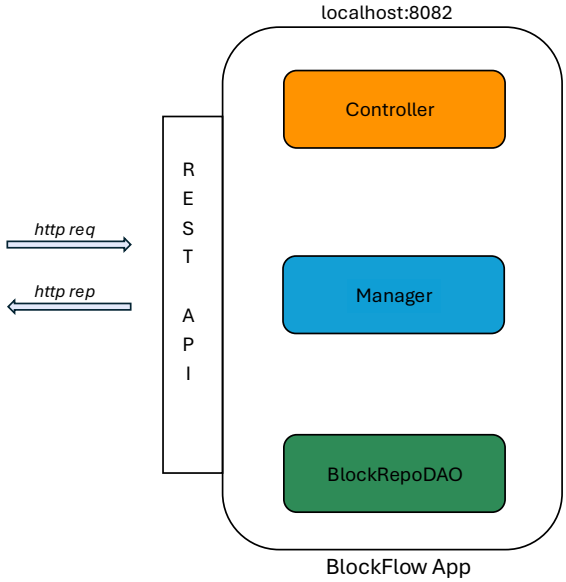

Figure 3.4: BlockFlow application components.

The first 3 elements of this bullet list belong to the Control Plane, responsible for managing, orchestrating, and configuring the modular DNN architecture, respecting 1-to-1 the system updates provided by the resource optimization engine.
In particular, the system control entry-point is a REST API implemented in CherryPi, exposing some CRUD methods to manage the system such as selective block pinging, system updates notification and termination of one or more blocks.
The *Manager* is a python object (class) in charge of handling the full block's life-cycle as well as keeping track about the processes alive. When a new *system update* arrives, the *manager* instance is called and it spawns as processes the set of new blocks to be deployed and selectively terminate the unneeded ones. In addition to that, it periodically performs health checks to the active blocks by sending ping messages through the controller.
The *Controller* is a python object devoted to deliver control messages with ZeroMQ to the various active blocks in the systems. It serves the needs of the manager offering the possibility to notify block specific messages such as ping, store and termination messages as well as health-status messages for monitoring purposes. It operates in a synchronous way and it incorporates a simple block fail detection

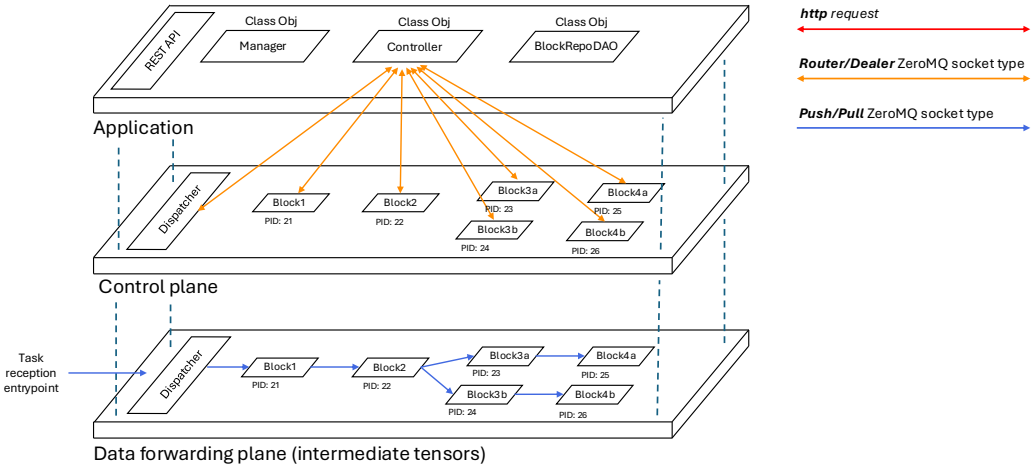

Figure 3.5: BlockFlow application planes.

technique.
The Data Plane of BlockFlow is split into two logical topologies: one devoted to the distribution of control messages while another devoted to the distribution of intermediate results between blocks in the pipeline.
ZeroMQ offers several degrees of flexibility in terms of communication patterns to be implemented as discussed in sec 3.3.1. Among the possible alternatives, the ROUTER/DEALER pattern perfectly suits the role needed to deliver control messages for the following reasons: it has an explicit entity handling engine, it can work in a synchronous and asynchronous way at the same time, only the ROUTER element occupies a TCP port in the host machine. As can be seen in Figure 3.5 layer 2, control messages flow over a star topology. Once deployed, each block establishes a connection with the controller registering itself as a DEALER and notifying its readiness.
On the contrary, intermediate results between blocks flows over an asynchronous PUSH/PULL chain of connections. The PUSH/PULL communication paradigm provided by ZeroMQ brings the logic of the Producer/Consumer architecture allowing a loosely-coupled message exchange between blocks. It is straightforward

to notice that each block is in charge to handle two communication channels, the *control channel* and *inference channel* based on the two different data types they carry. For that reason, each block manages the incoming messages with two separate threads.

3.5.1 Block Design

The main block's software components for a generic Block X are represented in Figure 3.6.

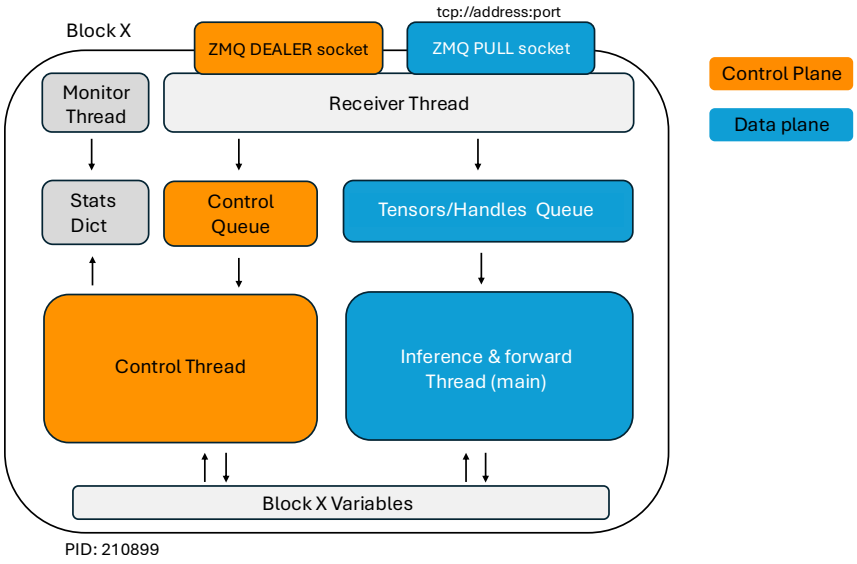

Figure 3.6: Block Components.

Each block launches three or possibly four threads at startup, each with specific tasks. Messages are received by the *receiver thread*, which polls incoming messages in both the channels and split them into the right message queue waiting to be processed.

The Control thread is the actor devoted to the execution of control messages received from the Controller. It performs GET operations in the Control Queue with a blocking behavior (if the queue is empty, it remains blocked on that instruction

line thus reducing the CPU utilization).

The Inference & forward thread (main thread) is the most important block component. It iteratively GET tensors/handles from the dedicated queue and perform inference plus forwarding of the intermediate result to the next block. A detailed workflow of the Block functioning is reported in sec 3.6.

The Monitor thread collects process specific stats with resolution of $250ms$ through the use of dedicated libraries such as *pynvml* or *psutil*. This thread can be dynamically turned on or off through a dedicated control message. However, it is essential to keep it up and running to provide useful insight during testing procedure and performance assessment. Notice that each block does not perform any routing decision since routing information of each tensor comes with it as metadata. They operate in the data plane forwarding each tensor or the tensor IPC handle if TensorMQ is active.

3.5.2 Dispatcher Design

The Dispatcher acts as an entry point for the entire block pipeline, exposing a TCP port through which it can be contacted by various clients offloading tasks. According to the system design proposed, the Dispatcher performs a simple but delicate task: for each input tensor belonging to a specific task received from a generic client, the dispatcher adds information about the optimal path that task must follow and then forward it to the first block required. This *taskID-path* translation mechanism is done by means of a **translation table** that reflects the decision of the resource optimization engine. In particular, for each system update, after deploying the missing blocks, the *manager* orders the controller to send an updated translation table to the dispatcher. The software structure of the Dispatcher does not differ much from that of a generic block, but it does not perform any inference.

3.6 System Workflow

This section provides an overview of the general functioning of the system, with particular emphasis on the most important phases.

As reported in figure 3.7, to deploy and/or terminate DNN blocks an optimization engine submits a *system update request* towards the BlockFlow REST API **(step 1)**. A system update request takes the form of an http POST request that contains information regarding the set of blocks needed and the set of admitted tasks, each of them decorated with its optimal forward pass. The body of the incoming http request is parsed and the *manager* is notified **(step 2)**. It reads a JSON file where the latest changes made are stored and calculates the missing blocks to be deployed. At the same time, if a block that was active until then is no

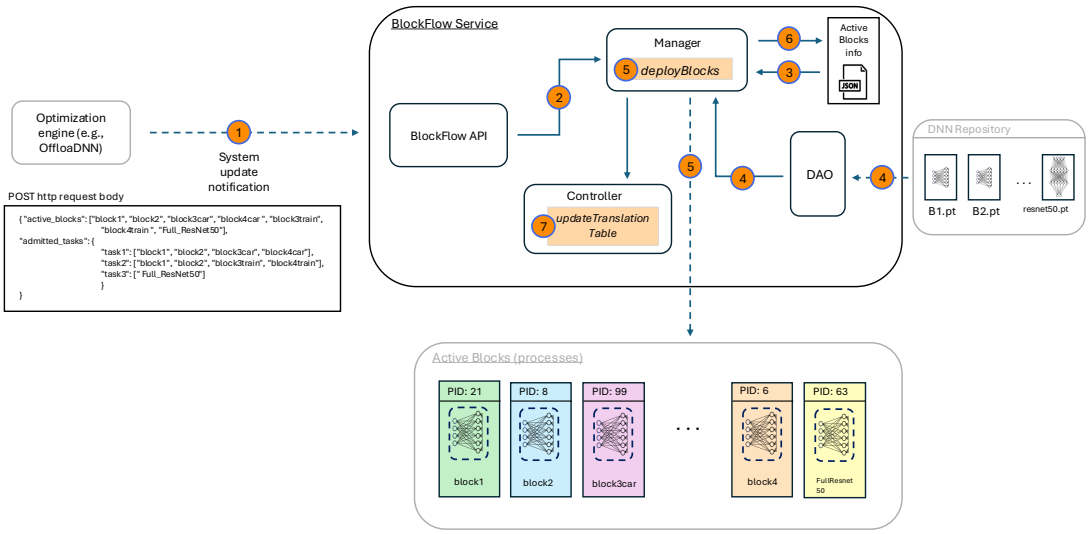

Figure 3.7: System Update workflow.

longer present in the new system update, it marks it as a block to be terminated (**step 3**). After obtaining the actual set of blocks to be deployed, it gets the reference path of the blocks (**step 4**) and iteratively spawns the new processes (**step 5**). Consequently, it updates the *active_blocks_info* json file (**step 6**) and notifies the *dispatcher* about new tasks admitted with their respective optimal paths through a control message (**step 7**).
It is important to notice that on the block deployment procedure, the *manager* passes as parameters the block operating mode (e.g, TensorMQ or ZeroMQ), the device where it has to move its own model (if more than one GPU is available in the node), the communication protocol to be used, its port and contact information of the controller (i.e controller port & controller address).

3.6.1 Block workflow

The Block's life-cycle depends on three main phases:

- *startup*,

- *running*
- *termination.*

The startup phase is devoted to the initial setup of the block, with operations like model loading, ZeroMQ context initialization and channel binding. Once the startup phase is ended and the method run is invoked, the block enters the full-operating condition.
As discussed in sec 3.5.1, each block launches three additional threads in addition to the main thread, devoted receive the incoming messages and put them in the right data queue, process control message and perform KPI monitoring. The main thread is responsible for inference and for forwarding the output tensor to the next block.

Receiver Thread

The receiver thread receives messages in an intelligent way with low impact on the CPU utilization (Algorithm 1).

Algorithm 1 Receiver Thread Routine

```
1: function RECEIVER THREAD
2:     Print: "receiver thread started..."
3:     Initialize zmq.Poller object
4:     Register control channel and data socket to poller (zmq.Poller.register)
5:     while receiver thread is active do
6:         Poll events or Wait for incoming events with short timeout
7:         if One of the channel have data then:
8:             if data socket has new data then
9:                 Receive message (sender, next hops, payload)
10:                Record current time
11:                Enqueue received data into processing queue
12:            end if
13:            if control channel has new message then
14:                Receive control message
15:                Enqueue control message into control queue
16:            end if
17:        end if
18:    end while
19:    Print: "receiver thread terminated..."
20: end function
```

The proposed solution involves the use of a ZMQ.Poller object, which allows efficient message polling between multiple ZeroMQ sockets in the same cycle. It also allows to control the number of messages received per time unit by manually setting parameters such as the receive high water mark. The main advantage of this solution resides on the possibility to set a timeout interval in which, if there are no messages in any of the registered channels, it adopts a blocking behavior, saving CPU cycles. At the same time, a sudden arrival of a message in one of the two channels interrupts the timeout.
The dummy solution that would not have included the use of the poller would have been to instantiate two separate threads, one for receiving control messages and one for receiving tensors or IPC handles, but this would have not only complicated the code but also wasted system resources. Another solution could have been to use a single thread for reception and inference, but this would have led to reduced system modularity and the inability to monitor certain system statistics such as the number of clients in the queue waiting to be served. Finally, it would have been problematic to manage the incoming data since the two channels receive different data types.

Inference Thread (main thread)

For each inference, the block performs a set of operations described in Algorithm 2. The data object extracted from the queue at the i-th operation is a tuple containing the following information: **arrival time** of the tensor within the block, sender, **remaining block to be traversed**, and **data object**. What is extracted from the queue is the same whether TensorMQ is active or not. If TensorMQ is active, the "data object" component is a serialized dictionary-type object structured as is reported in the Figure 3.8.

```
metadata = {
    "shape": tuple(pred.shape),
    "dtype": str(pred.dtype).split('.')[-1],
    "stride": tuple(pred.stride()),
    "handle_bytes" : ipc_handle
    }
```

Figure 3.8: Data Object in TensorMQ.

Conversely, if TensorMQ is not active, the data object is a serialized $torch.Tensor$ obj. The serialization library used is *pickle*.
The parameter *zeroCopy* (line 3 of alg 2) indicates whether the block works with

TensorMQ (if branch) or with normal ZeroMQ (else branch).

Algorithm 2 Block Inference Loop with TensorMQ Support

```
1: while inference_thread_on do
2:     (arrival_time, sender, next_hops, data) ← get element from tensor queue
3:     if zeroCopy is True then                                  ▷ TensorMQ branch
4:         tensor_metadata ← deserialize(data)
5:         next_hops ← deserialize(next_hops)
6:         ipc_handle ← tensor_metadata["handle_bytes"]
7:         tensor ← reconstruct_tensor_from_handle(ipc_handle)
8:         output ← model(tensor)
9:         next_hop_device ← GetNextHopDevice(next_hops)
10:        if block_device is different to next_hop_device then
11:            move output to next device
12:        end if
13:        new_handle ← getIpcHandle(output)
14:        new_metadata ← create_new_metadata(new_handle, output)
15:        serialized_metadata ← serialize(new_metadata)
16:        forwarding_socket ← GetForwardingSocket(next_hops)
17:        remaining_hops ← remove first element from next_hops
18:        forwarding_socket.send(serialized_metadata, remaining_hops)
19:    else
20:        tensor ← deserialize(data)
21:        next_hops ← deserialize(next_hops)
22:        move tensor to device (CUDA)
23:        output ← model(tensor)
24:        move output to host (CPU)
25:        output ← serialize(output)
26:        forwarding_socket ← GetForwardingSocket(next_hops)
27:        remaining_hops ← remove first element from next_hops
28:        forwarding_socket.send(output, remaining_hops)
29:    end if
30: end while
```

The pseudo code reported, clearly shows the main differences between the two approaches: in the if branch, there are no explicit memory copies between system memory (RAM) and video memory (VRAM) as in the else branch. The functions *reconstruct_tensor_from_handle*, *get_ipc_handle*, and *create_new_metadata* specific to the TensorMQ branch contain wrapper functions for low-level CUDA functions provided by PyTorch.

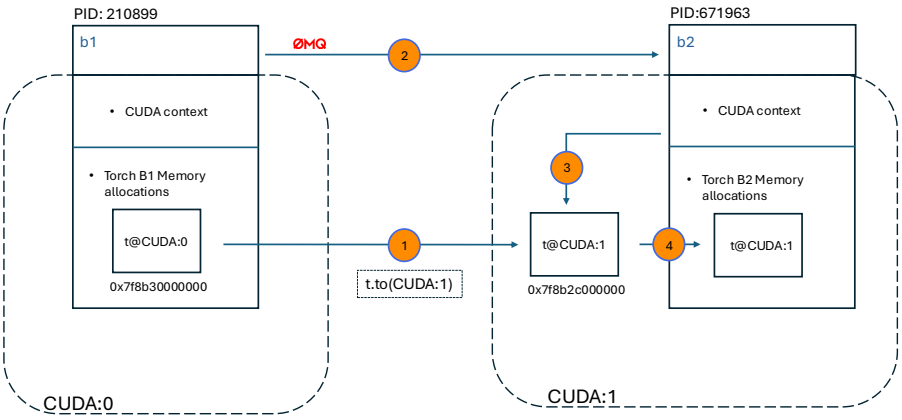

Figure 3.9: *TensorMQ* Tensor transfer procedure between 2 blocks over 2 different GPUs: **(step1)** block 1 invokes `.to(device)` Pytorch function, subsequently it generates an IPC handle of the in the new device and forward it to block2 with ZeroMQ**(step2)**. Once received, the handle is deserialized and block2 can access the tensor content outside its memory allocations **(step3)**. Finally, it reconstruct the tensor **(step 4)** and invoke `cuda.ipc_collect()` to free-up the occupied temporary memory resources and avoid memory fragmentation.

In **single-node-single-GPU** scenarios, the ability to populate a memory address directly on the GPU via IPC Handle means that the transmission of intermediate results between one block and another one running on the same CUDA device takes place without explicit transfers to system memory.
This mechanism of handle sharing and reconstruction to improve intermediate tensor deliver latency, is particularly useful also in **single-node-multi-GPU** scenarios. In this case, the transmitter block moves the tensor on the same device the next block is operating through the function *.to*(*device*) of *PyTorch*, generate the IPC handle and finally deliver it to the receiver block.
PyTorch automatically considers the most efficient channel to move tensors from a device to another one such as NVlink, PCIe bridge and SYS basing on the physical

connection available between devices.
It is essential to notice the role that *TensorMQ* plays in this case as well.

Figure 3.9 illustrates a hypothetical transfer of an intermediate tensor t between two blocks $b1$ and $b2$, operating on two different GPUs (e.g, CUDA:0, CUDA:1). If $b1$ moves tensor t from CUDA:0 to CUDA:1, this does not mean that $b2$ can automatically access it because each process has its own isolated CUDA context that maintains its own address space on the device. TensorMQ allows to notify the successful transmission of the tensor from one block to another one and at the same time, it allows the receiver block to reconstruct the tensor within its memory space through the IPC handle generated.

The forwarding of intermediate results between one block and another is performed using the *GetForwardingSocket* function, which takes as input the ordered list of blocks remaining to be traversed for the specific tensor. The list is structured as a list of tuples whose fields are: (*hop_address*, *hop_port*).

As reported in Algorithm 3, the *GetForwardingSocket* function extracts the

Algorithm 3 Get Forwarding Socket

```
1:  function GetForwardingSocket(next_hops_list)
2:      next_hop ← first element of next_hops_list
3:      for all connection in block_active_connection do
4:          if connection["name"] = next_hop["name"] then
5:              return connection
6:          end if
7:      end for
8:      new_socket ← zmq.socket(PUSH)
9:      if protocol = "TCP" then
10:         address ← format_tcp_string(next_hop["port"])
11:     else
12:         address ← format_ipc_string(next_hop["name"])
13:     end if
14:     new_socket.connect(address)
15:     update_connection_list(new_socket)
16:     return new_socket
17: end function
```

next_hop from the list (first element) and checks whether the block already has an open connection to that block. If so, it returns the object of type ZMQ.Socket to that block; otherwise, it instantiates a new connection to the block, updates the list of open connections, and returns the object of type ZMQ.Socket.

It is important to notice that the number of open connections to next_hops

must be aligned with the system updates delivered from the optimization engine; this is done by means of *update_connection* function invoked by the Control thread as soon as it receives the *update* command.

Algorithm 4 Get Next Hop device

```
1: function GetNextHopDevice(next_hops_list)
2:     next_hop ← first element of next_hops_list
3:     for all block in block_devices_translator do
4:         if block["name"] = next_hop["name"] then
5:             return block["device"]
6:         end if
7:     end for
8: end function
```

Control & Monitor Threads

Control and Monitor Threads are two threads launched at the block startup to perform control operations and KPIs monitoring, respectively. They have a similar implementation but the main difference lies on the fact that Control Thread implements a blocking mechanism over the control message queue to improve system resource consumption as it is shown in line 4 of alg. 5. Conversely, Monitor thread, once launched, collects hardware specific magnitudes related to the block with a predefined frequency (e.g, 10 Hz).

Algorithm 5 Control Thread Routine

```
1: function ControlThread
2:     Print: "control thread started..."
3:     while control thread is active do
4:         Retrieve message from control queue or wait the reception if empty
5:         Execute control action with the message
6:         Sent result back to the controller over the control channel
7:     end while
8:     Print: "control thread terminated..."
9: end function
```

More in detail, the Monitor thread registers over time the following magnitudes: process-specific allocated torch VRAM, process-specific cached torch VRAM, process total VRAM, GPU utilization, GPU clock, GPU temperature, process RAM, CPU clock, process CPU utilization etc.

3.6.2 Dispatcher workflow

The dispatcher has a similar block's workflow, but it does not perform any inference in the main thread. Its main responsibility is to apply the taskID-optimalPath translation mechanism and route the tensor to the first node requested in the pipeline. If the dispatcher is located on the same host as the first hop, it also generate the cuda IPC handle and forward it. The Dispatcher has a Control thread and a Receiver thread that work in the same way as the one launched in a generic block. Control thread messages play a crucial role since they carry the information related to block topology updated and the taskID-optimalPath translation dictionary.

The taskID-optimalPath translation mechanism is performed by the receiver thread whose logic is reported in Algorithm 6.

Algorithm 6 Receiver Thread Routine with Task Path Translation

```
1: function ReceiverThread
2:     Initialize poller
3:     Register control channel and data channel to poller
4:     while receiver thread is active do
5:         Wait for incoming events with short timeout
6:         if interface has received data then
7:             Receive message (sender, payload)
8:             Record current time
9:             Determine next hops using task path translation
10:            Enqueue data with metadata into processing queue
11:        end if
12:        if control channel has received message then
13:            Print: "[name]:  new control message received"
14:            Retrieve control message
15:            Enqueue control message into control queue
16:        end if
17:    end while
18:    return
19: end function
```

Chapter 4

Experimental Results

In this chapter, the main findings regarding the implemented tools are presented. BlockFlow has been extensively tested in different working conditions. This work also focuses on the inference performance of the modular DNN architecture. We collected time-related metrics and hardware-specific KPIs from two host machines with different computing capabilities to conduct a thorough analysis. Subsequently, those data were analyzed and insightful graphical illustrations were generated.
The main goal was to assess the effectiveness of TensorMQ in a scenario where modular-DNN architecture are deployed in physical machines with one single hardware accelerator device and more than one hardware accelerator device.

4.1 Experimental setup

Every experiment carried out in this thesis work was conducted in two different computing platforms with different computing capabilities. Table 4.1 reports the technical specifications of each of the two servers. *FullSuper* server which is a single-GPU machine (NVIDIA Quadro GV100 [1]) while *Awenode* is equipped with 8 NVIDIA L40S GPUs [2]. Since it has two distinct NUMA nodes, different GPUs have different NUMA node affinity. The GPU topology is reported in Appendix A.1.

The DNN architecture selected for the test was ResNet50, whose architectural schema and details were provided in sec. 2.1.2. However, it is important to notice that these tests can be performed with any neural network architecture whose

[1] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-volta-gv100-data-sheet-us-nvidia-704619-r2-web.pdf

[2] https://resources.nvidia.com/en-us-l40s/l40s-datasheet-28413?ncid=no-ncid

forward pass can be divided into one or more parts.

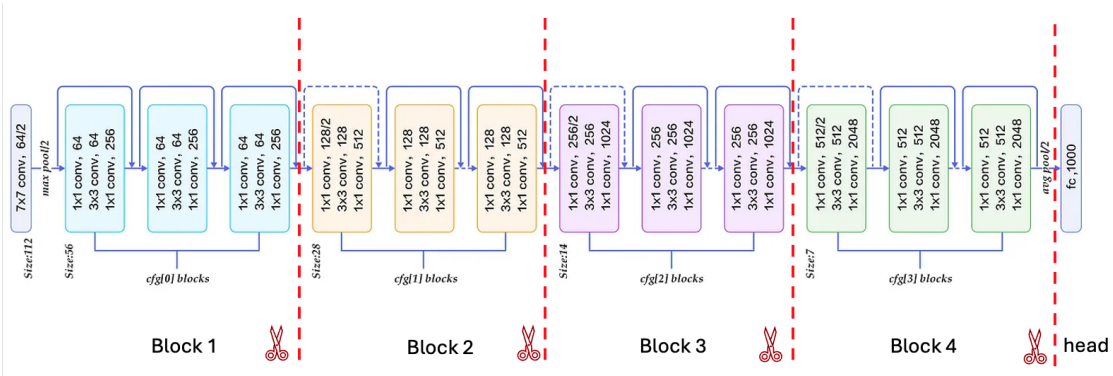

Figure 4.1: ResNet50 Splitting points.

ResNet50 has been split into 4 main blocks in strategic points of its architecture as represented in Figure 4.1. Afterwards, each block has been exported in TorchScript by scripting and tracing and in both cases, each block copy (scripted version and traced version) has been stored into the filesystem.

Component	**single-GPU-server**	**multi-GPU-server**
Host	FullSuper	Awenode
Operating System	Ubuntu 22.04.5 LTS (x86_64)	Ubuntu 24.04.2 LTS (x86_64)
Kernel	Linux 6.8.0-57-generic	Linux 6.8.0-62-generic
CPU	AMD EPYC 7601 @ 2.20 GHz	2× AMD EPYC 9374F @ up to 4.3 GHz
RAM	256 GB	1.5 TB
GPU	1x NVIDIA Quadro GV100, 32 GB VRAM	8× NVIDIA L40S, 46 GB VRAM
CUDA Version	12.7	12.6
NVIDIA Driver	565.57.01	560.35.05
NUMA Nodes	1	2

Table 4.1: Comparison between single-GPU-server and multi-GPU-server.

4.1.1 Split-ResNet50 Offline analysis

In this analysis, each block (one at time) has been instantiated on the target device and 1000 inferences were performed collecting their relative computing time. Simultaneously, a secondary tool implemented in python was collecting targeted GPU related metrics over time such as the amount of VRAM used by PyTorch as well as the cached torch VRAM memory and the total amount of VRAM used by the process. For each of the process-specific timeline kpi collected, average and standard deviation were computed whose values are reported in 4.2. The table data core focus was to assess, if exists, differences in terms of inference performance and resource consumption between traced and scripted models.

Type	Model	Average Inference latency (ms)	Std Inference latency (ms)	Average Torch allocated VRAM (MB)	Std Torch allocated VRAM (MB)	Average Torch reserved VRAM (MB)	Average Process occupied VRAM (MB)
SCRIPTED	block1	2,401	0,142	4,858	1,230	26	394
	block2	2,718	0,154	9,689	1,046	26	394
	block3	4,021	0,253	29,755	0,967	56	424
	block4	2,276	0,161	59,612	2,619	110	478
	class_h	0,287	0,055	16,328	0,000	22	396
TRACED	block1	2,254	0,176	4,896	1,362	26	394
	block2	2,641	0,145	9,536	0,763	26	394
	block3	3,914	0,575	29,746	1,040	56	424
	block4	2,233	0,170	59,442	0,989	110	478
	class_h	0,287	0,067	16,339	0,104	22	396

Table 4.2: Comparison of memory and latency metrics for `scripted` and `traced` blocks.

The data showed that there were no substantial differences between the two approaches in terms of performance or resource consumption. For this reason, all tests were performed with `scripted` blocks.
The summation of the average inference time for each block lies as a baseline for future experiments, since it does not take into account the inter-block transmission time hence allowing to understand the impact of the communication overhead. Figure 4.2 shows the parameter distribution across ResNet50 blocks. Notably, most of the model's parameters are concentrated in the final layers (the tail) of the network.

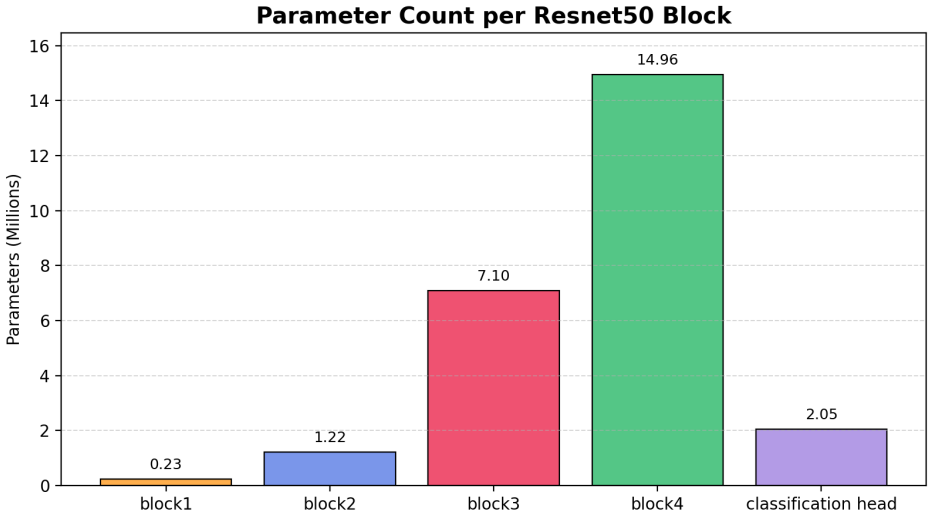

Figure 4.2: ResNet50 Parameters (millions) per Block.

4.1.2 ZeroMQ forwarding performance analysis

In this test, pure forwarding performance of ZeroMQ were evaluated for the 2 different servers and for 3 different data-types: `string`, `json` and `torch.Tensor`. The test has been realized by letting 2 distinct processes synchronously communicate with the REQ/REP communcation paradigm provided by ZeroMQ. The performance were analyzed by collecting the Round-Trip-Time (RTT) in ms for each message sent.
This test's goal was twofold: firstly, it allowed to assess the suitability to adopt ZeroMQ as high-performance and light-weight communication library and secondly, it provided an idea on how much *Awenode* server outperforms *FullSuper*.

Data Type	HTTP RTT (ms)	ZeroMQ RTT (ms)	Ratio (HTTP / ZeroMQ)
String	1.676	0.069	∼24x
JSON	2.273	0.106	∼21x
Tensor	10.885	1.054	∼10x

Table 4.3: RTT Comparison on `awenode` (ms).

For each test, 10000 messages were sent and the average RTT was computed. The test results are reported in tables 4.3. Please notice that the average computation takes into account the system warm-up time by dicarding the first 250 subsequent inferences. Although the performance disparity between the two servers is significant, the average RTT measured for *FullSuper* remains substantially below 1 millisecond, confirming the performance expectations that ZeroMQ had assured.

For completeness, the first column of the table 4.3 reports the KPI of the HTTP test. This test was carried out with the same logic but exposing an uri of a REST API implemented in CherryPi. Another process sends HTTP POST requests to this URI and measure the RTT of each. Clearly, HTTP performance is drastically worse than ZeroMQ performance with a reduction in average RTT up to 24x.

4.2 Pipeline performance analysis

This section presents an extensive performance analysis of the system, with a particular focus on performance times under different operating conditions

BlockFlow uses multi-processing to deploy DNN blocks based on system updates coming from a system resource optimization engine. Supposing a simple and unique offloaded task (e.g, *task1*), the system update is received as payload of an http POST request to the API and it can be structured as reported in listing 4.1. Once received, BlockFlow is in charge to 1-to-1 recreate the modular architecture requested by deploying the blocks and instantiate the communication channels between them.

```
{
    "blocks": ["block1", "block2", "block3", "block4_class"],
    "tasks": {
            "task1": ["block1", "block2", "block3", "block4_class"
   ]
    }
}
```

Listing 4.1: Generic System Update POST request.

To test the deployed pipeline, a `TaskGenerator` object was implemented and launched. Its functioning simulates a client sending tasks toward the system. It opens a communication channel towards the Dispatcher and it starts generating tasks. In case of a simulated environment with N clients with N different tasks, N different `TaskGenerator` were launched.
`TaskGenerator` object can operate with ZeroMQ or with TensorMQ and it can handle synchronous or asynchronous communcation. Notice that if the asynchronous mode is chosen, a parameter λ is needed. In this particular case, tasks are generated according to an exponential distribution (poisson process) with parameter λ in order to provide a more realistic scenario and introduce variability into the system. Conversely, if the synchronous mode is chosen, the client receives the inference result of the previous message before sending the subsequent one. It is straightforward to understand that, in this case the client adopts a "gentle" behavior versus the system respecting its computing time and avoiding to overwhelm its resources.

4.2.1 Mathematical model

Considering a task t and with its optimal forward pass $\pi_t = [block_1, block_2, ..., block_n]$, the *pipeline computing time* T_t^i for the $i-th$ inference of a task t is given by the sum of the contributions of each block in the forward pass and it can be written as:

$$T_t^i = \sum_b T_b^i = \sum_b o_b^i + m_b^i + w_b^i \tag{4.1}$$

where:

- $b \in \pi_t$ and it stands for the set of blocks to be traversed;
- T_b^i is the *time spent in block b* for inference i belonging to task t;
- o_b^i indicates the overhead time spent in the block b for inference i;
- m_b^i indicates the inference time of the block b for inference i (e.g, time related to the model inference);
- w_b^i indicates the waiting time before being served in block b for inference i.

The overhead component o_b^i is had by the contribution of several operations such as: *deserialization*, *tensor reconstruction/tensor move to GPU**,*IPC handle creation/move to CPU*, *serialization*, *forwarding*.

In this context, the term End-to-End delay, borrowed from networking as *the transmission delay for a packet from the source node to the destination node*, is synonymous with the pipeline computing time for a single inference.

Surrogate latency measurements of 4.1 for real case scenarios involve the transmission time T_{tx} from the client to the inference pipeline and vice-versa T_{rx} also called Round-Trip-Time (RTT). Due to the setup of the experiments carried out, since also the clients are running on the same physical machine the propagation delays of the message is negligible. Experimental results confirm that T_{tx} and T_{rx} account on average for around 1 ms. For that reason we assumed that RTT for an inference i of a task t can be the equal to $T_t^i + 2ms$.

As introduced in the eq.1.1, T_b^i depends on the device d hosting the block and the block load ρ_b.
The block load ρ_b depends on the arrival rate λ_b and on the block service rate μ_b:

$$\rho_b = \frac{\lambda_b}{\mu_b} \tag{4.2}$$

Notice that μ_b is the reciprocal of m_b.

Finally, the actual arrival rate λ_b experienced from a generic block b is function of the number N of different task t sharing the same block b. Since the assumption

of exponentially distributed arrival rates towards the system, given N different tasks with N different arrival rates λ, the actual arrival rate λ_b in a generic block b is had by the "competition" of N different Poisson processes and it can be written as:

$$\lambda_b = \sum_{i=1}^{N} \lambda_i \tag{4.3}$$

From queuing theory, a well conditioned system should not overpass the ergodicity condition $\rho < 1$. In this case each block should have $\rho_b < 1$ but since a pipeline is considered, the condition that hold is:

$$\lambda_b < \mu_{bottleneck}, \forall b \in \pi_t$$

where $\mu_{bottleneck} = min(\mu_b), \forall b$ with $b \in \pi_t$

The ergodicity condition is strictly maintained with clients operating in synchronous mode.

4.2.2 Single-host-single-GPU

Pipeline average computing time

Figure 4.3 reports the average pipeline computing time using HTTP, ZeroMQ and TensorMQ with a single client in synchronous mode. The last bar serves as baseline and it reports the total computing time as a summation of each computing time measured per block in the offline test described in sec 4.1.1. The offline term is 11.703 ms.
The average is computed over a 120 seconds long test and it takes into account the system warmup discarding the first 250 inference times collected.

The asterisk in HTTP indicates a slightly different test procedure. That histogram represents the average round-trip time of an entire ResNet50 exposed via a REST API. The result of the test with HTTP performed in the “split” version shows unacceptable latencies, and since the difference in performance between the HTTP-based solution and the others is enormous, it has not been included for visualization reasons.
The solution implemented in HTTP shows an average computing time of around 200ms, while ZeroMQ values are almost a fourth of it (44.9 ms). However, TensorMQ reduced the pipeline computation time of ~2.8x against ZeroMQ and ~11x with respect to HTTP in single block (up to ~40x in split architecture).

Overhead comparison ZeroMQ and TensorMQ

The efficiency of TensorMQ stems from its significantly reduced overhead in block-level communication. Figure 4.4 presents the average inference time and the

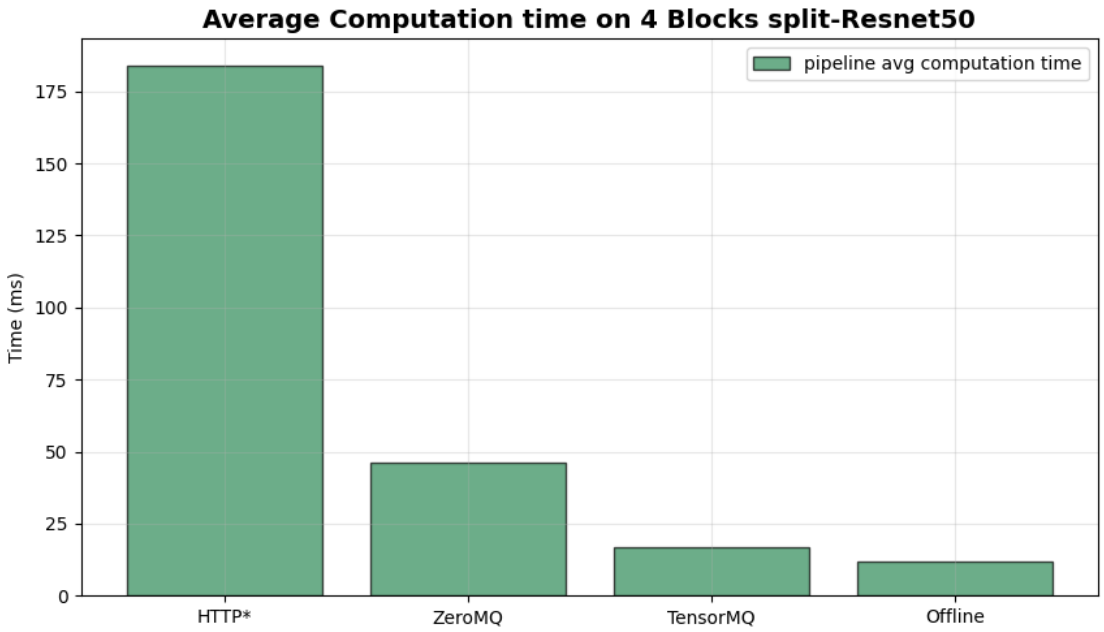

Figure 4.3: 4-block pipeline average computing time.

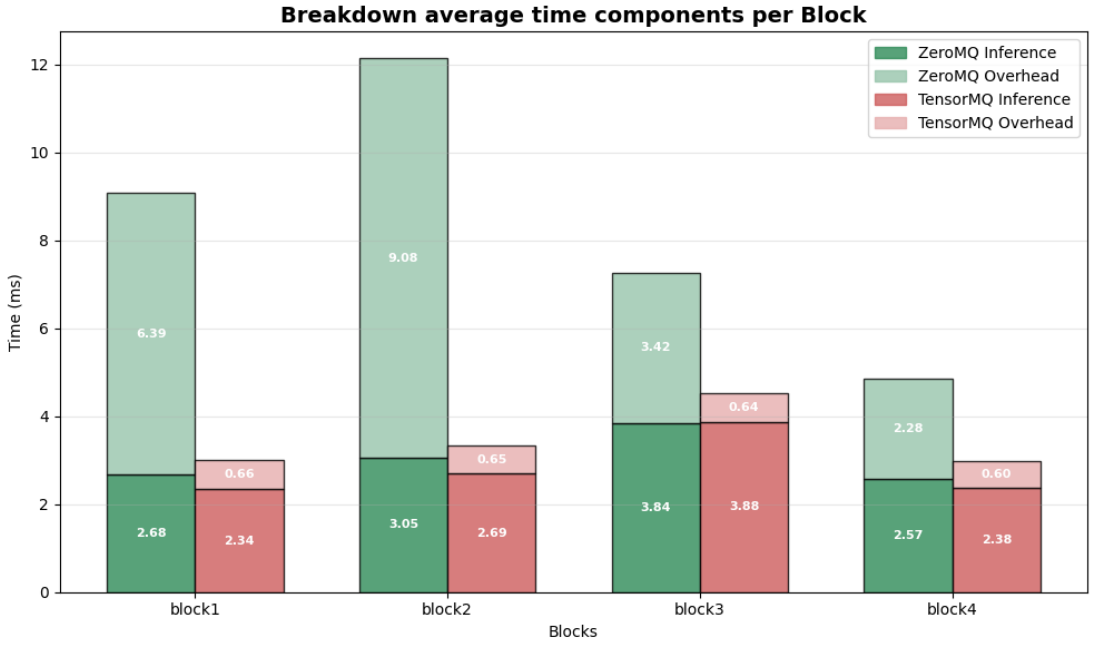

Figure 4.4: Time components per block.

corresponding overhead per block, categorized by the communication technique employed. Since the inference times are nearly identical for both ZeroMQ and TensorMQ, the primary advantage of TensorMQ lies in its ability to minimize communication overhead, thereby improving overall system efficiency. Moreover, TensorMQ overheads present much less variability between blocks and between subesequent inferences. For a better visualization, the sum of each block contribution has been done and the ePDF of the inference times and overheads for both ZeroMQ and TensorMQ is reported in fig. 4.5. Almost all the magnitudes collected appear to be normally distributed but focusing on the overheads, while the pipeline overheads of ZeroMQ present a large variance (third figure starting from left), the one related to TensorMQ are perfectly wrapped around the mean value. The average pipeline overhead of TensorMQ is 2.55ms while the one from ZeroMQ is 21.18 ms.
The ePDF of inference and overhead each block in the pipeline can be found in Appendix B.

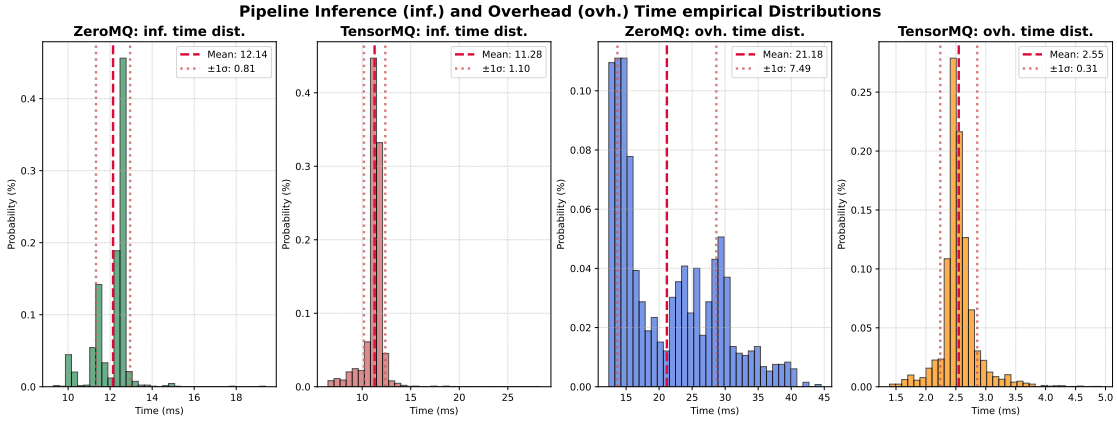

Figure 4.5: Pipeline inference and overhead time distributions.

GPU & CPU utilization

Considering a generic client operating synchronously and transmitting at the maximum allowed rate (transmit the new tensor as soon as it receive the result of the previous one), CPU and GPU utilizations data were collected over time. GPU utilization (%) can be directly monitored through PyNVML, while it was necessary to consider only the CPU used by the processes involved. By the use of Psutil, process-specific CPU utilization timelines were collected, summed element-wise among all the processes active (blocks in the pipeline) and finally divided element-wise by the number of the blocks. Experimental results are reported in Figure 4.6, whose left subfigure presents the comparison between the GPU utilization between

ZeroMQ and TensorMQ while the right one presents the same but for CPU.

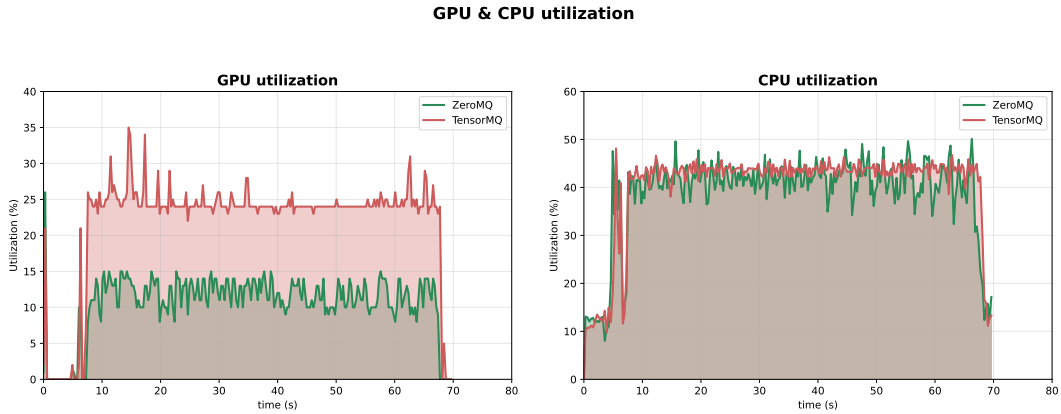

Figure 4.6: GPU and CPU utilization.

As Figure 4.6 shows, TensorMQ presents an almost doubled utilization (%) with respect to ZeroMQ over time; this is originated by the fact that the number of inferences performed in the time unit is greater, leading to a better exploitation of resources due to less idle times of the GPU. Moreover, it is important to highlight that the increment in average GPU utilization does not reflects the increased amount of inferences performed in the time unit for both the solutions; indeed, considering a single synchronous client the average pipeline computing time in TensorMQ is 2.8x less with respect to ZeroMQ while the average GPU utilization increases of only ~2.0x.

Pipeline scalability

The results presented in the previous subsection involved the use of tensors with $batch = 1$. In order to understand how the pipeline would behaves with batch sizes greater than 1, several tests were carried out varying this parameter and analyzing how the latency performance were impacted.
The batch size values tested were: [2,4,8,16,32].

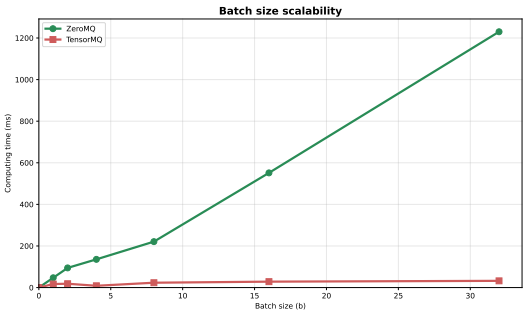

Figure 4.7: Pipeline Computing time under different batch size

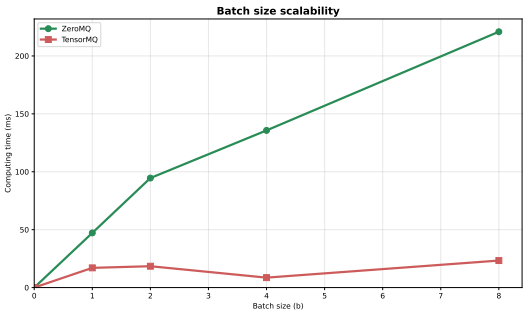

Figure 4.8: Pipeline Computing time under different batch size (zoomed)

Figures 4.7 and 4.8 display the variation of the pipeline computing time versus the tensor batch size. The rate of scaling of TensorMQ significantly outperforms the one of ZeroMQ for a simple reason: by varying the batch size, the amount of MegaBytes transmitted between a block and the subsequent one varies in a proportional way. Conversely, in TensorMQ even if the tensor size increases, the IPC handle remains of fixed size (around 180 Bytes). Hence, the increase of computing time experienced is due only to the increased inference time (which scales very well in GPUs) while the overheads remain constant.

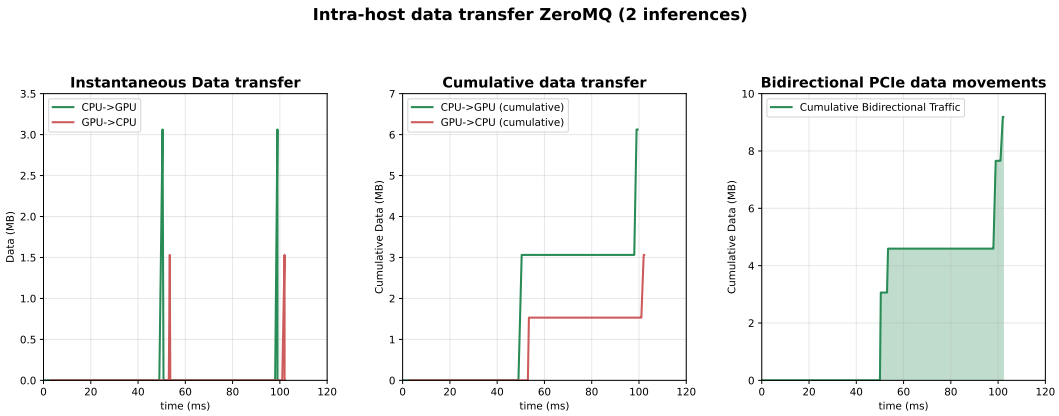

Figure 4.9: Data Movement analysis with NVIDIA nvprof for 2 consecutive inferences with ZeroMQ.

In this context, figure 4.9 provides a good contribution about what physically happens under the block's hood during 2 consecutive inferences with ZeroMQ. More in detail, during a test, one process among the available on the pipeline (e.g, block1) was profiled with `nvprof` and cuda kernels were collected and stored in

a .csv file. Among all, rows with `[CUDA memcpy HtoD]` and `[CUDA memcpy DtoH]` kernels were extracted as well as their details. Figure on the left reports the instantaneous data transfer (MB) from the host towards the device (Red line) and vice-versa (Green line). The period of the signals is consistent with the results since the distance between 2 consecutive peaks is around 50 ms.
Figure 4.9 (right) reports the cumulative amount of data exchanged over the PCIe interface showing around 9MB for just 2 single inferences. By doing the same test with TensorMQ active, no `[CUDA memcpy HtoD]` and `[CUDA memcpy DtoH]` were reported in the log file, which indicates that no explicit memory copies between GPU and CPU are performed at inference time.

Less data movement and more inferences per time unit lead to a more efficient solution in terms of energy consumption. This hypothesis is confirmed by Figure 4.10 which reports on average the estimated pipeline energy consumption per inference with a focus on increasing batch size scenarios.

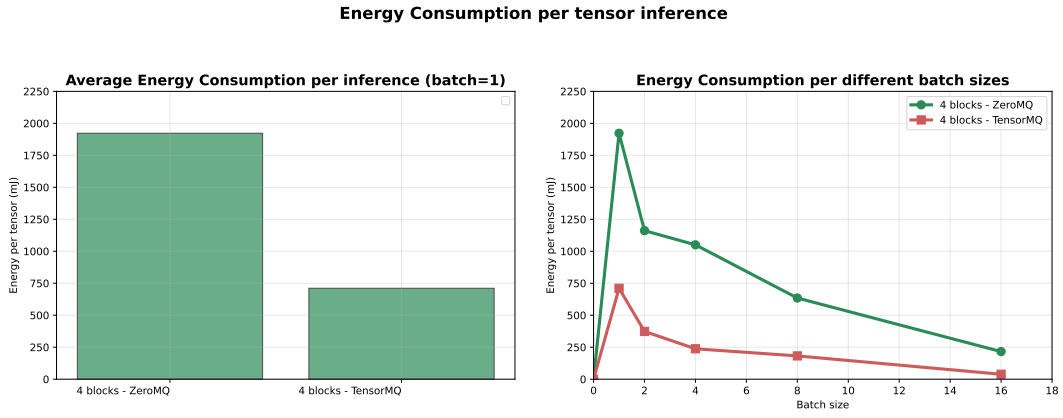

Figure 4.10: Energy consumption per inference.

On a 4-block pipeline, the solution that involves the use of TensorMQ presented an average energy consumption of ~710 mJ per tensor when the batch size is equal to 1. Conversely, for the same setting, ZeroMQ solution had an average energy consumption of ~1920 mJ per tensor (2.71x more). These values decreases exponentially if the batch size increases, as expected.

In sec 4.2.1, we introduced the concept of *actual block arrival rate* λ_b in case the block b is shared among N different tasks. To understand how the pipeline computing time varies when more than one client requests for the same forward pass, a multi-client scalability test was performed. In particular, for each test run the average Round-Trip-Time experienced by each client is reported by increasing the number of clients from 1 to 10 operating in synchronous mode. In other words, if 5 clients were active simultaneously, each block of the pipeline were shared among

5 different tasks. The results are shown in figure 4.11.

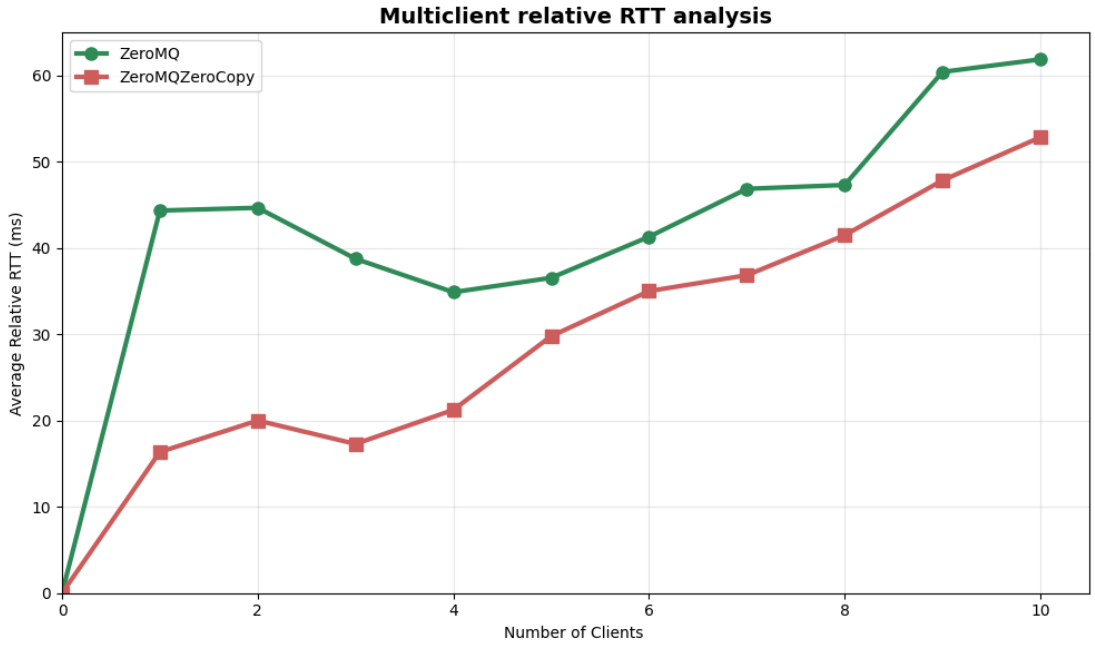

Figure 4.11: Pipeline scalability in multi-client scenario.

Both the lines show a similar pattern but with 2 main differences: clearly, the TensorMQ one always stays under the ZeroMQ one since the computing time is always less for the reasons discussed in the above sections. Secondly, there is a plateau with client number ranging from 1 to 4 where the average RTT time experienced by each client remains constant (approximately 18ms) in TensorMQ or even decreases in the case of ZeroMQ (from 45ms to 35ms). Focusing on the latter, this phenomenon is due to the fact that each block waits for a certain amount of time in an idle state between one inference and the next; this results in inefficient use of GPU resources by each block. Conversely, as soon as the number of client starts increasing the experienced arrival rate increases as well leading to a better GPU resource exploitation. From 5 clients on, the RTT increases linearly as expected.

Ergodicity condition analysis

To determine the ergodicity condition, several system runs were performed. They involved the use of a client sending tensors with $batch\_size = 1$ asynchronously and with increasing sending rate λ. The system throughput (TH) was measured as the amount of results received from a *Collector* entity per time unit. The

experimental results are presented in fig.4.12.

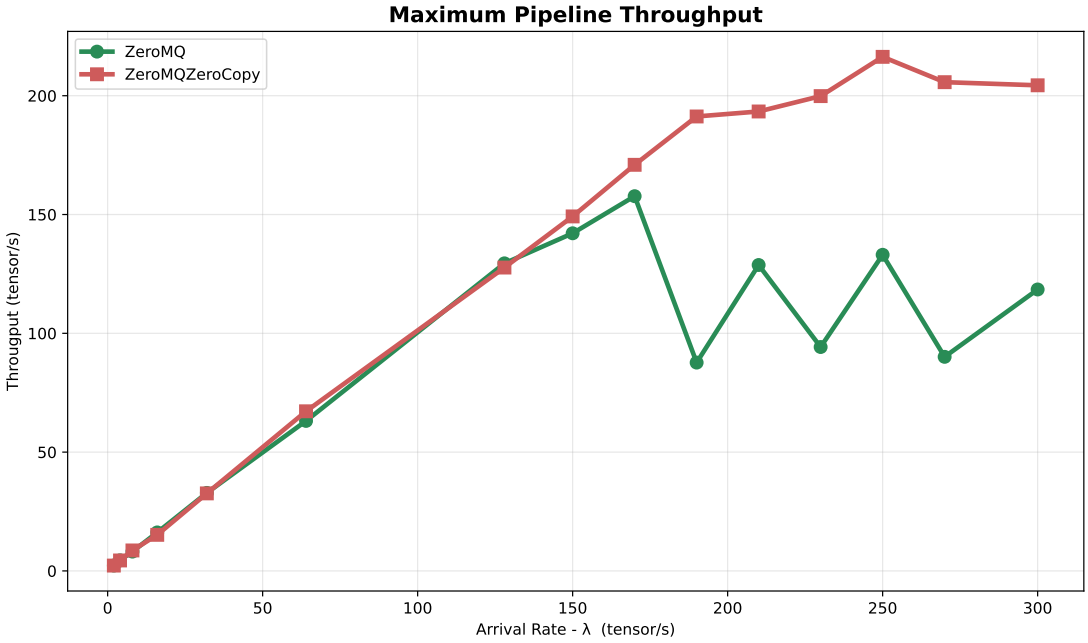

Figure 4.12: 4-blocks Pipeline Throughput analysis.

As expected, the pipeline saturation point is higher with the use of TensorMQ. Both the communication paradigms present a linear behavior in the first part of the plot according to the Little's Law (i.e what enters in the system goes out) but ZeroMQ solution reaches its maximum TH around 160 tensors/s and then it continue with noisy results ranging between 125 and 97 tensors/s. Conversely, TensorMQ curve reaches the maximum at 217 tensors/s (on *FullSuper*) recording an increased TH of ~42% over the same machine with the same computing capabilities. It is important to notice that data of TH in absolute values are depending on the hosting machine used for test.
For arrival rates larger than the one detected experimentally, the system would end up accumulating tensors within their queues leading to an increased usage of RAM and VRAM.

In this context, if the ergodicity condition is passed, TensorMQ exhibits more aggressive VRAM usage as the Figure 4.13 shows. This particular behavior is due to the intrinsic functioning of TensorMQ and handle sharing and it is simple to explain. Whenever a block ends doing the inference, it puts the output tensor on the shared memory and generate an IPC handle to be sent to the next block. For

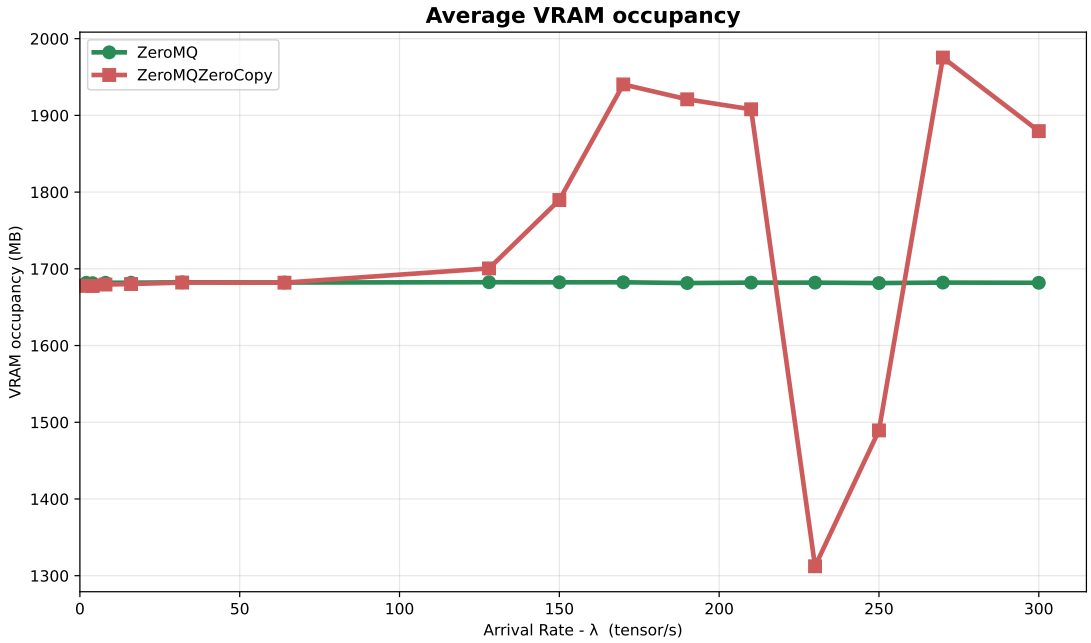

Figure 4.13: Average VRAM occupancy under different arrival rates.

the time span in which this output tensor is waiting to be processed, it remains in the VRAM causing an increased average VRAM occupancy. Clearly, this does not happen when ZeroMQ with full tensor forwarding is used. The latter, is more gentle with the VRAM but on the contrary, it is not with the RAM since for the amount of time the serialized tensor is sent and it is waiting to be served by one of the blocks, it is buffered in RAM.
The minimum average VRAM occupied coincides with the saturation point of around 220 tesors/second. It is quite difficult to explain the reason behind that but this can be originated by the fact that around this load values, the GPU computing resources are perfectly exploited and with almost zero idle times but also small waiting queue sizes.

VRAM occupancy components

In order to better understand how VRAM is used during the working period, we carefully analyzed how each block occupies it. Taking into consideration Figure 4.2 and Table 4.2, results presented in Figure 4.14 faithfully reflect what has already been discussed previously during the offline test.

The amount of average VRAM used by PyTorch is proportional to the number

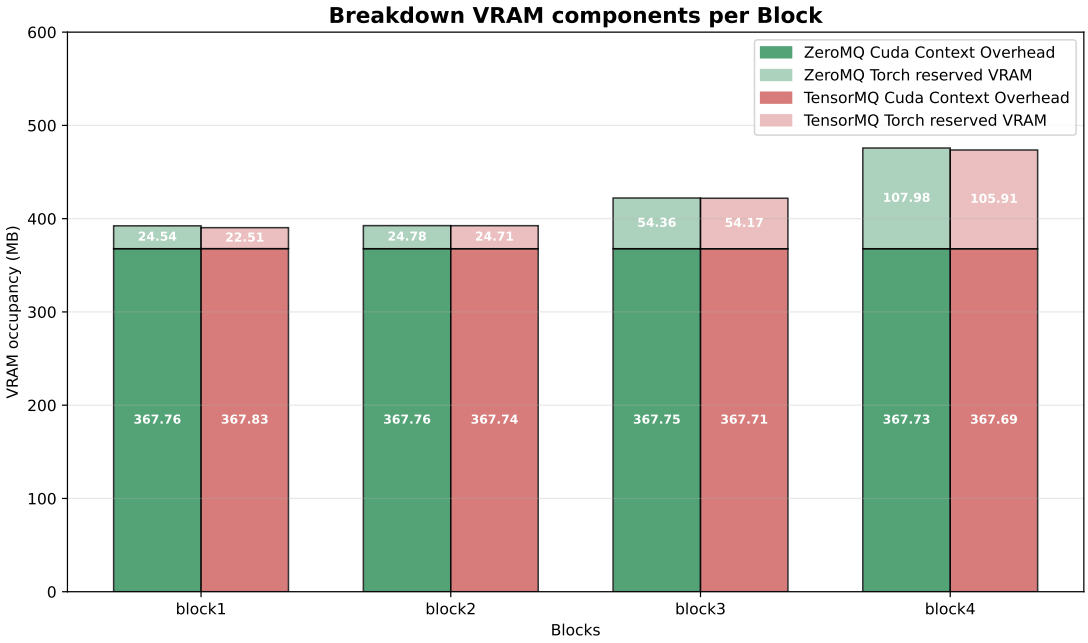

Figure 4.14: Average VRAM occupancy per block.

of parameter of each block and no significant differences were recorded between ZeroMQ and TensorMQ. It is important to recall that the allocated PyTorch VRAM is aligned with the registered in the offline test while the cached is slightly more (from 30 to 40 %).
However, the main purpose of this test was to assess how much fixed VRAM overhead is needed to pay for the deployment of each process. In this case, this amount is around 370MB per process.

4.2.3 Single-host-Multi-GPU

In order to assess how the system would perform in case it is not possible to host all the required blocks in a single devices, a multi-GPU server was used for that purpose. Another reason which led us to test the BlockFlow in such scenario was to understand how TensorMQ would have helped into the management of tensor movement between devices.
As discussed previously, each running process operating on GPU, occupies a portion of VRAM that cannot be accessed by any other process. Coherently, if a process moved a generic tensor from a GPU device d_i to another one d_{i+1}, the latter would not be able to directly access or modify it.
TensorMQ helped on the successful tensor transfer notification from one device to

another one between blocks, allowing to access the portion of memory the tensor resides and therefore its correct collection for subsequent handling.

Before delving into the test configuration details, we introduce the concept of Path Distribution Index (PDI) computed as:

$$PDI(\%) = \left(\frac{N_{GPU}}{N_B}\right) * 100 \tag{4.4}$$

where N_{GPU} is the number of different GPUs over which the blocks are hosted and N_B is the number of blocks within the optimal path.
The test carried involved the use of a multi-GPU server with blocks deployed on different GPUs according to:

- $PDI = 25$: given 1 available device, 100% of the blocks on that device (*4 Blocks-1 GPU*),
- $PDI = 50$: given 2 available devices, 100% of the blocks equally divided on that devices (*4 Blocks-2 GPUs*),
- $PDI = 100$: given 4 available devices, 100% of the blocks equally divided on that devices (*4 Blocks-4 GPUs*).

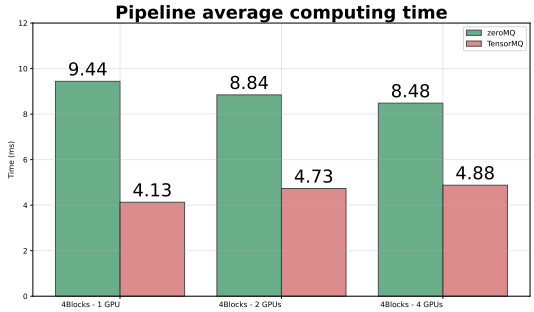

Figure 4.15: Blocks average **computing time** with different deployment scenarios.

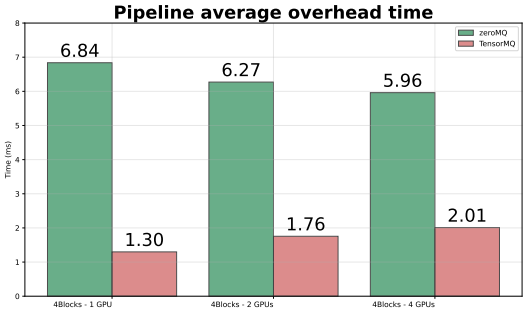

Figure 4.16: Blocks average **overhead time** with different deployment scenarios.

In addition, a single client with exponential sending rate $\lambda_c = 300$ tensors/second and tensor with batch size = 1 were used.
Figure 4.15 and Figure 4.16 report the average timings for computing and overheads respectively per each block, collected in all the system configurations presented above.

By looking at Figure 4.16, it is straightforward to notice that TensorMQ outperforms ZeroMQ in all of the tested deployment scenarios. However, the 2

communication paradigms show opposite trends. TensorMQ reports an average overhead time reduction of ~**5.26x**, ~**3.56x** and ~**2.96x** for the configuration 4B-1GPU, 4B-2GPUs, 4B-4GPUs, respectively.

With TensorMQ, the average total communication overhead increases according to the PDI. This result is consistent on what we expected from theory since when all the blocks are hosted on the same GPU, TensorMQ allows for essentially a ZeroCopy solution to share the intermediate tensors for one block to another one. Conversely, if a least one block of the optimal forward path is hosted in a different device, there is the need to transfer it physically. This latency depends on the physical communication technology used (NVLink, Infiniband, direct PCIe or CPU bounded transfer) and/or on the computing capability of the hosting machine (in case of CPU bounded transfer).
On the other hand, with the use of ZeroMQ with full tensor forwarding, the overhead decrease with an inverse proportionality with respect to the PDI. This behavior could be originated by an high number of context switches when all the blocks operates on the same device.

4.3 Modular DNN deployment vs Standard Deployment

In order to be aligned with the requirements of modularity, isolation and flexibility on the deployment of different portion of the DNN over different GPUs or even across different hosts, a multi-processing solution was chosen with huge attention on performance based metrics such as latency and system throughput.
However, the initialization of a CUDA context for each process involves a fixed overhead of approximately 370 MB (as it is shown in Figure 4.14).
Based on the experiments conducted and the data collected, we provide a comparison between the modular DNN deployment based on multiprocessing and standard deployment in different hypothetical working scenarios.
For better clarity of explanation, we provide a simple mathematical formulation of the problem.

4.3.1 Mathematical formulation

Let $T = \{\tau_1, \tau_2, \ldots, \tau_N\}$ be the set of N tasks admitted by the system. Let $B = \{b_1, b_2, b_3, b_4, b_{\text{resnet}}\}$ be the set of DNN blocks obtained from the decomposition of a base model such as ResNet50, plus a full-specialized Resnet50 (b_{resnet}) for the most accuracy and delay-sensible offloaded tasks.

In traditional deployment, a one-to-one mapping holds between each admitted task and its corresponding deployed model. Hence, N tasks require N independent

DNN models to be deployed.
Conversely, modular deployment encompasses the possibility to decompose a full DNN structure into blocks and thus, share one or more blocks among multiple admitted tasks deploying only a subset of task-specific specialized block instead of the full DNN structure.
Let $H \in \{-1, 0, 1\}^{N \times |B|}$ be the specialization matrix, where entry h_{ij} indicates whether block b_j is required (and specialized) for task τ_i.
Specifically:

$$h_{ij} = \begin{cases} 1 & \text{if } b_j \text{ is a specialized Block for task } \tau_i \\ -1 & \text{if } b_j = b_{resnet} \text{ (i.e, full DNN specialization) for task } \tau_i \\ 0 & \text{otherwise} \end{cases}$$

Let $O = [o_1, o_2, o_3, o_4, o_{\text{resnet}}]$ denote the VRAM occupancy of each block in B, where each element o_j is given by:

$$o_j = M_{i\_params} + M_{i\_torch} \tag{4.5}$$

where: M_{i_params} (MB) is the contribution given by the number of parameters for the i-th block and M_{i_torch} is the PyTorch VRAM allocation (excluding model size) for block b_j. Moreover, each deployed process presents an additional VRAM occupancy contribution given by the CUDA context initialization ($C_{context}$) that was experimentally proved to be constant and independent on the block size. It accounts for around 370 MB in the tested environments.
The total amount of VRAM occupied by the **traditional approach** is:

$$M_{\text{trad}} = N \times (o_{resnet} + C_{context}) \tag{4.6}$$

The total amount of VRAM occupied by the **modular approach** is:

$$M_{\text{modular}} = M_{\text{shared}} + M_{\text{specialized}} \tag{4.7}$$

$$M_{\text{specialized}} = \sum_{i=1}^{N} \sum_{j=1}^{|B|} (o_j + C_{context}) \times |H_{ij}| \tag{4.8}$$

Assuming that $B_{shared} = \{b_1, b_2, b_3\}$:

$$M_{\text{shared}} = \sum_{j \in B_{shared}} (o_j + C_{context}) \tag{4.9}$$

where $M_{\text{specialized}}$ is the total VRAM occupied by task-specific blocks (e.g, specialized blocks for a target application), M_{shared} is the total VRAM occupied by shared blocks.

The task-specific requirements of the N tasks admitted cause the algorithmic response of the resource optimization engine (e.g., OffloaDNN) to differ from task to task and, consequently, the number of shared blocks will also differ.
Hence, each task will have its own number of specialized blocks out of a total number of blocks required for the forward pass. By observing the system in a generic instant t, the total number of specialized block is:

$$N_{\text{specialized\_blocks}} = \sum_{j \in T} \sum_{k \in O} \begin{cases} 1 & \text{if } H_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

Let us introduce the **Path Sharing Index** for a task $\tau_i \in T$ as such that:

$$PSI_i\ (\%) = \left(\frac{N_{\text{specialized\_blocks}}}{N_{process}} \right) * 100 \tag{4.11}$$

Scenario	Class A	Class B	Class C
Scenario 1	33%	33%	33%
Scenario 2	10%	30%	60%
Scenario 3	5%	20%	75%
Scenario 4	5%	5%	90%
Scenario 5	1%	4%	95%
Scenario 6	60%	30%	10%
Scenario 7	75%	20%	5%
Scenario 8	90%	5%	5%

Table 4.4: Task distribution across different classes, for the considered scenarios.

In a simplistic scenario that could mimic real system working conditions, we can assume that the admitted tasks can belong to 3 different service level classes. Each class, is associated with a PSI value, as follow:

- **Class A -** $PSI_i = 0$: specialized full-DNN (ResNet) for that task τ_i;
- **Class B -** $PSI_i = 50$: 2 blocks specialized out of 4 blocks τ_i;
- **Class C -** $PSI_i = 75$: 3 blocks specialized out of 4 blocks τ_i;

4.3.2 Numerical Comparison

Now, let us consider 8 different operational scenarios for partitioning (%) the N admitted task into service classes. Table 4.4 reports the class distribution scenarios tests. The analysis spans over different scenarios, with particular focus on the variation in the number of highly specialized tasks (scenarios 6, 7, 8) compared to

Scenario	MOD. (MB)	MOD. W/ B12 FUSED (MB)	MOD. W/ B12 & B34 FUSED (MB)	TRAD. DEP. (MB)	MOD. SAVINGS	MOD. W/ B12 FUSED SAVINGS	MOD. W/ B12 & B34 FUSED SAVINGS
scenario 1	64503	64113	51078	54100	-19,23%	-18,51%	5,59%
scenario 2	62239	61849	49999	54100	-15,04%	-14,32%	7,58%
scenario 3	57889	57299	49399	54100	-6,63%	-5,91%	8,69%
scenario 4	51344	50954	48979	54100	5,09%	5,82%	9,47%
scenario 5	50188	49798	48218	54100	7,23%	7,95%	10,87%
scenario 6	65439	65049	53199	54100	-20,96%	-20,24%	1,67%
scenario 7	62169	61779	53879	54100	-14,91%	-14,19%	0,41%
scenario 8	56784	56394	54419	54100	-4,96%	-4,24%	-0,59%

Table 4.5: ResNet50 modular vs traditonal deployment VRAM occupancy analysis.

a low number of tasks with low latency and accuracy constraints (class C tasks) and vice versa (scenarios 2,3,4,5). The results were obtained with N=100 and N = 1000, although no particular differences were found. Tables 4.5 and report and 4.6 the numerical results for all the tested scenarios with N=100.

For each scenario, the amount of VRAM memory occupied by the required blocks in absolute terms was computed according to the equations presented and with the data collected from the system runs and therefore compared with respect to the one that a traditional deployment would have occupied. Moreover, results of a slight variation of the modular system is presented with a small optimization in the blocks deployment that takes into account the possibility to fuse block1 and block2 and/or block3 and block4. Notice that, it is always possible to fuse block1 and block2 due to our assumption that the minimum Path Sharing Index is 50% but on the other hand, fusion of block3 and block 4 can be performed only to *Class B* tasks since they require both block3 and block4 to be fine-tuned and dedicated. The idea of block fusion stems from the need to reduce CUDA context initialization overhead by amortizing it over bigger blocks or by reducing the number or deployed processes in absolute terms.
For completeness we provide the mathematical formulation of the optimized model deployment with block fusion. In case of only b_1 & b_2 fused, the only term updated is the M_{shared}:

$$M_{\text{shared}} = (o_1 + o_2) \times C_{context} + o_3 \times C_{context} \tag{4.12}$$

In case of b_3 & b_4 merging the updated formulation is:

$$M_{\text{specialized}} = \sum_{j \in T} \left[\sum_{k \in O} o_k \times |H_{jk}| \right] + C_{context} \tag{4.13}$$

$$M_{\text{shared}} = (o_1 + o_2) \times C_{context} + o_3 \times C_{context} \tag{4.14}$$

Table 4.5 reports the main outcomes of the experiments and provide a complete overview of the numerical results. The best possible operating conditions for the modular system are found to be in scenario 4 and scenario 5 where most of the system admitted tasks belong to *Class C* and thus the system can benefit of the maximum Path Sharing Index (75%). Indeed, with a modular deployment, saving in terms of total average VRAM occupancy is 5.09% in Scenario 4 and 7.23% in Scenario 5 for the modular deployment with no block-fusion. Moreover, numerical results presents significant VRAM occupancy reduction when blocks are fused. In particular, in almost all the tested scenarios, when Block1 & Block2 and Block3 & Block4 are fused (according to what discussed above) is registered an average VRAM saving up to ~25% (scenario 1, from -19.23% to +5.59%).

Scenario	**MOD. (MB)**	**MOD. W/ B12 FUSED (MB)**	**MOD. W/ B12 & B34 FUSED (MB)**	**TRAD. DEP. (MB)**	**MOD. SAVINGS**	**MOD. W/ B12 FUSED SAVINGS**	**MOD. W/ B12 & B34 FUSED SAVINGS**
scenario 1	74329	73940	60321	66830	-11,22%	-10,64%	9,74%
scenario 2	63027	68938	56557	66830	5,69%	-3,15%	15,37%
scenario 3	63048	62660	54406	66830	5,66%	6,24%	18,59%
scenario 4	54893	54504	52441	66830	17,86%	18,44%	21,53%
scenario 5	53176	52787	51136	66830	20,43%	21,01%	23,48%
scenario 6	77742	77353	64972	66830	-16,33%	-15,75%	2,78%
scenario 7	74829	74441	66187	66830	-11,97%	-11,39%	0,96%
scenario 8	69198	68801	66746	66830	-3,54%	-2,96%	0,13%

Table 4.6: ResNet152 modular vs traditonal deployment VRAM occupancy analysis.

For a better understanding, an additional numerical comparison between modular deployment and traditional was carried out, this time considering a bigger DNN architecture. The main idea was to assess if bigger blocks in terms of number of parameters would have contributed to better amortize CUDA context overhead occupation. For this purpose, **ResNet152** was selected, split in 4 blocks and tested. Experimental results are reported in Table 4.6.

Clearly, the gains of the modular DNN deployment are also significantly evident in scenarios 2 and 3 that see an higher portion of specialized task for which is required less shared blocks. According to that line, in scenario 4 and 5 where the majority of the task allows for high Task Sharing Index, average VRAM occupancy savings reach up to ~**20,5%** and arrive to **23.48**% if block fusion is enabled (when possible) compared to traditional full-DNN deployment.

In conclusion, bigger models can benefit more from modular deployment if one of the main goal is to maximize the number of logical "forward pass" offered to

clients and thus, in an offloading scenario, the task acceptance rate as it is stated in OffloaDNN.

Chapter 5

Conclusions & Future research

Offloading computationally-intensive AI tasks, particularly in Computer Vision, to more powerful devices is a valid strategy for enabling services that would otherwise be unusable on resource-constrained hardware. This research work focuses on the development of BlockFlow, an open-source tool for the deployment of DNN models at the Edge.
The main focus was to meet the requirements of high modularity and flexibility of an inference system based on one or more DNN architectures distributed on the same machine or on multiple hosts. Given these requirements, the best option was to develop a dynamic multi-processing system whose management is handled by BlockFlow based on algorithmic responses from a resource optimization engine such as the one presented in OffloaDNN.
Indeed, BlockFlow bridges the gap between a purely algorithmic solution and the real-world deployment allowing innovative research approaches to be validated.
A series of comprehensive experiments show that the communication latency between the blocks constituting a neural network structure plays a crucial role in overall performance. In particular, communication that is not optimized for tensor forwarding can cause overheads in terms of latency that can be up to 200% beyond the baseline inference time.
The main contribution of this work was to implement a high-performance communication system for the transmission of intermediate tensors between the various blocks deployed by BlockFlow, called TensorMQ.
TensorMQ is based on the open-source ZeroMQ library and implements a GPU-aware communication pattern, ensuring extremely low overheads. It can operate in scenarios where two consecutive blocks in the pipeline are on the same host and GPU, on the same host but on separate GPUs, and potentially even on different

hosts as long as they are connected to the internet.
Experimental results confirm that using TensorMQ compared to ZeroMQ can reduce the overheads of a ResNet50-based pipeline of 4 blocks by up to ~**8.30x**. This reduction in total inference time improves the utilization of computational resources. From an offloading perspective, it also increases the number of tasks the system can accept per unit of time. Finally, more inferences performed per unit of time with the same computational resources allow for better amortization of the fixed energy consumption.

However, although BlockFlow with TensorMQ makes a novel contribution in the field of optimized solutions for high-performance and practical Edge offloading, sharing one or more blocks between one or more tasks leads to inefficiencies in terms of memory usage. In fact, experiments have shown that each process requires a significant amount of RAM (~700MB) but, above all, VRAM overhead due to CUDA context initialization (~370MB). Although there are solutions that might reduce this overhead in absolute terms, this type of design based on the block-process relationship remains inefficient for blocks with a relatively small number of parameters. Indeed, bigger models, and hence bigger blocks in terms of number of parameters can, mitigate this issue and better benefit from a modular deployment.

Future investigations can be split into 2 main research paths: one possibility is to test the feasibility and the performance of a single-process application with different threads, analyzing how the system behaves in high traffic scenarios sacrificing a relatively small amount of flexibility on the deployment. This design approach might lead to a lower system resource consumption. On the other hand, an improvement in terms of performance might be achieved by introducing a dynamic batching capability within the data forwarding plane. Accordingly, we might observe a dramatic increase in the overall system throughput while maintaining good latency performance using TensorMQ.
Another scenario involves the introduction of optimization policies before the actual deployment of the blocks, considering the possibility of joining two or more blocks within a single process, ensuring the best tradeoff in terms of minimization of the context initialization overhead while respecting the deployment strategy and the task service level constraints.

Bibliography

[1] Zhuo Li, Hengyi Li, and Lin Meng. «Model compression for deep neural networks: A survey». In: *Computers* 12.3 (2023), p. 60 (cit. on p. 1).

[2] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. *Mobile Edge Computing – A Key Technology Towards 5G*. Tech. rep. White Paper No. 11. Sophia Antipolis, France: European Telecommunications Standards Institute (ETSI), Sept. 2015. URL: `https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf` (cit. on pp. 2, 19, 20).

[3] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. «Edge computing: A survey». In: *Future Generation Computer Systems* 97 (2019), pp. 219–235 (cit. on p. 2).

[4] Marcelo VB da Silva, Maria Barbosa, Anderson Queiroz, and Kelvin L Dias. «A 5G-Edge Architecture for Computational Offloading of Computer Vision Applications». In: *arXiv preprint arXiv:2501.04267* (2025) (cit. on p. 2).

[5] Matthias Frei, Piotr Karbownik, Reinhard German, and Anatoli Djanatliev. «Accessing the Edge: Delay Evaluation to Distributed Edge Services in a City-Level 5G Network». In: *2024 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2024, pp. 197–205 (cit. on p. 2).

[6] Chetna Singhal, Yashuo Wu, Francesco Malandrino, Marco Levorato, and Carla Fabiana Chiasserini. «Distributing Inference Tasks Over Interconnected Systems Through Dynamic DNNs». In: *IEEE Transactions on Networking* (2025) (cit. on p. 3).

[7] Jiawei Shao and Jun Zhang. «Communication-computation trade-off in resource-constrained edge inference». In: *IEEE Communications Magazine* 58.12 (2021), pp. 20–26 (cit. on p. 3).

[8] Juliano S Assine, José Cândido Silveira Santos Filho, Eduardo Valle, and Marco Levorato. «Slimmable encoders for flexible split dnns in bandwidth and

resource constrained iot systems». In: *2023 IEEE 24th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE. 2023, pp. 1–9 (cit. on p. 4).

[9] Corrado Puligheddu, Nancy Varshney, Tanzil Hassan, Jonathan Ashdown, Francesco Restuccia, and Carla Fabiana Chiasserini. «OffloaDNN: Shaping DNNs for Scalable Offloading of Computer Vision Tasks at the Edge». In: *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2024, pp. 624–634 (cit. on pp. 4, 21, 23, 24).

[10] Khadijeh Alibabaei, Pedro Gaspar, Tânia Lima, Rebeca Campos, Inês Girão, Jorge Bordalo Monteiro, and Carlos Lopes. «A Review of the Challenges of Using Deep Learning Algorithms to Support Decision-Making in Agricultural Activities». In: *Remote Sensing* 14 (Jan. 2022), p. 638. DOI: 10.3390/rs14030638 (cit. on p. 14).

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385 (cit. on p. 15).

[12] Jack Bell. *Operating Systems: Threads*. https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html. n.d. (Cit. on p. 18).

[13] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. «Mobile edge computing: Survey and research outlook». In: *arXiv preprint arXiv:1701.01090* (2017), pp. 1–37 (cit. on p. 21).

[14] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. «What is being transferred in transfer learning?» In: *Advances in neural information processing systems* 33 (2020), pp. 512–523 (cit. on p. 22).

[15] Corrado Puligheddu, Jonathan Ashdown, Carla Fabiana Chiasserini, and Francesco Restuccia. «SEM-O-RAN: Semantic and flexible O-RAN slicing for NextG edge-assisted mobile systems». In: *IEEE Infocom 2023-IEEE Conference on Computer Communications*. IEEE. 2023, pp. 1–10 (cit. on p. 25).

[16] PyTorch Team. *Pipeline Parallelism — PyTorch Distributed Documentation*. https://docs.pytorch.org/docs/stable/distributed.pipelining.html. 2025 (cit. on p. 29).

[17] Pritam Damania et al. «Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls». In: *Proceedings of Machine Learning and Systems* 5 (2023), pp. 219–231 (cit. on p. 30).

[18] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013 (cit. on p. 30).

[19] Vishesh Narendra Pamadi, Dr Ajay Kumar Chaurasia, and Dr Tikam Singh. «Comparative Analysis OF GRPC VS. ZeroMQ for Fast Communication». In: *International Journal of Emerging Technologies and Innovative Research (www. jetir. org)* 7.2 (2020), pp. 937–951 (cit. on p. 31).

[20] Mekhriddin Rakhimov, Doniyor Mamadjanov, and Abulkosim Mukhiddinov. «A high-performance parallel approach to image processing in distributed computing». In: *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE. 2020, pp. 1–5 (cit. on p. 31).

[21] Joel Lauener, Wojciech Sliwinski, and Geneva CERN. «How to design & implement a modern communication middleware based on ZeroMQ». In: *Proc of ICALEPCS*. Vol. 17. 2017, pp. 45–51 (cit. on p. 31).

[22] Ties Robroek, Neil Kim Nielsen, and Pınar Tözün. «TensorSocket: Shared Data Loading for Deep Learning Training». In: *arXiv preprint arXiv:2409.18749* (2024) (cit. on p. 31).

[23] PyTorch Team. *TorchScript — PyTorch JIT Documentation*. `https://docs.pytorch.org/docs/stable/jit.html`. 2024 (cit. on p. 31).

[24] CherryPy Team. *CherryPy - A Minimalist Python Web Framework*. `https://cherrypy.dev/`. 2024 (cit. on p. 32).

[25] Giampaolo Rodola. *psutil: Cross-platform process and system utilities*. `https://psutil.readthedocs.io/en/latest/`. 2024 (cit. on p. 33).

[26] NVIDIA Corporation. *pynvml: Python Bindings for the NVIDIA Management Library*. `https://pypi.org/project/pynvml/`. 2024 (cit. on p. 33).

[27] NVIDIA Corporation. *CUDA Profiler User's Guide*. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`. 2024 (cit. on p. 33).

[28] NVIDIA Corporation. *CUDA Runtime API Documentation*. Accessed: July 13, 2025. 2025. URL: `https://docs.nvidia.com/cuda/cuda-runtime-api` (cit. on p. 34).

Appendix A

Awenode GPU topology

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	**CPU Affinity**	**NUMA Affinity**	**GPU NUMA ID**
GPU0	X	NODE	NODE	NODE	SYS	SYS	SYS	SYS	0–31,64–95	0	N/A
GPU1	NODE	X	NODE	NODE	SYS	SYS	SYS	SYS	0–31,64–95	0	N/A
GPU2	NODE	NODE	X	NODE	SYS	SYS	SYS	SYS	0–31,64–95	0	N/A
GPU3	NODE	NODE	NODE	X	SYS	SYS	SYS	SYS	0–31,64–95	0	N/A
GPU4	SYS	SYS	SYS	SYS	X	NODE	NODE	NODE	32–63,96–127	1	N/A
GPU5	SYS	SYS	SYS	SYS	NODE	X	NODE	NODE	32–63,96–127	1	N/A
GPU6	SYS	SYS	SYS	SYS	NODE	NODE	X	NODE	32–63,96–127	1	N/A
GPU7	SYS	SYS	SYS	SYS	NODE	NODE	NODE	X	32–63,96–127	1	N/A

Table A.1: GPU Topology and Affinity (multi-GPU-server)

Appendix B

ZeroMQ vs TensorMQ time distributions per Block

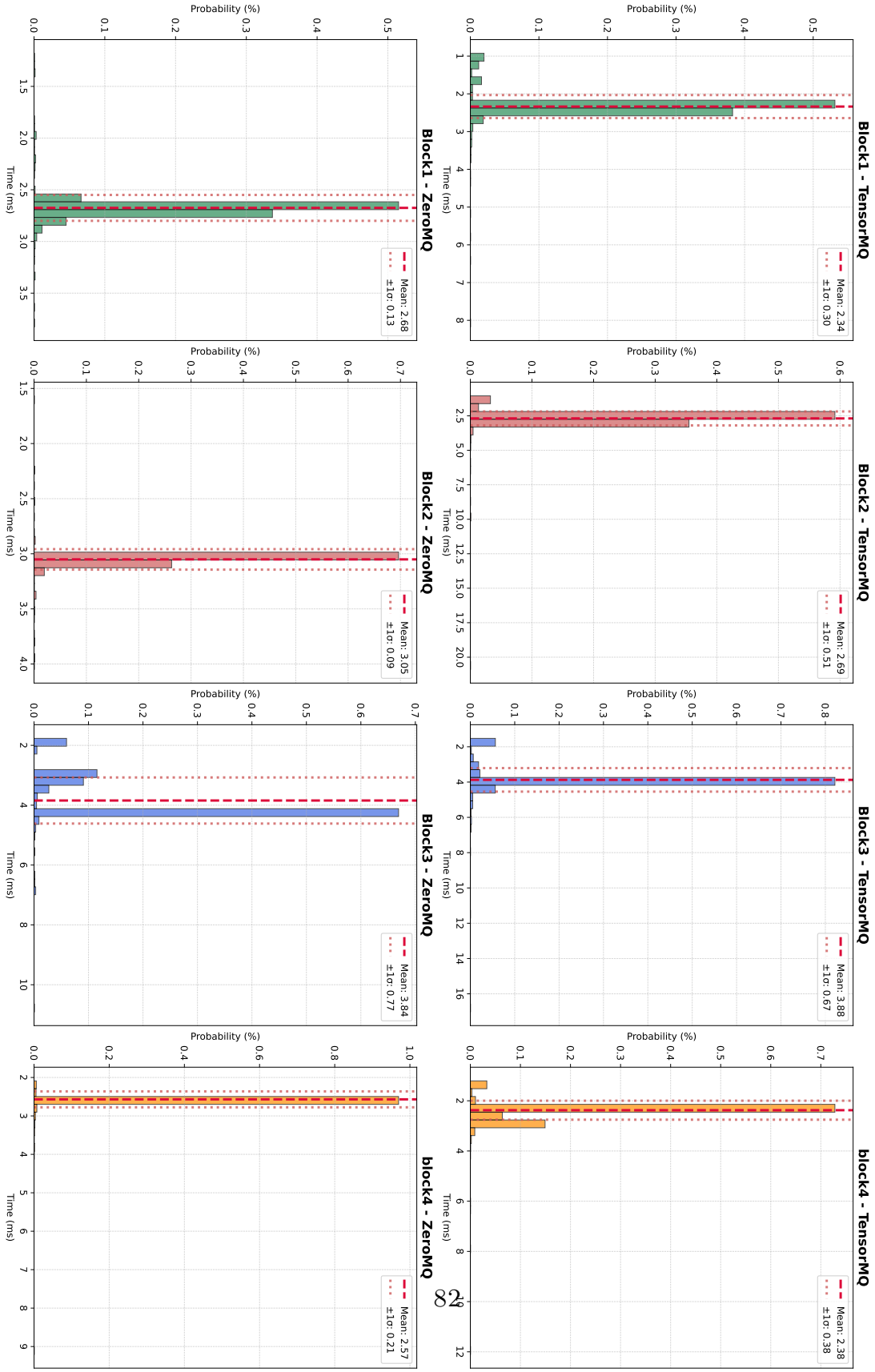

Figure B.1: ZeroMQ vs TensorMQ inference time distributions per block

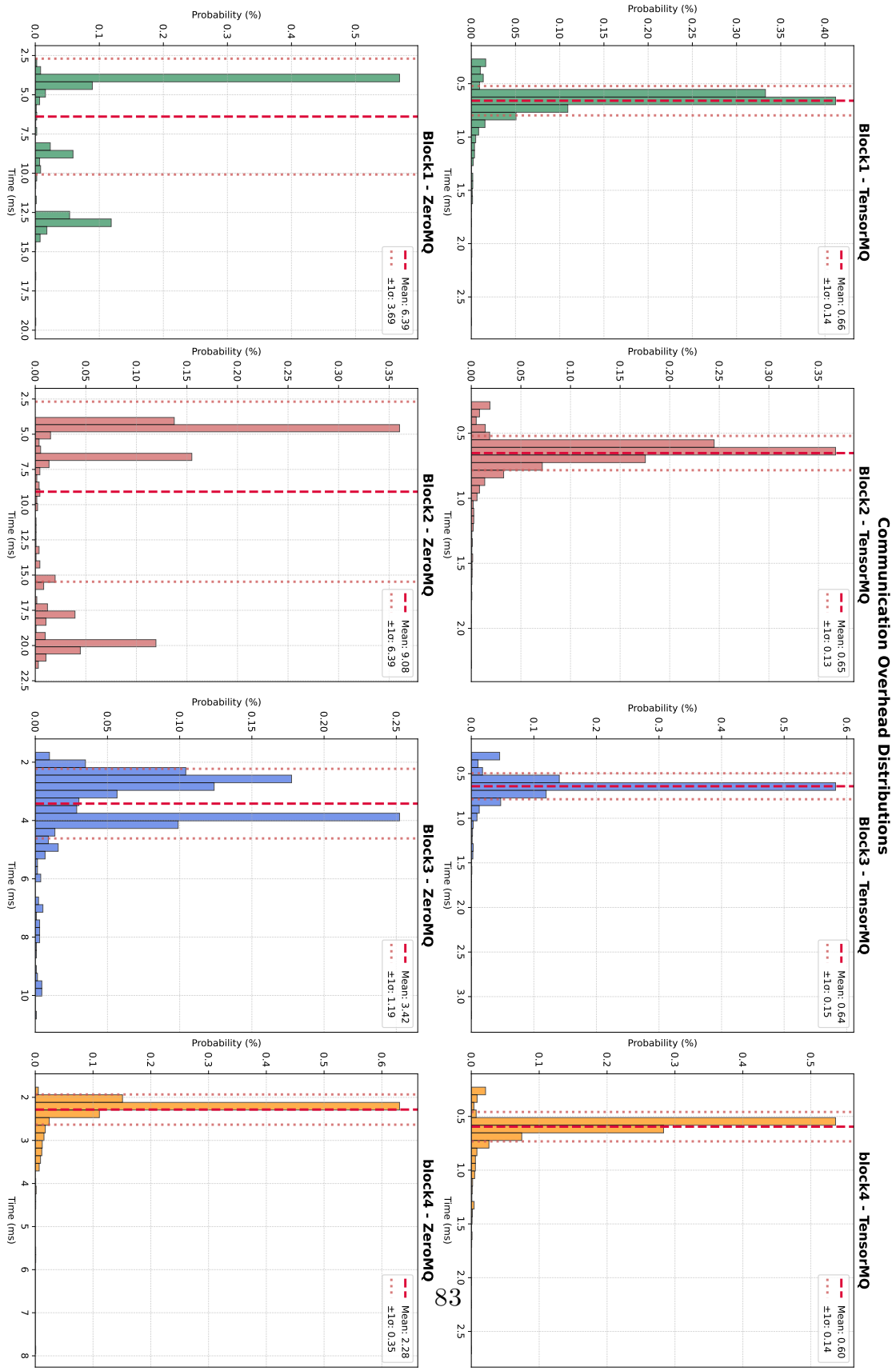

Figure B.2: ZeroMQ vs TensorMQ overhead time distributions per block