

Diese Arbeit wurde vorgelegt am  
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

# Vergleich und Modellierung der Leistung verschiedener ML-Frameworks und Hardwarebeschleuniger in einer gekoppelten OpenFoam+ML-Applikation

## Comparing and Modelling the Performance of Different ML Frameworks and Hardware Accelerators in a Coupled OpenFoam+ML Application Masterarbeit

Kim Sebastian Brose  
Matrikelnummer: 322935

Aachen, den 1. Juni 2022

Erstgutachter: Prof. Dr. rer. nat. Matthias S. Müller (')  
Zweitgutachter: Prof. Dr. Christian Hasse (\*)  
Betreuer: Fabian Orland, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University  
IT Center, RWTH Aachen University

(\*) Simulation reaktiver Thermo-Fluid Systeme, TU Darmstadt

Communicated by Prof. Dr. rer. nat. Matthias S. Müller



Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 1. Juni 2022



# Kurzfassung

Wir untersuchen und modellieren die Leistung einer mit Machine Learning gekoppelten HPC Anwendung. Sie basiert auf einer reaktiven Thermo-Fluid-Simulation, welche durch den Speicherbedarf einer dem Stand der Technik entsprechenden Implementierung mit einer Wertetabelle limitiert ist. ML ist ein altes Konzept, das in den letzten Jahren durch die Entwicklung leistungsstarker ultraparalleler Prozessoren mit Vektorisierungsfähigkeit vorangetrieben wurde, denn diese können die Berechnung von für ML Algorithmen typischen Matrixoperationen beschleunigen. Die Implementierung mit Wertetabelle wird durch ein ML Modell ersetzt, indem sie durch Kopplungen, welche die bestehende Software mit verschiedenen ML Frameworks verknüpfen, ausgetauscht wird. Zum Vergleich implementieren wir Kopplungen mit bekannten und verbreiteten sowie vielversprechenden neuartigen Frameworks, welche teils unterschiedliche Hardware, darunter CPUs, GPUs und Vector Engines, oder Herangehensweisen unterstützen.

Wir messen und modellieren die Leistung der verschiedenen Kopplungen unter verschiedenen Bedingungen. Wenn wir das ML Modell vergrößern, steigt die benötigte Laufzeit wesentlich langsamer als die quadratisch wachsende Komplexität der Rechnung. Die wiederholte Ausführung der Inferenz für verschiedene Eingangsdaten kann durch Stapelverarbeitung beschleunigt werden, wenn die Stapelgröße so gewählt wird, dass der resultierende Arbeitsaufwand die unteren sowie oberen Grenzen der spezifischen Hardwarearchitektur des benutzten Gerätes berücksichtigt.

Wir prüfen die Leistung der Kopplungen auf starke Skalierbarkeit nach Amdahl. Der ML Teil der Applikation ist de facto komplett datenparallel und skaliert entsprechend optimal durch Hinzufügen weiterer Prozessoren. Der Mehraufwand durch die Kommunikation steigt, wenn wir dem HPC-Teil der gekoppelten Applikation mehr MPI Prozesse geben, aber ist im Allgemeinen vernachlässigbar im Vergleich zur eigentlichen Laufzeit des ML Frameworks. Obwohl die Berechnung auf GPUs schneller ist, entdecken wir, dass es unter gewissen Umständen effizienter sein kann, den ML Teil auf CPUs auszuführen. Mit unserem ML Modell erreichen die GPUs am ehesten ihre Spitzenleistung, gefolgt von CPUs, während die Vector Engines am weitesten davon entfernt sind.

Wir untersuchen die vom EU Center of Excellence POP etablierten Leistungsmetriken als Modell für unsere gekoppelte HPC+ML-Applikation in den verschiedenen Skalierungsszenarios. Die Kopplungen haben alle eine verhältnismäßig geringe Laufzeit im Vergleich zur gesamten Applikation und daher keinen klar erkennbaren Einfluss auf die für die ganze Applikation gemessenen Metriken. Eine genauere Analyse wird benötigt um eindeutigere Ergebnisse zu liefern.

**Stichwörter:** HPC, MPI, Machine Learning, Vector Engine



# Abstract

We examine and model the performance of a coupled HPC+ML application. It is based on a reactive thermo-fluid simulation which is limited by the memory footprint of the lookup table that is integral to the state of the art implementation. Machine learning is an old concept that has become more attainable in the recent years thanks to the advent of powerful ultra parallel processors and vectorization capabilities that allow efficient computation of matrix operations common to machine learning algorithms. The lookup table implementation is replaced with a machine learning model by implementing couplings that connect the existing software to established as well as novel machine learning frameworks that use different kinds of hardware such as CPUs, GPUs and vector engines.

We measure and model the performance of the different couplings under different conditions. As we increase the size of the machine learning model, the runtime increases significantly less than the quadratically increasing computational complexity. The execution of the model for multiple input data samples can be accelerated by choosing a batch size such that the resulting amount of work fits the lower and upper bounds given by the architecture of the hardware accelerator device.

We examine the coupling's performance with regard to strong scaling. Since the machine learning part is practically completely parallelized, it scales perfectly as we add more devices. The communication overhead becomes measurable as we add more MPI ranks to the HPC part of the application, but remains insignificant compared to the actual framework time. We discover that under certain circumstances, it can be more efficient to run the machine learning framework on CPUs despite GPUs generally performing faster. Overall, the GPUs are by far the closest to reaching their peak performance with our application, followed by CPUs, and VEs are the farthest from reaching their peak performance.

We investigate performance metrics established by the EU Center of Excellence POP as a performance model for the coupled HPC+ML application in the different scaling settings. The couplings have a small run time compared to the whole application, and thus no discernible impact on the metrics measured for the whole application. A more in-depth analysis is required to obtain conclusive results.

**Keywords:** HPC, MPI, Machine Learning, Vector Engine



# Contents

|  |             |
|--|-------------|
| <b>List of Figures</b>   | <b>xi</b>   |
| <b>List of Tables</b>  | <b>xiii</b> |
| <b>List of Listings</b>  | <b>xv</b>   |
| <b>1 Introduction</b>  | <b>1</b>    |
| <b>2 Overview</b>  | <b>3</b>    |
| 2.1 The Base Application . . . . .                                     | 3           |
| 2.2 Optimizing the Application with Machine Learning . . . . .         | 4           |
| <b>3 Frameworks Coupling</b>   | <b>9</b>    |
| 3.1 Design of the Coupling Interface . . . . .                         | 9           |
| 3.2 General Consideration of Machine Learning Framework Interfaces . . | 10          |
| 3.2.1 Converting Models . . . . .                                      | 11          |
| 3.2.2 Tensor Format . . . . .  | 11          |
| 3.2.3 Batching and Batch Size Estimation . . . . .                     | 13          |
| 3.3 PyTorch . . . . .  | 14          |
| 3.4 TensorFlow . . . . .   | 16          |
| 3.5 Nvidia Triton . . . . .  | 21          |
| 3.6 Torch-TensorRT . . . . .   | 25          |
| 3.7 SOL . . . . .  | 27          |
| <b>4 Performance Model</b>   | <b>29</b>   |
| 4.1 POP Model . . . . .  | 29          |
| 4.2 POP Model for Hybrid Codes . . . . .                               | 30          |
| <b>5 Results</b>   | <b>33</b>   |
| 5.1 Hardware, Framework and Library Versions . . . . .                 | 33          |
| 5.2 Instrumenting for Profiling and Tracing . . . . .                  | 35          |
| 5.2.1 Nvidia GPUs . . . . .  | 35          |
| 5.2.2 NEC Vector Engines . . . . .                                     | 36          |
| 5.2.3 Extrae . . . . .   | 37          |
| 5.3 Measurements with GPU-Accelerated Frameworks . . . . .             | 37          |
| 5.3.1 GPU Baseline Measurements . . . . .                              | 38          |
| 5.3.2 Homogeneous Scaling with GPUs . . . . .                          | 47          |
| 5.3.3 Heterogeneous Scaling with GPUs . . . . .                        | 51          |

## Contents

|          |  |           |
|----------|--|-----------|
| 5.4      | Measurements with VE-accelerated SOL . . . . . | 55        |
| 5.4.1    | VE Baseline Measurements . . . . .             | 55        |
| 5.4.2    | Homogeneous Scaling with VEs . . . . .         | 58        |
| <b>6</b> | <b>Evaluation</b>                              | <b>61</b> |
| 6.1      | GPU-Accelerated Frameworks . . . . .           | 61        |
| 6.1.1    | GPU Performance . . . . .                      | 61        |
| 6.1.2    | POP Efficiency . . . . .                       | 65        |
| 6.2      | VE-Accelerated Framework . . . . .             | 66        |
| <b>7</b> | <b>Conclusions</b>                             | <b>69</b> |
| <b>8</b> | <b>Future Work</b>                             | <b>73</b> |
|          | <b>Bibliography</b>                            | <b>79</b> |

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Schema of a multilayer perceptron (MLP)            | 4  |
| 2.2  | Creating workgroups by GPU-based coloring          | 6  |
| 2.3  | Creating workgroups with round robin assignment    | 7  |
| 2.4  | Communication and processing steps in the coupling | 8  |
| 3.1  | The conceptual interface for the coupling          | 9  |
| 3.2  | MPI concatenates row-major ordered arrays          | 13 |
| 3.3  | Nvidia Triton application flow schema              | 22 |
| 3.4  | Triton communication model “as-close-as-possible”  | 23 |
| 4.1  | POP model  | 30 |
| 4.2  | Hybrid POP model                                   | 32 |
| 5.1  | Optimization workflow using Nvidia Nsight tools    | 36 |
| 5.2  | Warm-up on GPUs                                    | 39 |
| 5.2  | Warm-up on GPUs (cont.)                            | 40 |
| 5.3  | Batch size on GPUs                                 | 44 |
| 5.4  | Hidden layer size on GPUs                          | 45 |
| 5.5  | Communication overhead on GPUs                     | 46 |
| 5.6  | Communication overhead, homogeneous GPU scaling    | 47 |
| 5.7  | Actual framework time, homogeneous GPU scaling     | 48 |
| 5.8  | POP metrics, homogeneous GPU scaling               | 50 |
| 5.9  | Communication overhead, heterogeneous GPU scaling  | 51 |
| 5.10 | Actual framework time, heterogeneous GPU scaling   | 53 |
| 5.11 | POP metrics, heterogeneous GPU scaling             | 54 |
| 5.12 | Warm-up on VEs                                     | 55 |
| 5.13 | Batch size and hidden layer size on VEs            | 56 |
| 5.14 | Communication overhead on VEs                      | 57 |
| 5.15 | Actual framework time, VE scaling                  | 59 |
| 8.1  | Triton architecture without MPI                    | 75 |
| 8.2  | Triton architecture with load balancer             | 75 |



# List of Tables

- 3.1 Unexpected results when using TensorFlow . . . . . 11
- 5.1 Missing coupling timings . . . . . 38
- 5.2 Missing POP metrics . . . . . 38
- 5.3 PyTorch GPU kernels . . . . . 41
- 5.4 TensorFlow GPU kernels . . . . . 42
- 5.5 Torch-TensorRT GPU kernels . . . . . 42
- 5.6 Hidden layer scaling . . . . . 56
- 5.7 SOL VE kernels . . . . . 57



# List of Listings

|      |   |    |
|------|---|----|
| 3.1  | Converting the OpenFOAM field-first data format to batch-first . . .                      | 12 |
| 3.2  | Syntax of MPI vector operations with varying length [23] . . . . .                        | 13 |
| 3.4  | Initialization of the PyTorch coupling . . . . .  | 14 |
| 3.3  | Creating our neural network model in PyTorch using Python . . . . .                       | 15 |
| 3.5  | MPI in the PyTorch coupling . . . . .   | 15 |
| 3.6  | Batch processing in the PyTorch coupling . . . . .  | 16 |
| 3.7  | Creating our neural network model in TensorFlow using Python . . .                        | 17 |
| 3.8  | Serializing our configuration using Python . . . . .                                      | 17 |
| 3.9  | Obtaining hardcoded values using <code>saved_model_cli</code> . . . . .                   | 18 |
| 3.11 | MPI in the TensorFlow coupling . . . . .  | 18 |
| 3.10 | Initialization of the TensorFlow coupling . . . . .                                       | 19 |
| 3.12 | Batch processing the TensorFlow coupling . . . . .  | 20 |
| 3.13 | Initialization of the Triton coupling . . . . .   | 24 |
| 3.14 | <code>MPI_Gatherv</code> in the Triton coupling . . . . .                                 | 24 |
| 3.15 | Sending an inference request in the Triton coupling . . . . .                             | 25 |
| 3.16 | <code>MPI_Scatterv</code> in the Triton coupling . . . . .                                | 25 |
| 3.17 | Creating our neural network model for Torch-TensorRT in PyTorch<br>using Python . . . . . | 26 |
| 3.18 | Initialization of the Torch-TensorRT coupling . . . . .                                   | 26 |
| 3.19 | MPI in the Torch-TensorRT coupling . . . . .  | 27 |
| 3.20 | Creating our neural network model in SOL and PyTorch using Python                         | 27 |
| 3.21 | <code>my_wrapper</code> manages memory between host and device . . . . .                  | 28 |
| 3.22 | Initialization of the SOL coupling . . . . .  | 28 |
| 3.23 | Batch processing the SOL coupling . . . . .   | 28 |
| 5.1  | Region markers with NVTX (PyTorch example) . . . . .                                      | 37 |



# 1 Introduction

Machine learning (ML) is a computer science discipline based on the idea of self-adjusting algorithms. A machine learning model “learns” a concept based on example input and output data by adjusting itself so the input data when fed into the algorithm would result in the output data. This process is also called “training” of a machine learning model. Once it has adapted to a certain use case, a model is able to “infer” or “predict” the matching output datum based on an input datum, even if it was never trained with this exact input datum. More complex tasks require more complex machine learning models, often inspired by biological neural networks (NN) and encompassing “deep” networks of multiple layers. Today, machine learning is prominently evolving thanks to the availability of powerful highly parallel processors such as (general purpose) graphics processing units ((GP)GPUs) that are well fit to compute the large matrix calculations required by the training of as well as inference using machine learning models. Hardware that is particularly fit for a certain type of calculations is also called a Hardware Accelerator. Some well known and established use cases include image recognition and computer vision for content annotation and self localization in self-driving cars, image synthesis for “deepfakes” or “filters”, speech and voice recognition as well as synthesis for digital assistants, and natural language translation. Many computer science and software engineering fields that are not employing machine learning already, are exploring how to leverage its power for their use cases.

We see three strong new possible use cases for machine learning in High Performance Computing (HPC).

- As usage of machine learning in research and engineering spreads, operators of HPC clusters may wish to make it as accessible and efficient to use to their users as possible [30].
- Machine learning can be applied to analyze and improve HPC deployments, including smarter scheduling and resource usage, suggesting code performance and correctness improvements, and data mining in support of operations [19].
- Replace parts of classic HPC programs with more time, space, cost, or energy efficient [18] and sufficiently accurate machine learning inference [30].

This thesis focuses on the third aspect in form of an explorative analysis, comparing the performance of some established and some new machine learning frameworks in the context of a HPC+ML application.

## 1 Introduction

We have seen performance studies of machine learning applications on HPC clusters before, e.g., [12, 18, 20], however these usually concentrate on running homogeneous machine learning applications on a cluster of compute nodes, each leveraging multiple GPUs. In contrast, we examine a given simulation application that is not fully converted to machine learning, but instead only one computationally expensive simulation step has been replaced by an alternative machine learning algorithm. Our study focuses primarily on a comparison of different machine learning frameworks, optimization tools, and different hardware types in order to find whether and which version performs vastly superior or inferior in comparison. We also investigate the heterogeneity of our application and model its performance.

The starting point to this thesis is a reactive thermo-fluid simulation test case implemented using the computational fluid dynamics (CFD) software OpenFOAM [6] at TU Darmstadt/NHR4CES. The application is part fluid simulation and part combustion simulation, the latter of which is implemented using the current state-of-the-art approach, leveraging a multi-dimensional lookup table (LUT) to perform linear interpolation of thermochemical quantities [32]. There has been research into replacing the LUT with an artificial NN in the past [14], but we reevaluate the viability of the approach based on aforementioned advances in computational capacity. The STFS research group of TU Darmstadt has implemented a prototype that replaces this established approach by a machine learning model of the thermochemical model, on which we base further experiments.

The rest of this thesis is structured as follows: We present the given base application and the optimization leveraging machine learning in detail in Chapter 2. Then we explain general considerations when implementing this optimization with machine learning frameworks as well as particular differences between each framework in Chapter 3. We apply the well-known performance model of the Performance Optimization and Productivity (POP) Center of Excellence (CoE) in Chapter 4 and present the results of our measurements in Chapter 5. We evaluate our results in Chapter 6. Finally we draw conclusions of our analysis and give an outlook on ways to expand our work in the future in Chapter 7.

## 2 Overview

### 2.1 The Base Application

We base our experiments on a simple version of a reactive thermo-fluid simulation, i.e., it simulates the movements of a mixture of multiple fluids and/or gases, as well as a combustion reaction of said mixture. It is implemented using the CFD software OpenFOAM [6] and the combustion simulation follows the method of van Oijen et al. [32]. The whole simulation is an iterative process, the next “state” of the model is based on a user configurable time step and it ends when the end time is reached. Part of the combustion simulation is a model of a flame called Flamelet Generated Manifold (FGM) that constructs a flame from one-dimensional representations of thermochemical relations called “flamelets”. There are formulas to compute these flamelets based on multiple variables, including convective, diffusive, and chemical, as well as possibly transport, curvature, and stretch properties ([32] Section 3.2.2). For runtime optimization, instead a set of “representative” flamelets, from which actual flames can be computed through interpolation, is created in advance and stored in a lookup table (LUT). Upon runtime, OpenFOAM calculates the values of the control variables, looks up the neighboring values in the LUT and performs interpolation to get the required flamelet, and finally the simulation continues with the result. The LUT accommodates for the aforementioned control variables, and thus needs one dimension for each control variable. Because the required storage is the product of the amount of sampling points in each dimension, it becomes very large very quickly.

The OpenFOAM simulation has a certain spacial domain represented by a computational mesh consisting of  $224 \times 224 \times 108 = 5\,419\,008$  “cells” by default, which are partitioned into multiple groups to enable multi core and multi node processing. Each cell in our simplified simulation has two “fields” with one double precision floating point value each. The computational mesh can be adjusted to change the precision in the configuration file `system/blockMeshDict`. The partitioning can be adjusted to the required number of Message Passing Interface (MPI) [23] ranks in the configuration file `system/decomposeParDict`. The time step granularity and simulation end time can be configured in the configuration file `system/controlDict`.

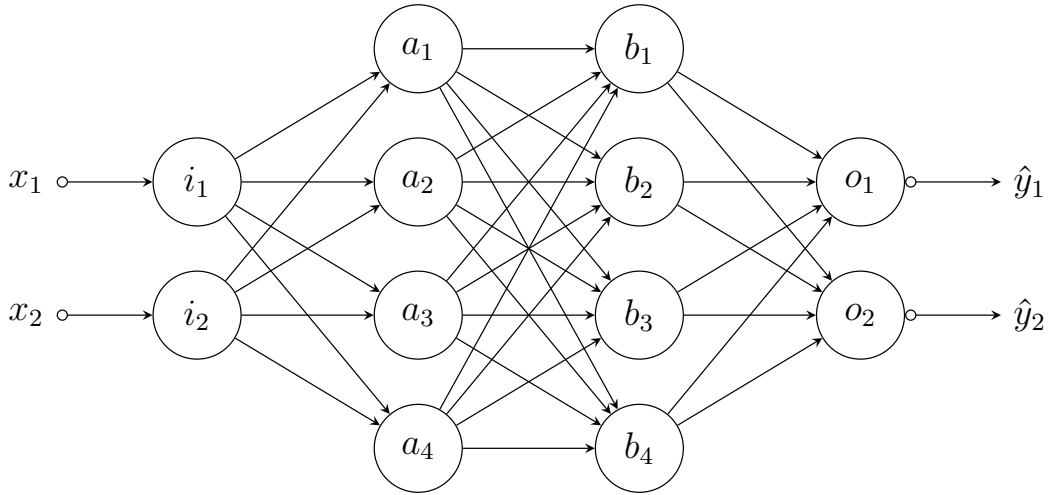


Figure 2.1: Schema of a multilayer perceptron (MLP) with two input values, two hidden layers of 4 neurons each, and an output layer with two neurons. Graphic based on [25].

## 2.2 Optimizing the Application with Machine Learning

As an alternative implementation, the LUT can be replaced by an algorithm that captures the thermochemical relations while still requiring fewer coefficients and thus memory than the full lookup table. One way to construct an algorithm that captures such complex relations is machine learning. The simplified test case provided to us already included a variant of the application where the LUT had been replaced by an artificial neural network structure called multilayer perceptron (MLP), which was already explored for this type of application by Christo et al. [14]. Figure 2.1 shows a schematic rendering of an MLP.

An MLP is one of the most basic NN structures. It consists of an input layer (the column with nodes labeled  $i_n$ ), one or more “hidden layers” (the columns with nodes labeled  $a_n$  and  $b_n$ ), and an output layer (the column with nodes labeled  $o_n$ ), hence “multilayer”. Each layer has a number of nodes which are also called “neurons”. In an MLP, each node in each layer serves as an input to each node in the consecutive layer as represented by the directed edges (arrows), e.g.,  $i_1$  and  $i_2$  are inputs for  $a_1$ , etc. Each edge is associated with a “weight” of the connection it represents, and each neuron is associated with a “bias”. In this way a neuron represents a linear combination with bias: The output value of the neurons connected via incoming edge to the regarded neuron are multiplied with the weights associated with the respective edge, the results are summed with the node’s bias. Then the nonlinear “activation function” is applied to determine whether the neuron “fires”, i.e., outputs the calculated value along the outgoing edges. This type of layer is called “fully connected”, “linear” [7], or “dense” [8]. The application of weights and biases to

the inputs can also be expressed as a vector-matrix multiplication with bias vector. Instead of vectors, it is possible to input a matrix of arbitrary dimension called a “tensor” in this context, simply expanding the calculation to use higher dimensional tensors instead of one dimensional vectors. This General Matrix Multiplication (GEMM) is a very common operation that is well supported, including hardware implementations, on devices such as GPUs [15, 1].

In the following we are going to measure the performance of different MLP configurations. We always have 2 input values and 2 output values as these are given by the number of scalar fields in the base application. We vary the amount of neurons of the hidden layers, but always use two hidden layers of equal size. We use the “Rectified Linear Unit” (ReLU) function [24] as activation function, i.e.,  $f(x) = \max\{0, x\}$ , so a positive value is output as-is, while a negative value is output as 0 value, i.e., discarded.

The size of the hidden layers has implications for our memory requirements, as we need to store the intermediate result at each layer to use it as input to the next layer. As we choose large hidden layers of hundreds or thousands of neurons, our requirement to store just two input values suddenly explodes to an amount of intermediate results equal to the amount of neurons per layer. The machine learning frameworks we examine in this thesis are able to process multiple input data samples at once, which we will regard more closely in Section 3.2.3. This mechanism called batching allows the framework to occupy the accelerator device completely thus making use of all its computational power. However, scaling the amount of input data also has a considerable memory impact. Assuming we use hidden layers with 1000 neurons and get 2 input values (1 “cell” with 2 “fields” each) in a model using double precision floating point values (8 B each), then the memory requirement of the intermediate values of one layer is 8 kB. If instead we feed 2 M input values (1 M “cells” with 2 “fields” each) in the same model, the intermediate memory requirement of one layer is

$$\frac{1000}{2} \times 2 \text{ M} \times 8 \text{ B} = 8 \text{ GB}. \quad (2.1)$$

During the calculation step from one hidden layer to the following, we have this memory requirement for both input and output for a sum of 16 GB.

The implementation given at the start of this thesis already implements one version of a ML coupling, replacing the LUT approach with an MLP, implemented using the PyTorch [26] ML framework. The compute nodes of our HPC cluster (see Section 5.1) have either two or no GPUs and 48 CPU cores, hence when occupying each node completely, we can not assign one GPU per MPI rank. Instead, the coupling partitions the available ranks into MPI “workgroups” such that in each workgroup is exactly one “GPU master” rank that collects the data from all ranks in its workgroup, runs the ML inference, and distributes the results back across the workgroup. Workgroups assignment is implemented with a naive round robin method that does not take rank locality into account, nor does it employ multi-threading, e.g., by means of OpenMP, to reduce MPI overhead when multiple ranks

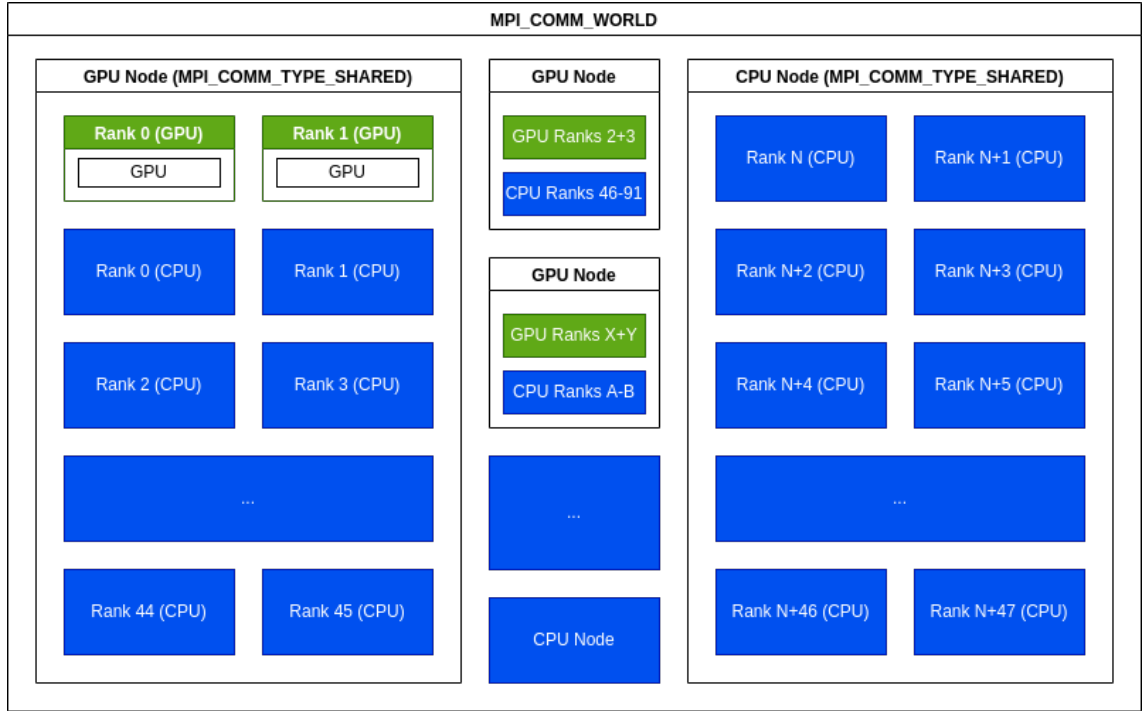


Figure 2.2: MPI ranks are colored based on their access to a GPU. Ranks with GPU are colored green, ranks without are blue.

run on a common node. This way, the OpenFOAM part of the simulation can be scaled over an arbitrary amount of CPU cores and the ML inference over an arbitrary amount of GPUs.

All ranks are colored based on whether they can access a GPU or not (Figure 2.2: GPU ranks are colored green, while CPU ranks are blue). The MPI\_COMM\_WORLD is again split using this coloring, so green ranks are on the GPU communicator, and blue ranks the CPU communicator. Ranks can now be enumerated per type: GPU master or not. In particular we now know how many workgroups there will be, and use that to assign the workgroups round robin (Figure 2.3).

At runtime, the ML inference step of the simulation works as follows. The OpenFOAM implementation, running on all ranks, calls the coupling code. All ranks know from the round robin assignment during setup whether they are a GPU master rank or not. Using the operation `MPI_Gatherv` (see Listing 3.2 in Section 3.2.2), the input data for the ML inference of each workgroup gets collected centrally on the GPU master rank. Then the GPU master rank performs the machine learning inference on that data, while the other ranks wait. Finally, the result data is distributed across the whole workgroup again using the `MPI_Scatterv` operation, where each rank receives the matching output data that was inferred using its input data. The process is illustrated in Figure 2.4.

The specifics of this process and the implementation using several ML frameworks is discussed in more detail in Chapter 3.

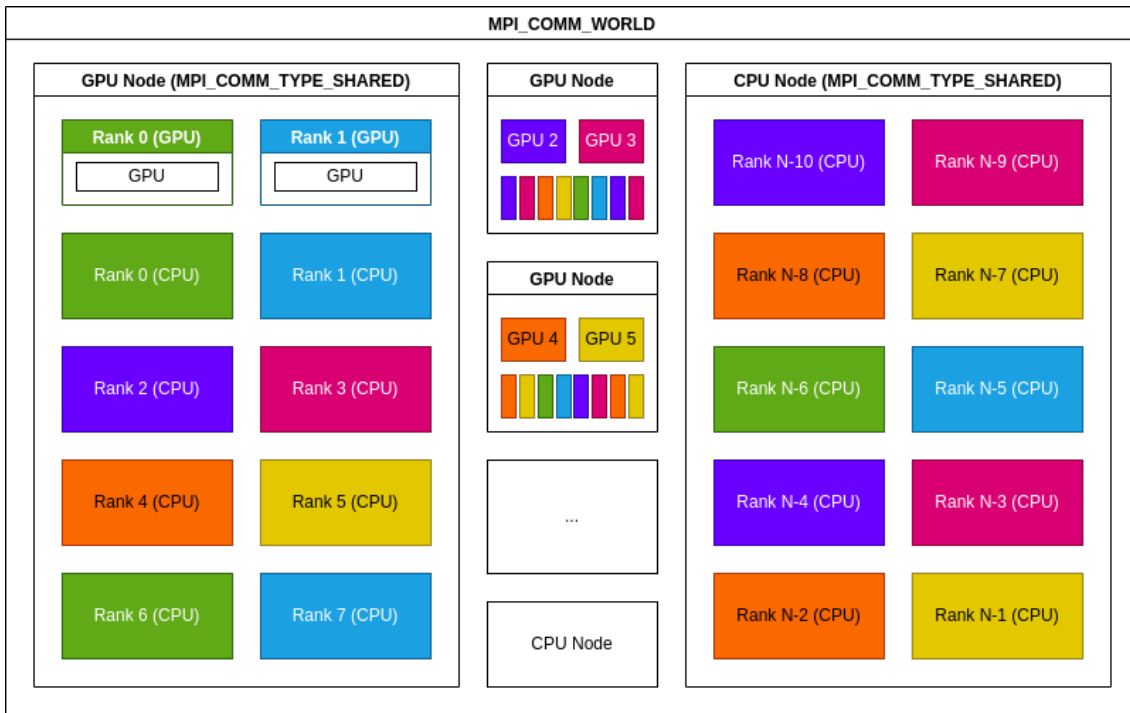


Figure 2.3: MPI ranks are colored based on their assigned workgroup, using  $G = 6$  colors as example.

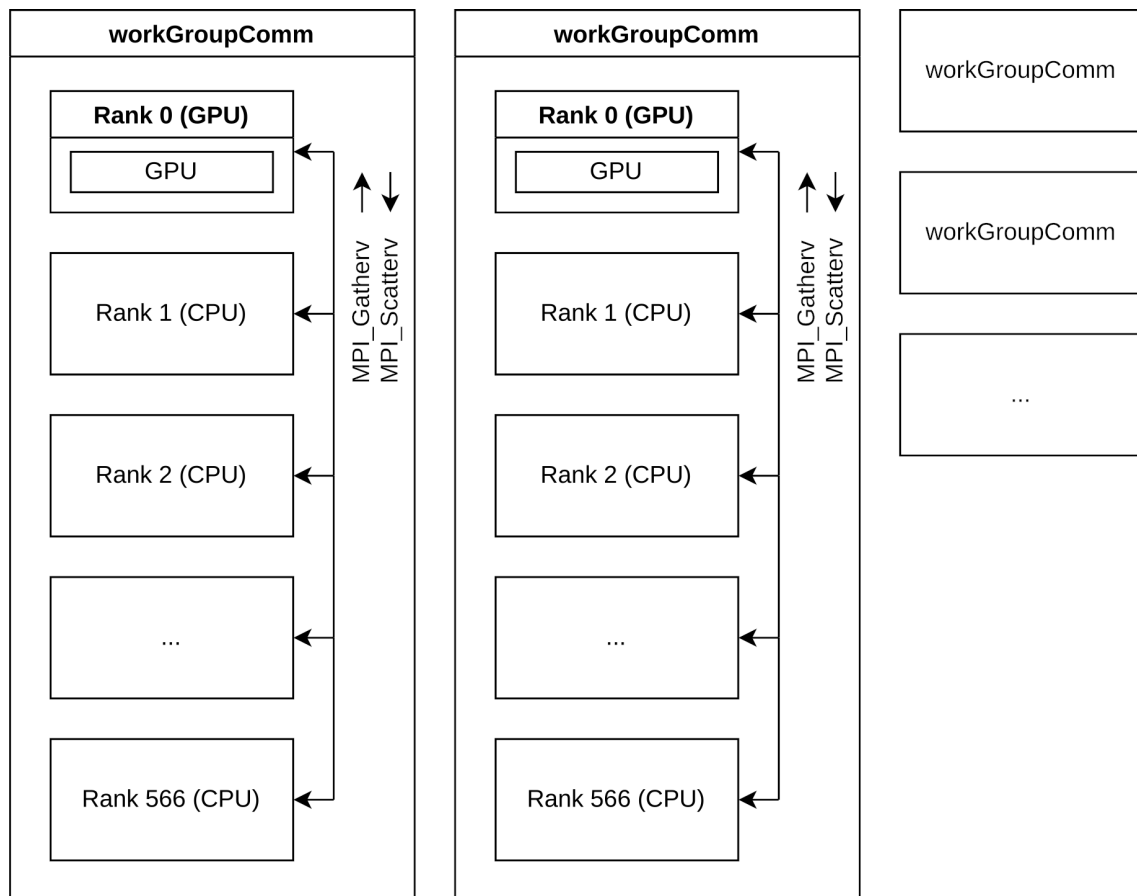


Figure 2.4: The input data of each workgroup is gathered on the GPU master rank, which performs inference and scatters the data back across its workgroup.

# 3 Frameworks Coupling

## 3.1 Design of the Coupling Interface

To implement the couplings of OpenFOAM to the several examined machine learning (ML) frameworks, we reused and expanded upon the class interface from the original PyTorch coupling given as a starting point. Figure 3.1 depicts the general concept. Upon construction, the `machinelearningInference` objects receive a pointer to the OpenFOAM `objectRegistry`, which contains the data of the application. The `machinelearningInference` interface itself most importantly offers the public function `forward()` which replaces the combustion simulation step that was originally implemented using a high dimensional lookup table (LUT).

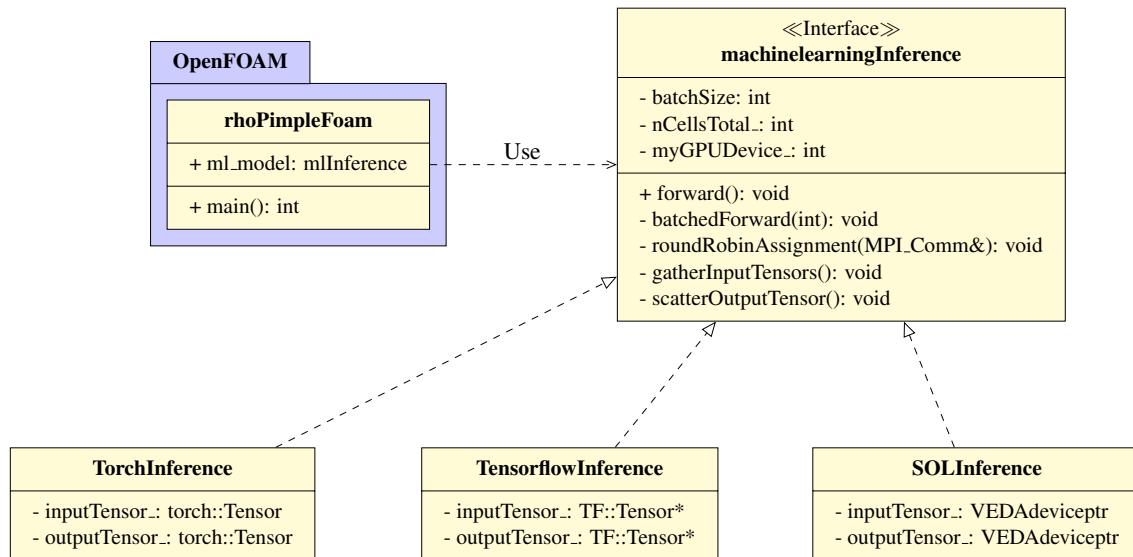


Figure 3.1: The conceptual interface for the coupling

All couplings are extremely similar in their control flow. The constructor handles all setup and initialization, including memory allocation for tensors (multidimensional matrices/arrays), assignment of MPI ranks into workgroups (see Section 2.2) and assignment of accelerator devices to MPI ranks (`roundRobinAssignment`), initialization of machine learning frameworks, and loading the machine learning model. The `forward()` step is outsourced to 3 helper functions representing the 3 required sub-steps.

1. `gatherInputTensors()`: extracts the tensor data from OpenFOAM to a C array like structure (Listing 3.1) which is then communicated to the GPU

### 3 Frameworks Coupling

master rank over MPI with `MPI_Gatherv`. In case it is impossible for MPI to use the target tensor as receiving buffer directly, the receiving rank repacks the data into the fitting tensor type for the respective framework.

2. `batchedForward()`: all examined machine learning frameworks support processing of batched data, i.e., process multiple pieces of data at once that would otherwise require multiple inference runs of the model, see Section 3.2.3.
3. `scatterOutputTensors()`: the reverse action of `gatherInputTensors()`, distributes the inferred data from the GPU master rank over the whole workgroup using `MPI_Scatterv`.

Some parts can use the exact same data types for variables, most importantly:

- `batchSize`: the amount of data that can be processed by the machine learning framework at once
- `nCellsTotal_`: the total amount of data in this workgroup which is derived from the detail grade of the OpenFOAM domain partitioning
- `myGPUDevice_`: the numeric ID of the GPU or other accelerator device assigned to this MPI rank.

In other parts the specific frameworks require the use of specialized data types while remaining conceptually equal or similar, for example all variants need variables to hold the `inputTensor_` and `outputTensor_` data, but each framework utilizes its own class to implement a tensor.

In our actual implementation of the inference classes, we do not make use of any inheritance to avoid conflicts between the partially shared dependencies and the different required versions thereof for each machine learning framework, however the concept does apply.

## 3.2 General Consideration of Machine Learning Framework Interfaces

Despite the differences in their implementation detail, the examined machine learning frameworks conceptually work in the same way for our purpose. We identified some pitfalls during this work, which can present difficulties when comparing frameworks not just for their performance, but also the correctness of the results. Some of these pitfalls might be apparent to machine learning experts, but we will present them here since they are nevertheless relevant for this work.

### 3.2.1 Converting Models

In the optimal case, we want to compare the exact same machine learning model in different frameworks. This is not a trivial task, because the two base frameworks we want to examine, PyTorch and TensorFlow, use their own file formats to store their models. Neither of both is able to import the other's model, and while there is an initiative for a common file format ONNX [5] to which at this point both frameworks are able to export their own models, neither is able to import it natively and officially. We investigated a third party tool [21] that aims to provide this functionality, but were unsuccessful to reproduce the same exact results of the original model after converting it with this tool.

Since our model is conceptually simple and uses only basic features supported by both frameworks, our next step was to try and generate the same model manually. As explained in Section 2.2, our model consists of four fully connected layers, where each neuron has both a weight and a bias associated with each of its connected neurons. Research has shown that neural networks perform better when these weights and biases are initialized with values drawn from a random distribution before training the model. We discovered that PyTorch and TensorFlow default to using different distributions for this task: PyTorch uses a Kaiming uniform distribution [17], also known as He uniform distribution, for both its weights and biases [7]. TensorFlow uses a Glorot uniform distribution<sup>1</sup> [16], also known as Xavier uniform distribution, for the weights, and zeroes (0) for the biases [8]. As a simple test, we initialized all weights and biases with fixed numbers, however we were still unable to reproduce the same results as shown in Table 3.1.

| Framework      | input   |         | output                       |                              |
|----------------|---------|---------|------------------------------|------------------------------|
|                | field 0 | field 1 | field 0                      | field 1                      |
| PyTorch        | 0       | 0       | 0                            | 0                            |
| TensorFlow     | 0       | 0       | $2.339\ 28 \times 10^{-310}$ | $3.289\ 64 \times 10^{-316}$ |
| Torch-TensorRT | 0       | 0       | 0                            | 0                            |
| SOL            | 0       | 0       | 0                            | 0                            |

Table 3.1: TensorFlow is the only framework that consistently produces random values diverging from the expected 0 result. We have seen values as small as  $10^{-316}$  that could be attributed to floating point errors, but it produces values of magnitudes such as  $10^5$  just as often.

### 3.2.2 Tensor Format

Both PyTorch and TensorFlow allow for batched processing of our input data. As a framework user, that means we are able to submit multiple input data to our model

<sup>1</sup>[https://www.tensorflow.org/versions/r2.6/api\\_docs/python/tf/keras/initializers/GlorotUniform](https://www.tensorflow.org/versions/r2.6/api_docs/python/tf/keras/initializers/GlorotUniform)

### 3 Frameworks Coupling

simultaneously as a single data structure and receive the results likewise, instead of sending all data one by one for example in a loop. From a performance perspective this is also desirable, because there are less context switches, the data submission is pipelined, and the algorithm can profit from possible optimizations on the hardware such as eliminating the need to load data common to operations concerning all input data (weights and biases) and reusing them instead.

Tensors in machine learning commonly have two or four dimensions, where one dimension  $N$  represents the batch size. In a common application of processing image data, three additional dimensions are used to represent vertical  $H$  and horizontal  $W$  pixel positions, and the color channel  $C$ . In our test case the different cells correspond to the batch dimension and we have multiple fields per cell which we represent with another dimension. PyTorch defaults to  $NCHW$  format<sup>2</sup>, while TensorFlow defaults to  $NHWC$ <sup>3</sup>. Both formats have advantages, most importantly the resulting memory locality can have strong implications for the performance depending on the used neural network and hardware architecture [13]. Since we only use the batch dimension  $N$  and one other dimension, we use a simple 2 dimensional tensor and assume no performance implications.

OpenFOAM provides access to the relevant data in a field-first batch-last order, which we need to convert to the batch-first field-last ordered tensors that our frameworks expect (Listing 3.1).

---

```
1 for(unsigned int j = 0; j < nFields_; j++)
2 {
3     const volScalarField &field = *inputFields_[j];
4     for(int k = 0; k < nCells_; k++)
5     {
6         inputValues_[j+k*nFields_] = field[k];
7     }
8 }
```

---

Listing 3.1: Converting the OpenFOAM field-first data format to batch-first

C arrays and C++ `std::vectors` use the row-major order to store data as illustrated by Figure 3.2a: The arrow represents the order of memory addresses of the cells, and the labeling of the cells represents the row (first number) and column (second number) of a two dimensional array. Once our data is stored in a manner that batch-first matches the row-major order, we can use the varying vector length MPI calls `MPI_Gatherv` and `MPI_Scatterv` (Listing 3.2) to communicate our partitioned-by-batch data (Figures 3.2a, 3.2b) across multiple ranks and expect it to get combined or split correctly (Figure 3.2c). In comparison to their non-varying length versions, `MPI_Gatherv` and `MPI_Scatterv` take modified `recvcounts[]/sendcounts[]` arguments as integer arrays instead of scalar integers to allow a different count per rank in the communicator. Additionally there is the new argument `displs[]` which defines where to place the incoming data per rank in form of an offset from `recvbuf`.

---

<sup>2</sup>[https://pytorch.org/docs/1.10/tensor\\_attributes.html#torch-memory-format](https://pytorch.org/docs/1.10/tensor_attributes.html#torch-memory-format)

<sup>3</sup><https://www.tensorflow.org/guide/tensor>

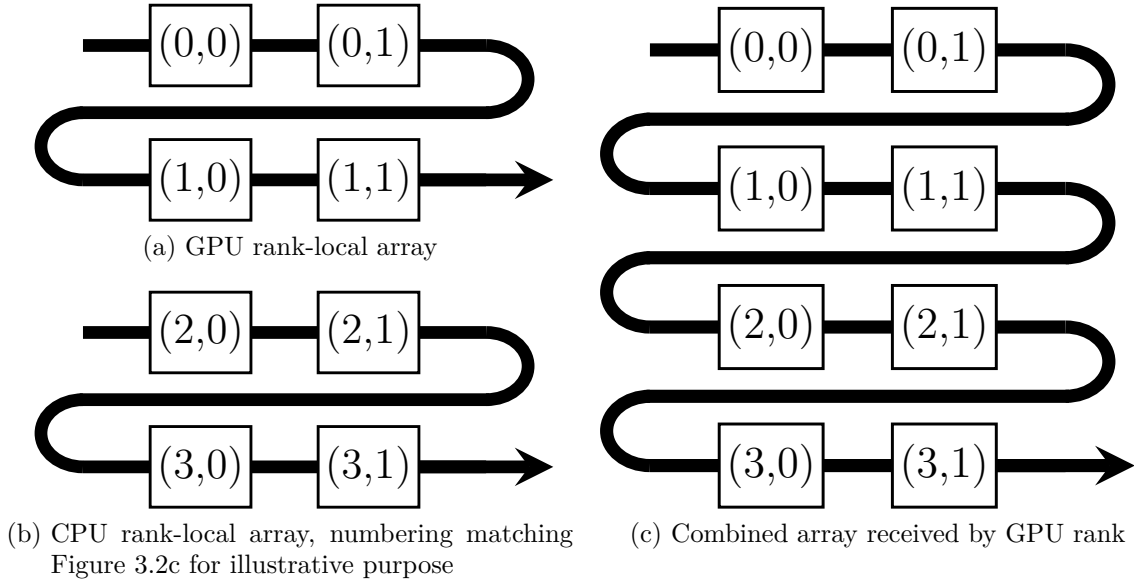


Figure 3.2: Combining row-major ordered arrays in C++ using `MPI_Gatherv`. The array cells (rectangles) are ordered and labeled as a 2D array. The arrows represent the incrementing memory addresses at which C++ stores the data.

---

```

1 int MPI_Gatherv(const void *sendbuf, int sendcount,
2               MPI_Datatype sendtype, void *recvbuf,
3               const int recvcnts[], const int displs[],
4               MPI_Datatype recvtype, int root, MPI_Comm comm)
5
6 int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
7                const int displs[], MPI_Datatype sendtype,
8                void *recvbuf, int recvcnt,
9                MPI_Datatype recvtype, int root, MPI_Comm comm)

```

---

Listing 3.2: Syntax of MPI vector operations with varying length [23]

### 3.2.3 Batching and Batch Size Estimation

A common difficulty with all frameworks except Triton is a missing automatism for balancing the amount of data submitted to the device at a time with the device’s inherent hardware constraints. Hardware resources of accelerator devices are limited, for our use case mainly by the amount of operations it can perform per time on one hand and memory bandwidth and size on the other hand. Due to the memory size constraint, it is usually not possible to feed *all* the data into the machine learning model at once, because each layer in the model temporarily requires an amount of space to store the intermediate results of the layer, compare Section 2.2. Hence we are required to submit the data to the device in batches that fit its available memory

even in the intermediate steps.

The Triton inference server has a dynamic batching capability that can be tweaked in multiple aspects, including maximal latency and minimal workload efficiency. We cannot find a similar support in the other frameworks, and neither can we stably express the estimation in an algorithm, hence we stick to a trial-and-error approach.

### 3.3 PyTorch

PyTorch [26] is a machine learning framework developed by Meta (formerly Facebook) since 2016. It offers a front end primarily in the Python programming language as its name implies, but also a C++ API that intends to follow the Python interface closely.

We modified the preexisting model (“Module” in PyTorch terms) creation Python code (Listing 3.3) in order to match it and the TensorFlow (Section 3.4) model as closely as possible due to the problem explained in Section 3.2.1 while retaining the MLP structure explained in Section 2.2. The sizes of our layers are defined by the respective constructor parameters (lines 6–7, 13). The model is constructed in lines 20–32 by first creating a fully connected layer (called “Linear”). In particular, we initialize the weights uniformly based on layer size (lines 23–24), and the bias to 0 (line 25). The activation function is added after each layer (line 27), except the last layer which is the output layer (lines 29–32). Running the model is defined by a function called `forward()` which simply applies layer after layer (lines 34–38) to some input. This is repeated for all layers, except no activation function is added after the output layer.

PyTorch’s C++ interface makes handling the framework very comfortable (Listing 3.4). Loading the model we previously exported from Python is as easy as calling the load command on the file name (line 1). We can then instruct PyTorch to load the model onto the CUDA-enabled GPU assigned to this rank (line 2). We also allocate several tensor objects (lines 3–5) that will be reused to hold the data in each `forward()` iteration.

---

```

1 torchModel_ = torch::jit::load(modelFileName_);
2 torchModel_.to(torch::Device(torch::kCUDA, myGPUDevice_));
3 torch::TensorOptions options(torch::kFloat64);
4 inputTensor_ = torch::ones({nCells_, nFields}, options);
5 outputTensor_ = torch::ones({nCells_, nFields}, options);

```

---

Listing 3.4: Initialization of the PyTorch coupling

We can communicate our tensor objects over MPI directly, because the tensors store their data in a C array-like memory format, to which we can simply get a pointer (Listing 3.5). Scattering the data works analogously to gathering.

The tensor objects offer a method `slice()` natively that enables us to extract chunks of the data in order to form our batches (Listing 3.6 line 3). We then copy the data onto the respective GPU similar to how we loaded the model onto the GPU before (line 4). Running the model is as simple as calling the model’s `forward()`

---

```

1 import torch
2 import torch.nn as nn
3 from typing import List
4
5 class FlexMLP(torch.nn.Module):
6     def __init__(self, n_inp: int, n_out: int,
7                 n_hidden_neurons: List[int] = [32, 32],
8                 activation_fn: torch.nn.Module = nn.ReLU):
9         super().__init__()
10
11         self.n_inp = n_inp
12         self.n_out = n_out
13         self.n_neurons = [n_inp] + n_hidden_neurons + [n_out]
14         self.hidden_layers = len(n_hidden_neurons)
15         self.layers = torch.nn.ModuleList()
16         self.activation = activation_fn
17
18         # construct the network
19         # -2 because we add the last layer manually to avoid an
20         # activation there
21         for i in range(len(self.n_neurons)-2):
22             _layer = torch.nn.Linear(self.n_neurons[i],
23                                     self.n_neurons[i+1])
24             torch.nn.init.constant_(_layer.weight,
25                                   1.0/n_hidden_neurons[0])
26             torch.nn.init.zeros_(_layer.bias)
27             self.layers.append(_layer)
28             self.layers.append(self.activation())
29
30             _layer = nn.Linear(self.n_neurons[-2], self.n_neurons[-1])
31             torch.nn.init.constant_(_layer.weight, 1.0/n_hidden_neurons[0])
32             torch.nn.init.zeros_(_layer.bias)
33             self.layers.append(_layer)
34
35         def forward(self, x):
36             # loop through all layer but last
37             for layer in self.layers:
38                 x = layer(x)
39             return x
40
41 m = FlexMLP(n_inp, n_out, n_neurons).double().requires_grad_(False)
42 s = torch.jit.script(m)
43 s.save(filename)

```

---

Listing 3.3: Creating our neural network model in PyTorch using Python

---

```

1 MPI_Gatherv(inputTensor_.data_ptr(), nCells_*nFields, MPI_DOUBLE,
2             inputTensorMaster_.data_ptr(), nCellDataRecv_,
3             nCellDataRecvOffset_, MPI_DOUBLE, 0, workGroupComm_);

```

---

Listing 3.5: MPI in the PyTorch coupling

method on the input data (line 6). We use the `slice()` method again, this time in reverse, to form the complete tensor out of the result batches (line 7).

---

```
1 for(int i = 0; i < nSlices; i++)
2 {
3     inputTensorSlice_ = inputTensorMaster_.slice(0, batchsize*i,
4         batchsize*(i+1));
5     inputTensorGPU_ =
6         inputTensorSlice_.to(torch::Device(torch::kCUDA, myGPUDevice_));
7     std::vector<torch::jit::IValue> inputTensors = {inputTensorGPU_};
8     outputTensorGPU_ = torchModel_.forward(inputTensors).toTensor();
9     outputTensorMaster_.slice(0, batchsize*i, batchsize*(i+1)) =
10         outputTensorGPU_.to(torch::kCPU);
11 }
```

---

Listing 3.6: Batch processing in the PyTorch coupling

## 3.4 TensorFlow

TensorFlow [10] is a popular machine learning framework developed by Google since 2015. It primarily offers a Python interface, but the installation page on the official website<sup>4</sup> also offers language bindings for the Java, C, and Go programming languages. Upon further investigation we found hints at a C++ API as well, but could not find any concrete information on the framework’s official channels nor elsewhere. Further, the project states the C API is documented only through the C header file `c_api.h`<sup>5</sup> itself and intended to build language bindings for other languages and not for end users. Since we are limited in our choice of programming language by the need to integrate with the C++ OpenFOAM code, we chose to use the C API regardless.

As with PyTorch, we create our TensorFlow model in Python as shown in Listing 3.7 for the ease of use. Instead of defining our own model class derived from a preexisting class, we create an instance of `Sequential` (line 9) which represents a model made up of a simple sequence of layers, and then add our layers based on the same arguments (line 7) that we used to create our PyTorch model. The fully connected layers are called “Dense” (lines 12, 18) in TensorFlow, and similar to PyTorch we override the framework’s default initializers for weights (“kernels” in TensorFlow, lines 15, 19) and biases (lines 16, 20) with constants to avoid the models diverging as explained in Section 3.2.1.

Compared to the PyTorch version, initialization of the TensorFlow coupling (Listing 3.10) is much more complicated. We immediately face the issue that TensorFlow tries to allocate all visible GPUs in a blocking, exclusive use manner, leading to the application crashing when we try to initialize two GPU master ranks per GPU-node.

---

<sup>4</sup><https://www.tensorflow.org/install>

<sup>5</sup>[https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c\\_api.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)

---

```

1 import tensorflow as tf
2 from typing import List
3 k = tf.keras
4 ini = k.initializers
5 l = k.layers
6
7 def FlexMLP(n_inp: int, n_out: int, n_hidden_neurons: List[int] =
      [32, 32], activation_fn = 'relu', dtype='float64', name=None):
8     k_ini = ini.Constant(1.0 / n_hidden_neurons[0])
9     m = k.Sequential()
10    m.add(k.Input(shape=(n_inp,), dtype=dtype))
11    for h in n_hidden_neurons:
12        m.add(l.Dense(h,
13                    activation = activation_fn,
14                    dtype = dtype,
15                    kernel_initializer = k_ini,
16                    bias_initializer = ini.Zeros()))
17
18    m.add(l.Dense(n_out, dtype = dtype,
19                kernel_initializer = k_ini,
20                bias_initializer = ini.Zeros()))
21    return m
22
23 m = FlexMLP(n_inp, n_out, n_neurons)
24 m.save(filename)

```

---

Listing 3.7: Creating our neural network model in TensorFlow using Python

There are multiple ways to manage which GPUs are visible to a GPU-enabled process, e.g., through CUDA environment variables. However most of these methods must be applied before launching the application itself. This is not feasible for us, we need to configure devices at runtime in order to accommodate the multitude of configurations we need to benchmark. We implement our GPU configuration with help from TensorFlow's Python API, which offers the ability to configure devices through a human-readable interface as shown in Listing 3.8. Once configured, we can export the configuration in form of a serialized string, which the C API can understand (`config[]`, Listing 3.10 line 4). In this otherwise default configuration, the last byte switches between values of hexadecimal `0x30` and `0x31` depending on whether we specify GPU index 0 or 1.

---

```

1 import tensorflow as tf
2 gpu_options = tf.compat.v1.GPUOptions(visible_device_list='0')
3 config = tf.compat.v1.ConfigProto(gpu_options=gpu_options)
4 serialized = config.SerializeToString()
5 print(list(map(hex, serialized)))

```

---

Listing 3.8: Serializing our configuration using Python

We also need to configure information about how to run the model (`tags`) and the names of the inputs and outputs (`serving_default_input_1` and

### 3 Frameworks Coupling

`StatefulPartitionedCall`) needed later on to run the model. These values are generated by TensorFlow and can differ from model to model, even when recreating the same model with a different amount of neurons or similar tweaks. We can obtain the values by use of the Python script `saved_model_cli` included in the TensorFlow Python package as shown in Listing 3.9.

---

```
1 saved_model_cli show --dir Script.tf --tag_set serve \  
2 --signature_def serving_default  
3  
4 # The given SavedModel SignatureDef contains the following  
   input(s):  
5 #   inputs['input_1'] tensor_info:  
6 #     dtype: DT_DOUBLE  
7 #     shape: (-1, 2)  
8 #     name: serving_default_input_1:0  
9 # The given SavedModel SignatureDef contains the following  
   output(s):  
10 #   outputs['dense_2'] tensor_info:  
11 #     dtype: DT_DOUBLE  
12 #     shape: (-1, 2)  
13 #     name: StatefulPartitionedCall:0  
14 # Method name is: tensorflow/serving/predict
```

---

Listing 3.9: Obtaining hardcoded values using `saved_model_cli`

Finally we initialize the coupling (Listing 3.10). We load the model by providing all the arguments we obtained and the model directory name (line 11). We also create the input and output objects for the inference step (lines 14–20) and allocate tensor objects to store the data (lines 22–23).

Similar to the PyTorch tensor objects, we can access the contents of the TensorFlow tensor data by pointer using the `TF_TensorData` function, however we ran into problems when using this pointer directly for MPI communication hence we copy the data from a temporary array used with MPI to the tensor (Listing 3.11 lines 3–10). Scatter is again implemented by reversing the gather process.

---

```
1 MPI_Gatherv(&inputValues_[0], nCells_ * nFields_, MPI_DOUBLE,  
   &inputValuesMaster_[0], nCellDataRecv_, nCellDataRecvOffset_,  
   MPI_DOUBLE, 0, workGroupComm_);  
2 if (isGPUMaster_) {  
3   double* inputTensorMasterData =  
   (double*)TF_TensorData(inputTensorMaster_);  
4   for(unsigned int j = 0; j < nFields_; j++)  
5   {  
6     for(int k = 0; k < nCellsTotal_; k++)  
7     {  
8       inputTensorMasterData[j+k*nFields_] =  
       inputValuesMaster_[j+k*nFields_];  
9     }  
10  }  
11 }
```

---

Listing 3.11: MPI in the TensorFlow coupling

---

```

1 graph_ = TF_NewGraph();
2 opts_ = TF_NewSessionOptions();
3 const size_t len = 5;
4 uint8_t config[len] = {0x32, 0x3, 0x2a, 0x1, 0x30};
5 // for a single GPU, the last value seems to indicate the GPU,
   counting up from 0x30
6 if (nodeRank > 0) config[len - 1] = 0x30 + nodeRank;
7 TF_SetConfig(opts_, (void*)config, len, status_);
8 const char* saved_model_dir = "Script.tf";
9 const char* tags = "serve";
10 int ntags = 1;
11 session_ = TF_LoadSessionFromSavedModel(opts_, NULL,
   saved_model_dir, &tags, ntags, graph_, NULL, status_);
12 if (TF_GetCode(status_)) printf("Tensorflow Load Session Error:
   %s\n", TF_Message(status_));
13
14 ninputs_ = noutputs_ = 1;
15 inputs_ = (TF_Output *)malloc(sizeof(TF_Output) * ninputs_);
16 outputs_ = (TF_Output *)malloc(sizeof(TF_Output) * noutputs_);
17 TF_Output i0 = {TF_GraphOperationByName(graph_,
   "serving_default_input_1"), 0};
18 inputs_[0] = i0;
19 TF_Output o0 = {TF_GraphOperationByName(graph_,
   "StatefulPartitionedCall"), 0};
20 outputs_[0] = o0;
21
22 inputTensor_ = TF_AllocateTensor(opt_type, dims, ndims,
   sizeof(double) * nCells_ * nFields_);
23 outputTensor_ = TF_AllocateTensor(opt_type, dims, ndims,
   sizeof(double) * nCells_ * nFields_);

```

---

Listing 3.10: Initialization of the TensorFlow coupling

Contrary to the PyTorch API, TensorFlow’s tensors do not support slicing on their own. Instead we implement the slicing manually using pointer arithmetic, which is possible since we can copy the backing data between tensor objects like raw memory (Listing 3.12). A tensor slice of the size of a batch gets copied to a new batch-sized tensor (line 10). Then the model performs inference (`TF_SessionRun`, line 13) and the batch result gets copied into the full-sized output tensor (line 16). The pointers are adjusted to the next batch start after each iteration (lines 11, 17). Because it is unlikely the full amount of data is divisible by the batch size without remainder, the first iteration uses smaller batch size and tensors to accommodate just the remainder (lines 19–28). Notably, we need not manage the target device for the tensors here in contrast to the PyTorch model execution (Listing 3.6).

Both PyTorch and TensorFlow support models with their layers interconnected in a Y or X shape, i.e., more than a single in-/output layer. This is the reason for the APIs accepting the tensors within a list-like data structure such as an array or `std::vector`: Note the `ninputs_` and `noutputs_` arguments in line 13 of List-

---

```

1 double* inputTensorMasterData =
    (double*)TF_TensorData(inputTensorMaster_);
2 double* outputTensorMasterData =
    (double*)TF_TensorData(outputTensorMaster_);
3 TF_Tensor* inputTensorGPU = TF_AllocateTensor(TF_DOUBLE,
    dimsMaster, ndims, sizeof(double) * nCellsRemaining * nFields_);
4 TF_Tensor* outputTensorGPU = TF_AllocateTensor(TF_DOUBLE,
    dimsMaster, ndims, sizeof(double) * nCellsRemaining * nFields_);
5 double* inputTensorGPUData =
    (double*)TF_TensorData(inputTensorGPU);
6 double* outputTensorGPUData =
    (double*)TF_TensorData(outputTensorGPU);
7
8 for(int i = 0; i < nSlices; i++)
9 {
10     std::memcpy(inputTensorGPUData, inputTensorMasterData,
        sizeof(double) * nCellsRemaining * nFields_);
11     inputTensorMasterData += nCellsRemaining * nFields_;
12
13     TF_SessionRun(session_, NULL, inputs_, &inputTensorGPU,
        ninputs_, outputs_, &outputTensorGPU, noutputs_, NULL, 0, NULL,
        status_);
14     if (TF_GetCode(status_)) printf("Tensorflow Run Session Error:
        %s\n", TF_Message(status_));
15
16     std::memcpy(outputTensorMasterData, outputTensorGPUData,
        sizeof(double) * nCellsRemaining * nFields_);
17     outputTensorMasterData += nCellsRemaining * nFields_;
18
19     if (i == 0) {
20         TF_DeleteTensor(inputTensorGPU);
21         TF_DeleteTensor(outputTensorGPU);
22         nCellsRemaining = batchsize;
23         dimsMaster[0] = nCellsRemaining;
24         inputTensorGPU = TF_AllocateTensor(TF_DOUBLE, dimsMaster,
            ndims, sizeof(double) * nCellsRemaining * nFields_);
25         outputTensorGPU = TF_AllocateTensor(TF_DOUBLE, dimsMaster,
            ndims, sizeof(double) * nCellsRemaining * nFields_);
26         inputTensorGPUData = (double*)TF_TensorData(inputTensorGPU);
27         outputTensorGPUData = (double*)TF_TensorData(outputTensorGPU);
28     }
29 }
30 TF_DeleteTensor(inputTensorGPU);
31 TF_DeleteTensor(outputTensorGPU);

```

---

Listing 3.12: Batch processing the TensorFlow coupling

ing 3.12 and the `std::vector` used for PyTorch in line 5 of Listing 3.6. These lists should not be confused with the structure to define one’s batches, which are handled via the first dimension of the tensor itself (Section 3.2.2).

## 3.5 Nvidia Triton

According to its official website, “NVIDIA Triton™ Inference Server is an open-source inference serving software that helps standardize model deployment and execution and delivers fast and scalable AI in production.”[4] It has support for multiple machine learning frameworks built in, including the PyTorch, TensorFlow, and TensorRT frameworks we want to examine. Nvidia offers the Triton client library API for Python, Java, and C++ programming languages and we found the C++ interface well documented in the 3 relevant header files.

Triton is not a machine learning framework, but rather an “Inference Server” that implements advanced couplings for many frameworks [4] and exposes access to them over web APIs. Figure 3.3 shows a possible way of integrating Triton into an application. Multiple “thin” clients that are unsuited to run machine learning inference locally due to constraints including processing power or battery life instead connect to a Triton server over a text or binary (gRPC) HTTP API for offloading. A request includes information about what model to use and the input data for the inference. The Triton server schedules the request on its available hardware, including advanced operations such as batching the data and spreading it across multiple GPU devices depending on its configuration, and executes it using the appropriate framework coupling. Finally the result is sent as a response on the same API where the request was made.

This approach differs greatly from our usual couplings in the software architecture sense. Thanks to the use of HTTP APIs, it becomes possible to introduce a global HTTP load balancer, which can evenly spread the workload across workgroups, or even replace part of the MPI communication (see Chapter 8). The client library offers many configuration options that allow one to tweak the communication and model inference in much detail, as well as additional tools to benchmark certain configurations. At the same time it presents a challenge of how to run the servers within the requirement of using the RWTH CLAIX batch system and addressing over the IP-TCP-HTTP stack. Triton is also particularly interesting to us, because we see an opportunity to offer such a service for efficient machine learning offloading with a simple API in the HPC environment and want to understand performance of such application models.

In the end, we were unable to test our implementation due to troubles with the Triton client libraries’ build system: Nvidia provides the library precompiled, however these are intended for a supported Ubuntu version 20.04 LTS while the RWTH CLAIX cluster runs on CentOS 7.9 which proved incompatible. Building the library

---

<sup>6</sup><https://developer.nvidia.com/sites/default/files/akamai/ai-for-enterprise-print-update-to-triton-diagram-1339418-final-r3.jpg>

### 3 Frameworks Coupling

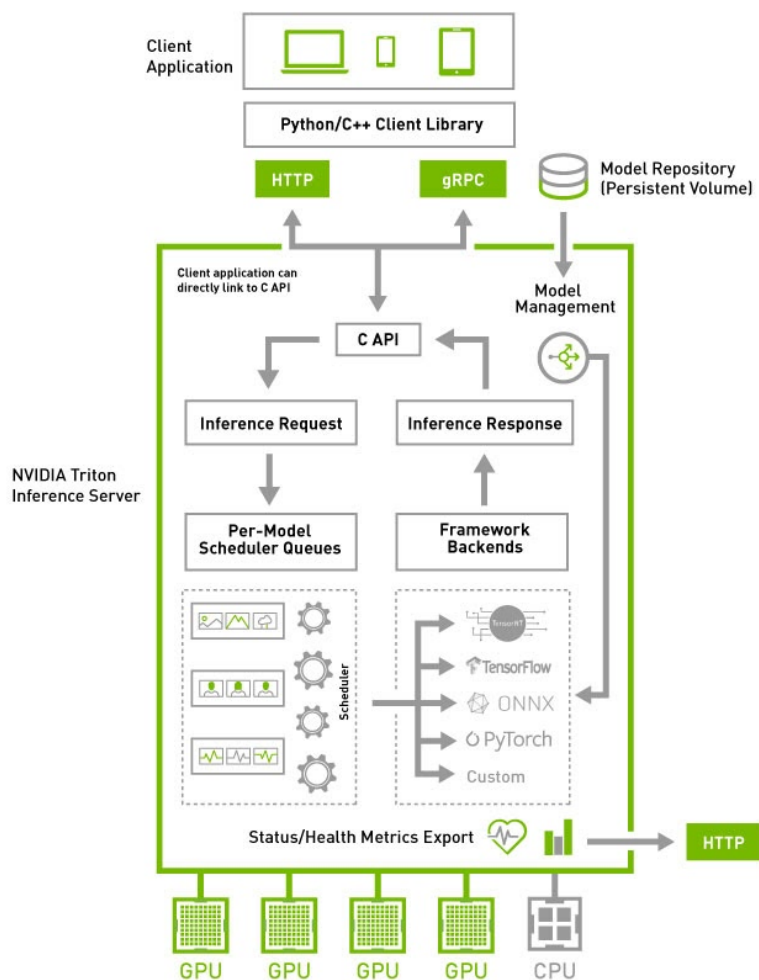


Figure 3.3: Nvidia Triton application flow schema, source: Nvidia<sup>6</sup>

on our own, we uncovered two defects in the build system, one of which concerning priorities of libraries used<sup>7</sup> was fixed soon after our report. The other problem involves a dependency missing from the build system<sup>8</sup>, however due to the HPC environment we were unable to provide the dependency system-wide as the build system (wrongly) expects and were unable to find the root cause within our time constraints. Despite our inability to test this implementation, we still explain the concept of our implementation of the “as-close-as-possible” application model (Figure 3.4) in the remainder of this section which follows the provided `simple_grpc_infer_client.cc`<sup>9</sup>

<sup>7</sup><https://github.com/triton-inference-server/server/issues/3948>

<sup>8</sup><https://github.com/triton-inference-server/server/issues/3922>

<sup>9</sup>[https://github.com/triton-inference-server/client/blob/27522b01698b71b3fdb7ebe90ffe88e962b82789/src/c++/examples/simple\\_grpc\\_infer\\_client.cc](https://github.com/triton-inference-server/client/blob/27522b01698b71b3fdb7ebe90ffe88e962b82789/src/c++/examples/simple_grpc_infer_client.cc)

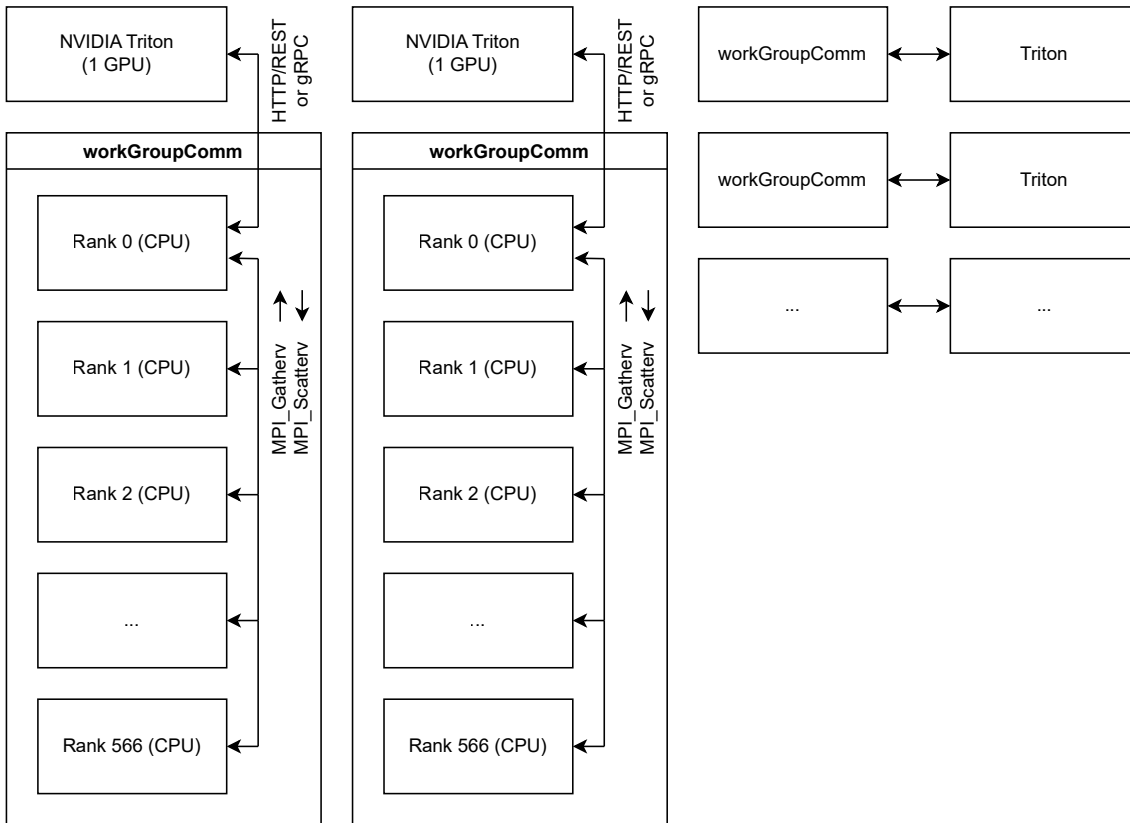


Figure 3.4: Triton communication model “as-close-as-possible” following the other frameworks’ communication model (Figure 2.4)

example.

Initialization (Listing 3.13) of the coupling is closely related to our other couplings. The first step as usual is to determine whether the rank is a “GPU master”, i.e., is supposed to a Triton instance. Contrary to the other couplings our rank should not allocate a GPU to itself, and further, since the Triton server has already claimed exclusive access to the GPU, we are unable to probe for available GPUs as an indicator. Instead, if we are eligible by MPI rank within the local node (line 4), we try to connect to a Triton and either are successful, or we are not, in which case we assume there wasn’t an error but we are on a node without GPU/Triton (lines 5–9). If we connected successfully, we then load the appropriate model (line 12), which was provided alongside some required model metadata to the Triton in advance via its configuration. We also create the input and output tensors, which get associated with the input’s and output’s names within the model itself known beforehand (lines 16–23).

The client API’s `InferInput` objects are different from other frameworks’ tensor objects in that they only receive data by passing them a pointer instead of owning a memory region with the data themselves. The `InferRequestedOutput` objects only indicate the requested data format for making a request to the server and do

---

```

1 int gpuMaster = 0;
2 std::string tritonUrl = "localhost:8000";
3 bool verbose = true;
4 if (nodeRank < 2) {
5     FAIL_IF_ERR(
6         tc::InferenceServerGrpcClient::Create(&tritonClient_,
7         tritonUrl, verbose),
8         "unable to create grpc client");
9     tritonClient_ ->IsServerLive((bool*)&gpuMaster);
10 }
11 if (gpuMaster) {
12     FAIL_IF_ERR(
13         tritonClient_ ->LoadModel(modelName),
14         "unable to load model " + modelName);
15 }
16 tc::InferInput* inputTensorMaster_;
17 tc::InferRequestedOutput* outputTensorMaster_;
18 FAIL_IF_ERR(
19     tc::InferInput::Create(&inputTensorMaster_, inputName, shape,
20     tritonDatatype),
21     "unable to get " + inputName);
22 FAIL_IF_ERR(
23     tc::InferRequestedOutput::Create(&outputTensorMaster_,
24     outputName),
25     "unable to get " + outputName);

```

---

Listing 3.13: Initialization of the Triton coupling

not hold any tensor data. We use intermediate helper `std::vector<double>`s for the MPI communication and data storage and then set the pointer to that data (Listing 3.14, lines 6–8).

---

```

1 MPI_Gatherv(&inputValues_[0], nCells_ * nFields, MPI_DOUBLE,
2     &inputValuesMaster_[0], nCellDataRecv_, nCellDataRecvOffset_,
3     MPI_DOUBLE, 0, workGroupComm_);
4
5 if (isGPUMaster_) {
6     inputTensorMaster_ptr_ ->Reset();
7     FAIL_IF_ERR(
8         inputTensorMaster_ptr_ ->AppendRaw(
9         reinterpret_cast<uint8_t*>(&inputValuesMaster_[0]),
10        inputValuesMaster_.size() * sizeof(cppDatatype)),
11        "unable to set data for" + inputName);
12 }

```

---

Listing 3.14: MPI\_Gatherv in the Triton coupling

Sending the inference request is a single call, because the Triton server’s dynamic batching capability takes care of batching for us. We provide pointers to the `results_` object, inputs, output definitions, `options_` containing the model’s

name, default `http_headers_`, and disable gRPC compression as we expect it to only cause overhead on our local socket (Listing 3.15).

---

```

1 std::vector<tc::InferInput*> inputs =
    {inputTensorMaster_ptr_.get()};
2 std::vector<const tc::InferRequestedOutput*> outputs =
    {outputTensorMaster_ptr_.get()};
3 FAIL_IF_ERR(
4     tritonClient_ ->Infer(
5         &results_, options_, inputs, outputs, http_headers_,
6         grpc_compression_algorithm::GRPC_COMPRESS_NONE),
7     "unable to run model");

```

---

Listing 3.15: Sending an inference request in the Triton coupling

We delay extracting the output data from the gRPC-owned buffer until we reach the `MPI_Scatterv`, because said buffer is owned by and usually lost upon leaving the scope of the `InferResult` (Listing 3.16, lines 6–8). By making it global thus prolonging its lifetime we can save one copy operation and directly copy from the buffer with MPI. We also do a quick sanity check if the expected amount of data was received (lines 9–13).

---

```

1 unsigned int nFields = outputFields_.size();
2 size_t outputValuesMaster_byteSize;
3 results_ptr_.reset(results_);
4 if (isGPUMaster_) {
5     FAIL_IF_ERR(
6         results_ptr_ ->RawData(
7             outputName, (const uint8_t**) &outputValuesMaster_,
8             &outputValuesMaster_byteSize),
9             "unable to get result data for " + outputName);
10    if (outputValuesMaster_byteSize != inputValuesMaster_.size() *
11        sizeof(cppDatatype)) {
12        std::cerr << "error: received incorrect byte size for " <<
13            outputName << ": "
14            << outputValuesMaster_byteSize << std::endl;
15        std::exit(1);
16    }
17 }
18 MPI_Scatterv(&outputValuesMaster_[0], nCellDataRecv_,
19             nCellDataRecvOffset_, MPI_DOUBLE, &outputValues_[0], nCells_ *
20             nFields, MPI_DOUBLE, 0, workGroupComm_);

```

---

Listing 3.16: `MPI_Scatterv` in the Triton coupling

## 3.6 Torch-TensorRT

TensorRT [3] is a “Deep Learning Optimizer and Runtime” [9] by Nvidia that aims to improve performance by transforming machine learning models to use Nvidia hardware as efficiently as possible. Torch-TensorRT [9] is a new add-on library to

### 3 Frameworks Coupling

PyTorch that facilitates this optimization and requires only very few changes to an existing PyTorch/libtorch implementation.

The most significant difference is that the Torch-TensorRT ahead-of-time model compiler is unable to produce a model that uses 64 bit floating point `double` values. We adjusted our model creation code from the basic PyTorch implementation (Listing 3.3) to use the default 32 bit floating point numbers instead for this reason (Listing 3.17).

---

```
1 m = FlexMLP(n_inp, n_out, n_neurons).float().requires_grad_(False)
```

---

Listing 3.17: Creating our neural network model for Torch-TensorRT in PyTorch using Python

The biggest change is in the initialization of the coupling (Listing 3.18) which is additionally complex due to our specific requirements. Compiling the model requires a specification of `input` tensor shapes for which the compiler optimizes the model. It is possible to define a single static shape, in which case the model can only be executed with input tensors of that shape later. We nearly always have a remainder batch because the total data amount is not divisible by the batch size. Hence we choose a dynamic input shape definition instead, consisting of a minimal shape we set to the remainder size, an optimal, and a maximal shape, the latter two of which we set to the full batch size as these will have the biggest impact even if there is only one full and one smaller remainder batch (lines 2–5). We need to define the target device to compile the model on because it will also be used to run the model (lines 7–9). Our testing has shown that the usual `to()` functions to which we are used from plain PyTorch are not sufficient to definitely execute a model on a particular device, instead the compilation locks the device in. The Tensors themselves are modified to use 32 bit floating point numbers (line 8) to match the model.

---

```
1 auto torchDtype = torch::kFloat;
2 std::vector<int64_t> remainder_shape = {remainder, nFields};
3 std::vector<int64_t> batch_shape = {batchSize, nFields};
4 torch_tensorrt::Input input(remainder_shape, batch_shape,
    batch_shape, torchDtype);
5 torch_tensorrt::torchscript::CompileSpec info({input});
6 info.enabled_precisions = {torchDtype};
7 torch_tensorrt::Device device;
8 device.gpu_id = myGPUDevice_;
9 info.device = device;
10 torchModel_ = torch_tensorrt::torchscript::compile(origModel,
    info);
11 torchModel_.to(torch::Device(torch::kCUDA, myGPUDevice_));
12
13 torch::TensorOptions options(torch::kFloat32);
14 inputTensor_ = torch::ones({nCells_, nFields}, options);
15 outputTensor_ = torch::ones({nCells_, nFields}, options);
```

---

Listing 3.18: Initialization of the Torch-TensorRT coupling

As we are dealing with 32 bit float data within the model now, we downsample the 64 bit double precision before communicating through MPI, saving 50% of transmitted data. We leave the MPI call unmodified apart from adjusting the data type (Listing 3.19).

---

```
1 MPI_Gatherv(inputTensor_.data_ptr(), nCells_*nFields, MPI_FLOAT,
             inputTensorMaster_.data_ptr(), nCellDataRecv_,
             nCellDataRecvOffset_, MPI_FLOAT, 0, workGroupComm_);
```

---

Listing 3.19: MPI in the Torch-TensorRT coupling

The batched execution of the model remains unmodified from the PyTorch implementation (Listing 3.6).

## 3.7 SOL

SOL is “an AI acceleration middleware” [33] by NEC that provides back ends for CPUs, GPUs, and the NEC SX-Aurora TSUBASA “Vector Engine” (VE). We have access to a host with 8 SX-Aurora devices (type 10B<sup>10</sup>) at the RWTH IT Center and evaluate them using PyTorch through SOL. NEC provides a VE Driver API (VEDA) [28] that is used to interact with the devices.

In order to run our model on the VEs from our C++ code, we first need to “deploy” it to the devices, which generates optimized C/C++ code. We do this by adding the deployment call on to our original PyTorch (Listing 3.3) Python script, see Listing 3.20. However note that contrary to Torch-TensorRT (Section 3.6 Listing 3.18 lines 2–5) we can only define one fixed batch size.

---

```
1 input = torch.ones(n_batch, n_inp, dtype=data_type)
2 dargs = {
3     "lib_name": "libFlexMLP",
4     "func_name": "forward",
5     "path": "./libFlexMLP",
6     "device_type": "ve"
7 }
8 sol.deploy(m, "shared_lib", dargs, input)
```

---

Listing 3.20: Creating our neural network model in SOL and PyTorch using Python

This script also generates a `CMakeLists.txt` which we use to compile the optimized model code for execution on the VEs using the NEC compiler. Due to the way VEDA handles memory pointers, we need to build a wrapper to dereference the memory pointers on the VEs before passing them into the `forward()` function against our compiled model, resulting in the device object `my_wrapper.vso` (Listing 3.21).

Within our coupling itself (Listing 3.22) we don’t need to use SOL-specific calls, instead we initialize our VE devices using VEDA, allocating a device to each “GPU master” rank (line 2). We load the device kernel `my_wrapper.vso` by filename

---

<sup>10</sup><https://www.nec.com/en/global/solutions/hpc/sx/VE-Card-Edge.html>

---

```

1 extern "C" void predict(VEDAdeviceptr input_, VEDAdeviceptr
  output_) {
2
3     const double* input = VEDAptr<double>(input_).ptr();
4     const double* rawInput;
5     vedaMemPtr(&rawInput, input_);
6
7     double* output = VEDAptr<double>(output_).ptr();
8     double* rawOutput;
9     vedaMemPtr(&rawOutput, output_);
10
11     forward(0, rawInput, rawOutput);
12 }

```

---

Listing 3.21: my\_wrapper manages memory between host and device

(line 3) and get its inference function `predict()` (line 4) we created in Listing 3.21, we also allocate memory for the tensors on the devices (lines 6–7).

---

```

1 CHECK(vedaInit(0));
2 CHECK(vedaCtxCreate(&ctx, 0, myGPUDevice_));
3 CHECK(vedaModuleLoad(&mod, modelName_.c_str()));
4 CHECK(vedaModuleGetFunction(&func, mod, "predict"));
5
6 CHECK(vedaMemAllocAsync(&inputTensorGPU_, batchSize * nFields_ *
  sizeof(double), 0));
7 CHECK(vedaMemAllocAsync(&outputTensorGPU_, batchSize * nFields_ *
  sizeof(double), 0));

```

---

Listing 3.22: Initialization of the SOL coupling

Execution of the model (Listing 3.23) is as easy as copying data to the device, launching the kernel with the previously set `func`, and copying the result back. We facilitate slicing using pointer arithmetic in the copy operations. Note that VEDA and the VE devices support asynchronous copying operations, i.e., they “don’t need to synchronize the execution between device and host.” [28]

---

```

1 for(int i = 0; i < nSlices; i++)
2 {
3     CHECK(vedaMemcpyHtoDAsync(inputTensorGPU_,
  inputTensorMaster_+(i*batchsize), batchSize * nFields_ *
  sizeof(double), 0));
4     CHECK(vedaLaunchKernel(func, 0, inputTensorGPU_,
  outputTensorGPU_));
5     CHECK(vedaMemcpyDtoHAsync(outputTensorMaster_+(i*batchsize),
  outputTensorGPU_, batchSize * nFields_ * sizeof(double), 0));
6 }
7 CHECK(vedaCtxSynchronize());

```

---

Listing 3.23: Batch processing the SOL coupling

# 4 Performance Model

## 4.1 POP Model

We model the performance of our coupling according to method proposed by Rosas et al. [31] and established by the European Union (EU) Center of Excellence (CoE) for Performance Optimization and Productivity (POP) [27]. The model is based on a hierarchy of efficiency metrics for parallel codes, and there are multiple tools to capture these metrics. The metrics are not directly measurable but instead derived from measuring the total run time  $TT$  of each process and thread, which is split up into computational time  $CT$  and time in parallel runtimes. In our case we only use MPI for parallelization, thus our run time is split into computational time and time spent within the MPI runtime (i.e., communication overhead). For the same reason and brevity we talk simply of “processes” in the following when in the full POP model metrics are based on parallelism through both multithreading and multiprocessing. The trace is simulated again to obtain theoretical optimal run times ( $TT_o, CT_o$ ) when using an ideal network with unlimited bandwidth and no delay. Figure 4.1 shows the metrics as nodes in a tree-like structure and connects them with arrows where one metric at an arrow tip may be calculated from the two others on the arrow tails by multiplication. Transfer efficiency  $TE$  is the ratio of the theoretical optimal total run time on an ideal network to the real total run time (Equation 4.1). Serialization efficiency  $SE$  is calculated as the maximum quotient of the theoretical computation time and the theoretical total run time on an ideal network (Equation 4.2). It describes the efficiency that is lost due do processes waiting for other processes to become ready to communicate. Load balance  $LB$  is the ratio of the mean real computation time to the maximum real computation time and indicates how equally the work is split up between all processes (Equation 4.3). Values for the metrics are between 0 and 1, where values  $> 0.8$  are considered good.

$$TE = \frac{TT_o}{TT} \quad (4.1)$$

$$SE = \max_{processes} \left\{ \frac{CT_o}{TT_o} \right\} \quad (4.2)$$

$$LB = \frac{\text{mean}_{processes} \{CT\}}{\max_{processes} \{CT\}} \quad (4.3)$$

We use this model to understand how efficiency is impacted by the need to switch between all ranks computing in the OpenFOAM phase and only the GPU master

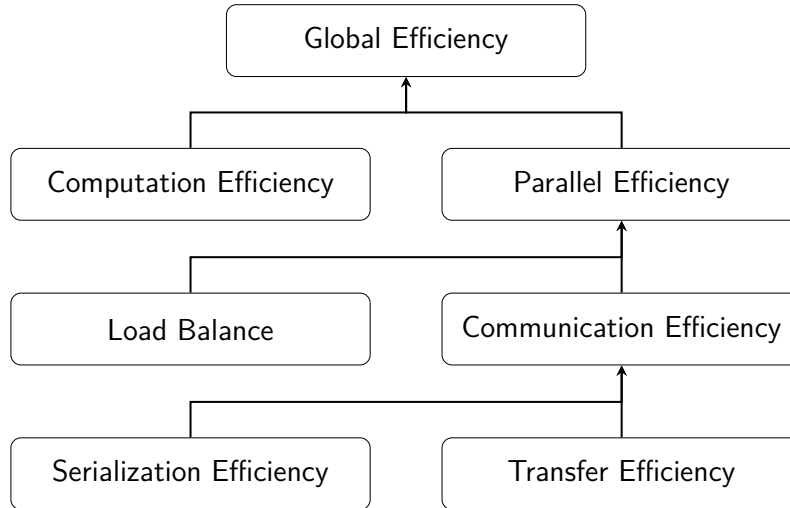


Figure 4.1: The metrics of the POP model and their relationships.

rank working in the ML phase. The idle time for CPU ranks during inference may impact load balance and serialization efficiency. Introduction of additional MPI communication necessary to gather and scatter data centrally on and from the GPU master ranks may impact serialization and transfer efficiency.

## 4.2 POP Model for Hybrid Codes

Giménez et al. proposed a model for “hybrid” codes that use multiple parallelization technologies such as MPI+OpenMP or MPI+CUDA based on the POP model. For example in our case we have a hybrid MPI+CUDA code for ML frameworks that use an Nvidia GPU as hardware accelerator. In the hybrid model the different parallel programming models’ performance metrics are measured and modeled independent from each other, load balance and communication efficiency can be given for only MPI, only CUDA, or both combined. The whole application’s hybrid load balance and communication efficiency can be calculated by multiplying the independent programming models’ respective metrics, similar to the POP model hierarchy. Additionally, the parallel efficiencies of each programming model can be calculated by multiplying its load balance and communication efficiency. Multiplying both programming models’ parallel efficiency again yields the hybrid parallel efficiency of the whole application. This new hybrid hierarchy is depicted in Figure 4.2. Formerly existing metrics are colored blue to indicate they are now hybrid metrics, and new metrics are colored green.

The hybrid model enables us to investigate a loss of efficiency more precisely by distinguishing between different programming models used in a hybrid application. Where we would see only a loss in the hybrid load balance or communication efficiency using the plain POP model, with the hybrid model we can easily see whether it is caused by the MPI or the CUDA part of the application. It also allows us to

compare the programming models' whole impact on the parallel efficiency directly.

In the end, we were unable to measure CUDA metrics in the POP context due to technical problems with the measuring tool and hence unable to apply the hybrid model to our measurements.

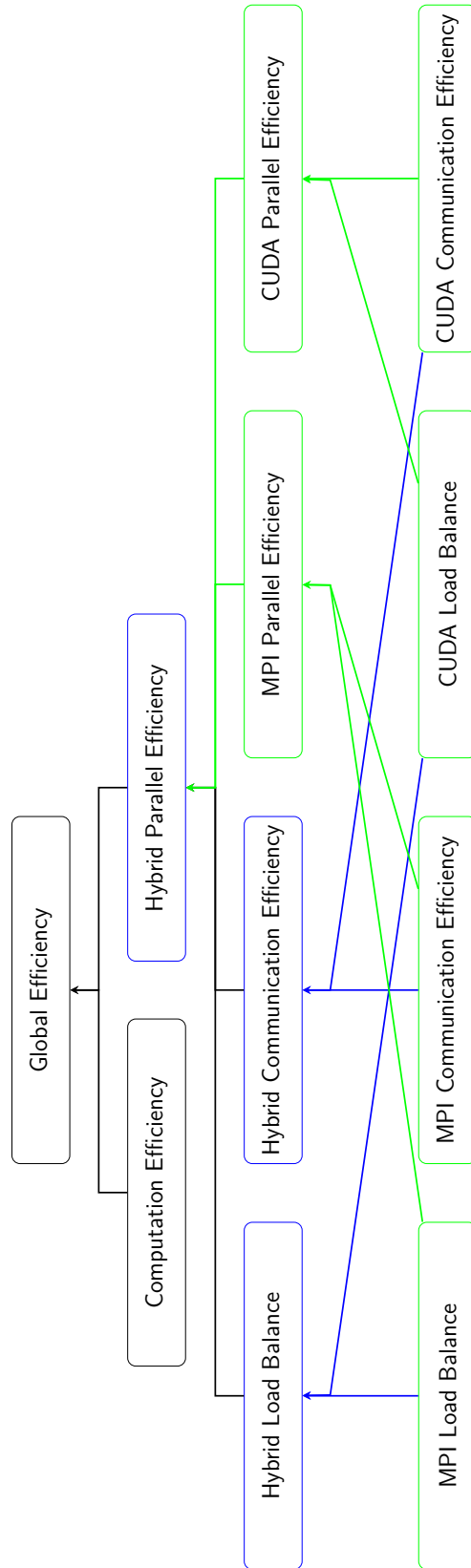


Figure 4.2: The metrics of the hybrid POP model and their relationships.

# 5 Results

## 5.1 Hardware, Framework and Library Versions

We conduct our experiments on the CLAIIX-2018<sup>1</sup> high performance computing cluster at RWTH Aachen University. The cluster consists of ca. 1250 nodes, each equipped with two Intel Xeon Platinum 8160 processors<sup>2</sup> and 192 GB main memory. Each CPU core has two AVX-512 FMA units that can compute two (multiply and add) operations for vectors of  $\frac{512 \text{ bit}}{64 \text{ bit}} = 8$  double precision floating point numbers per unit and clock cycle. Running these units on all cores yields  $2^{\frac{\text{AVX-512 FMA units}}{\text{core}}} \times 2^{\frac{\text{FLOPS}}{\text{FMA unit}}} \times \frac{8}{\text{AVX-512}} \times 24^{\frac{\text{cores}}{\text{CPU}}} \times 2 \text{ CPUs} \times 2.1 \text{ GHz} \approx 3.2 \text{ TFLOPS}$  theoretical peak performance per cluster node<sup>34</sup>. There are 54 more nodes of the same configuration with two additional Nvidia Tesla V100-SXM2-16GB GPUs. According to the data sheet<sup>5</sup> and whitepaper<sup>6</sup>, each GPU has 2560 double precision, 5120 single precision CUDA cores and 640 “mixed precision” (half/single) tensor cores for a peak performance of 7.8 TFLOPS (double precision), 15.7 TFLOPS (single precision), and 125 TFLOPS (mixed half/single tensor). Each GPU has 16 GB of HBM2 memory with a bandwidth of 900 GB/s.

The initial implementation of the PyTorch coupling provided to us used PyTorch/libtorch version 1.2.0 (Aug 8, 2019) built for CUDA 10.0. We updated it to version 1.10.1 (Dec 15, 2021) built for CUDA 11.3 to match the version required by Torch-TensorRT and also adjusted the dependencies accordingly. For compilation, we use the Intel compiler version 19.0.1.144 20181018 as our program crashes when using the GNU compiler, however we load the libstdc++<sup>7</sup> C++ standard library implementation shipped with GCC 8.2.0 to provide C++14 support.

We implemented the Torch-TensorRT coupling with its first official stable release version 1.0.0 (Nov 9, 2021), available as prebuilt binaries from GitHub<sup>8</sup>, where Nvidia also publishes the source code. Torch-TensorRT targets libtorch 1.10.0

---

<sup>1</sup><https://help.itc.rwth-aachen.de/service/rhr4fjjuttff/>

<sup>2</sup><https://help.itc.rwth-aachen.de/service/rhr4fjjuttff/article/fbd107191cf14c4b8307f44f545cf68a/>

<sup>3</sup><https://ark.intel.com/content/www/us/en/ark/products/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz.html>

<sup>4</sup><https://www.top500.org/system/179682/>

<sup>5</sup><https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf>

<sup>6</sup><https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

<sup>7</sup><https://gcc.gnu.org/onlinedocs/libstdc++/>

<sup>8</sup><https://github.com/NVIDIA/Torch-TensorRT/releases/tag/v1.0.0>

built for CUDA 11.3, cuDNN 8.2, and TensorRT 8.0.3.4 according to the project’s README file<sup>9</sup>. The exact CUDA and cuDNN versions are not available on the RWTH CLAIX module system, hence we used CUDA 11.4 and cuDNN/8.3.2 instead without issue.

We implemented the TensorFlow coupling with TensorFlow/libtensorflow 2.6.0 which was tagged already on Aug 11, 2021, but at the time of writing still the newest version of the C API prebuilt for Linux available on the official website<sup>10</sup>. The website<sup>11</sup> also states a requirement for CUDA 11.2 and cuDNN 8.1.0. Because cuDNN 8.1.0 is not available on the RWTH CLAIX module system, we replaced it by the patched version 8.1.1. For compilation, we use the GNU compiler version 10.1.0 which includes C++17 support as we had trouble with the intel compiler in this case.

We implemented the Nvidia Triton coupling against the r22.01 (January 2022) release branch of the client library. Nvidia primarily releases the Triton server in an OCI<sup>12</sup> compatible “container” package which one can run using a container engine such as Docker<sup>13</sup> or Podman<sup>14</sup>. One core concepts of such containers is that it delivers a software with all required dependencies such as CUDA drivers preinstalled, offering an easy installation process to the user as well as providing a fixed environment for the software. The combination of supported software versions can be found on Nvidia’s support matrix website<sup>15</sup>. The RWTH CLAIX clusters offer the “Singularity” (now renamed “Apptainer”)<sup>1617</sup> container engine version 3.8.7-1.el7 with which we were able to successfully run the provided container and execute a test example program.

We implemented the SOL coupling using SOL 0.4.2.1 (Sep 20, 2021) and VEDA version 0.10.6 available in the RWTH CLAIX module system. For compilation of the coupling itself, we reuse the compiler configuration from PyTorch. We use CMake 3.21.1 with GCC 8.2.0 next to the NEC compiler version 3.4.0 to compile the model and wrapper for offloading. SOL is compatible with multiple frameworks including TensorFlow<sup>18</sup>, but best supports PyTorch version 1.9.0, so we use a model created with that version as input to the optimizer. We conduct the tests with SOL on a single host with two Intel Xeon Silver 4108 processors, 92 GB main memory, and 8 NEC SX-Aurora TSUBASA 10B<sup>19</sup> Vector Engine devices, installed at the IT Center of

---

<sup>9</sup><https://github.com/NVIDIA/Torch-TensorRT/blob/8801573d1800b72320f99cac7b21c5c96a157596/README.md#dependencies>

<sup>10</sup>[https://www.tensorflow.org/install/lang\\_c#download\\_extract](https://www.tensorflow.org/install/lang_c#download_extract)

<sup>11</sup>[https://www.tensorflow.org/install/gpu#software\\_requirements](https://www.tensorflow.org/install/gpu#software_requirements)

<sup>12</sup><https://opencontainers.org>

<sup>13</sup><https://www.docker.com>

<sup>14</sup><https://podman.io>

<sup>15</sup><https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html>

<sup>16</sup><https://help.itc.rwth-aachen.de/service/rhr4fjjuttff/article/76576413f58040e78acab4332d9e68e3/>

<sup>17</sup><https://apptainer.org>

<sup>18</sup><https://sol.neclab.eu/docs/v0.4.2/frameworks/compatibility.html>

<sup>19</sup><https://www.nec.com/en/global/solutions/hpc/sx/Vector-Engine-Types.html#10B>

RWTH Aachen University. According to the data sheet, each device has 8 cores with a theoretical peak performance of 268 GFLOPS (double precision)/537 GFLOPS (single precision) each for a cumulative peak performance of 2.15 TFLOPS (double precision)/4.30 TFLOPS (single precision) per device. Each VE is equipped with 48 GB memory with a bandwidth of 1.22 TB/s.

We use the Intel MPI suite version 2018.4.274 for all couplings as there were difficulties getting the OpenMPI library to work with a heterogeneous mix (with and without GPU) of cluster nodes.

## 5.2 Instrumenting for Profiling and Tracing

### 5.2.1 Nvidia GPUs

We use Nvidia’s Nsight tools available from the RWTH CLAIX module system to generate and inspect traces of the frameworks running on Nvidia GPU back ends. The Nsight suite consists of a triad of tools that focus on different levels of performance analysis: Nsight Systems gives a performance overview over the whole application, Nsight Compute focuses on CUDA kernel performance, and Nsight Graphics focuses on graphics APIs in particular, including Direct3D, Vulkan, OpenGL, and more. The relationship of the three tools is shown in Figure 5.1 alongside the suggested workflow. More variants, like integrations for several IDEs, are available. Nsight Systems is able to give us enough insight into the kernels executed on the GPUs for our means while also indicating communication times which we’re interested in. Its version 2021.2.1.58-642947b is officially compatible to CUDA 11.4 of our PyTorch implementations and also worked with CUDA 11.2 of our TensorFlow coupling.

Nsight itself is called like a wrapper around our application, which enables it to inject tracing capabilities for its supported APIs, including CUDA without requiring further instrumentation. When viewing a full application trace however the GUI application fails to display any CUDA activity, and the trace is so big that the application takes very long to load it and crashes upon loading a second one. Nvidia themselves recommend in their documentation and workshops to trace only the specific parts of the program that one is interested in to reduce overhead and retain responsiveness in Nsight’s GUI. Nsight Systems offers several ways to trim down the trace range: it is possible to set a time range when starting the trace if it is known in advance, it is possible to trigger it manually by hotkey, and it can be triggered automatically by calls to the CUDA profiler or Nvidia Tools Extension Library (NVTX) APIs.<sup>21</sup> We choose NVTX as triggering mechanism because it allows us to annotate regions of our application at the same time which we can also view in the Nsight Systems GUI in the resulting trace and draw conclusions as to which parts have performance impacts.

<sup>20</sup><https://developer.nvidia.com/sites/default/files/akamai/Nsight-Diagram.png>

<sup>21</sup><https://docs.nvidia.com/nsight-systems/UserGuide/index.html>

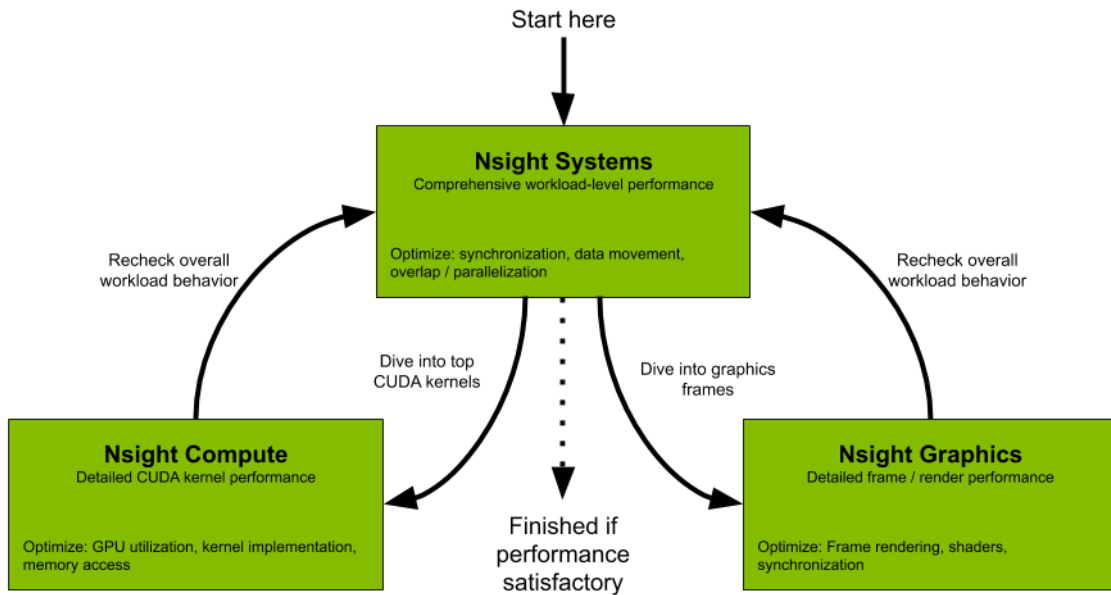


Figure 5.1: Optimization workflow using Nvidia Nsight tools, Source: Nvidia<sup>20</sup>

NVTX can mark ranges using “push-pop ranges” that are bound to a single thread and “start-end ranges” that are bound to a process instead of a thread, however using push-pop lead to confusing and likely broken results in our traces, hence we employed start-end ranges such as in Listing 5.1 which gave the expected results. Ranges are named by the user on creation in order to differentiate them later when viewing the trace, but also to indicate the specific range that shall be used to trigger the trace start and end when using NVTX as trigger. One pitfall with NVTX as trigger is that by default Nsight Systems only listens for specially “registered strings”<sup>2223</sup> in order to avoid overhead. We chose to disable this behavior by supplying the environment variable `NSYS_NVTX_PROFILER_REGISTER_ONLY=0` as we have few ranges in our comparatively little code.

## 5.2.2 NEC Vector Engines

The vector engine driver and runtime APIs offer their builtin `ftrace` tool [29]. We can enable it simply by adding the `-ftrace` argument to the compile options of our optimized model (compare Section 3.7) and setting the environment variable `VEDA_FTRACE=1` when running the application.

<sup>22</sup>[https://nvidia.github.io/NVTX/doxygen/group\\_\\_string\\_\\_registrat\\_i\\_o\\_n.html](https://nvidia.github.io/NVTX/doxygen/group__string__registrat_i_o_n.html)

<sup>23</sup><https://docs.nvidia.com/nsight-systems/UserGuide/index.html#example-interactive-cli-command-sequences>

---

```

1 #include "nvtx3/nvToolsExt.h"
2 void torchInference::batchedForward(int batchsize)
3 {
4     nvtxRangeId_t r1 = nvtxRangeStartA("batchedForward");
5     // batching setup code
6
7     for(int i = 0; i < nSlices; i++)
8     {
9         nvtxRangeId_t r2 = nvtxRangeStartA("batchedForward loop");
10        // batch submission code
11        nvtxRangeEnd(r2);
12    }
13    nvtxRangeEnd(r1);
14 }

```

---

Listing 5.1: Region markers with NVTX (PyTorch example)

### 5.2.3 Extrae

Extrae [2] is a performance data collection tool from the Barcelona Supercomputing Center (BSC) that supports many common platforms and programming models. An advantage over similar tools is the ability to use it by injecting a shared library into an already compiled and dynamically linked application by means of the LD\_PRELOAD environment variable on Linux systems such as RWTH CLAIX, hence requiring no source code modification or recompilation. We were able to successfully apply this mechanism with Extrae 4.0.0 to collect metrics on MPI, however we could not get the collection of CUDA data to work within the time constraints of the thesis. For viewing and evaluation of the traces we used the basic analysis tools version 0.3.6, paraver version 4.8.1, dimemas 5.4.1, numpy 1.21.5, and scipy 1.7.3.

## 5.3 Measurements with GPU-Accelerated Frameworks

The examined thermo-fluid simulation is computed by iteratively progressing over a time period. We call each iteration a “time step”. In most cases we were able to sample one “warm-up” and five more time steps of the simulation. We use that in the following to represent some statistical confidence in our data where applicable. However, for the configurations given by Table 5.1 we could only capture one warm-up and two more time steps. While this means that the statistical confidence is reduced in these cases, due the low variance in similar measurements we expect low variance for these values as well. Further, the Extrae tool was unable to capture data for the configurations given by Table 5.2.

| CPU nodes | GPU nodes | Framework   | Layer Size |
|-----------|-----------|-------------|------------|
| 1         | 0         | PyTorch-CPU | 500        |
| 1         | 0         | PyTorch-CPU | 1000       |
| 2         | 0         | PyTorch-CPU | 1000       |
| 4         | 0         | PyTorch-CPU | 1000       |
| 0         | 4         | TensorFlow  | 1000       |
| 8         | 0         | PyTorch-CPU | 1000       |

Table 5.1: These configurations were benchmarked with only three total time steps instead of six.

| CPU nodes | GPU nodes | Framework     | Layer Size |
|-----------|-----------|---------------|------------|
| 7         | 1         | TensorFlow    | 1000       |
| 8         | 0         | PyTorch-CPU   | 1000       |
| 15        | 1         | TensorFlow    | 1000       |
| 16        | 0         | OpenFOAM only | 1000       |

Table 5.2: No POP metrics could be captured for these configurations using Extrac.

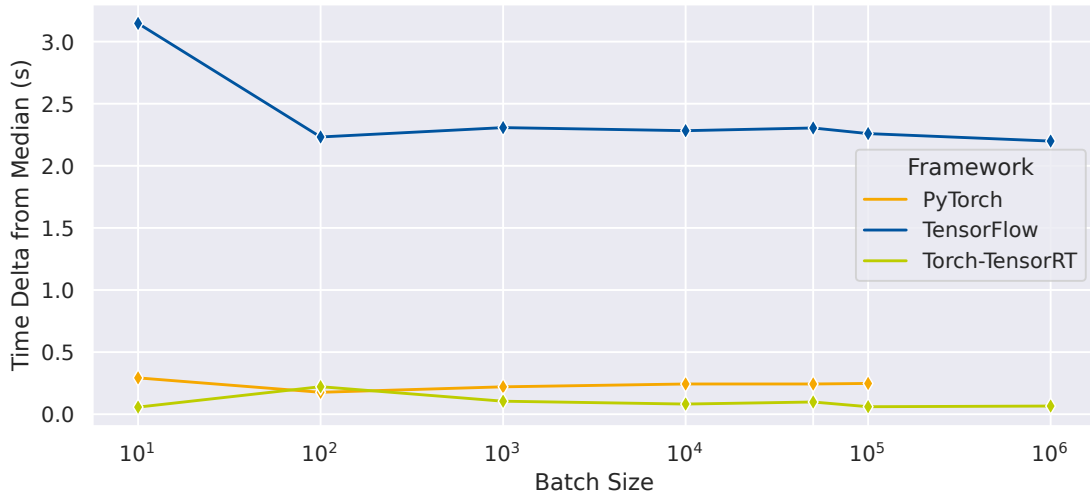
### 5.3.1 GPU Baseline Measurements

At first, we need to establish some ground truth about framework performance on a single GPU with regards to the basic variables of hidden layer size (amount of neurons per hidden layer) and batch size. We use a single cluster node (48 CPU cores, 2 V100 GPUs) for this benchmark and compare the run times of the ML coupling in different configurations. We capture the time of different stages of the coupling using `std::chrono::steady_clock::now()`, especially at the entry point from OpenFOAM into the coupling, after all input data has been gathered on the GPU master rank, after inference is finished, and after the results have been distributed across the workgroup. From these timestamps, we calculate the following time periods on each GPU master rank: “fromFOAM” for gathering and converting the OpenFOAM data to a tensor object, “Actual Framework Time” including batching and inference, “toFOAM” for scattering and converting from tensor to OpenFOAM, and “Total Coupling Time” for the sum of the former three. We calculate the maximum for each of the measured periods individually using `MPI_Allreduce`, for example `MPI_Allreduce(&myTotalMLTime, &maxTotalMLTime, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD)`. Note that the Maximum Total Coupling Time is not the sum of the three other maxima, but the total time the slowest GPU master took for the whole process.

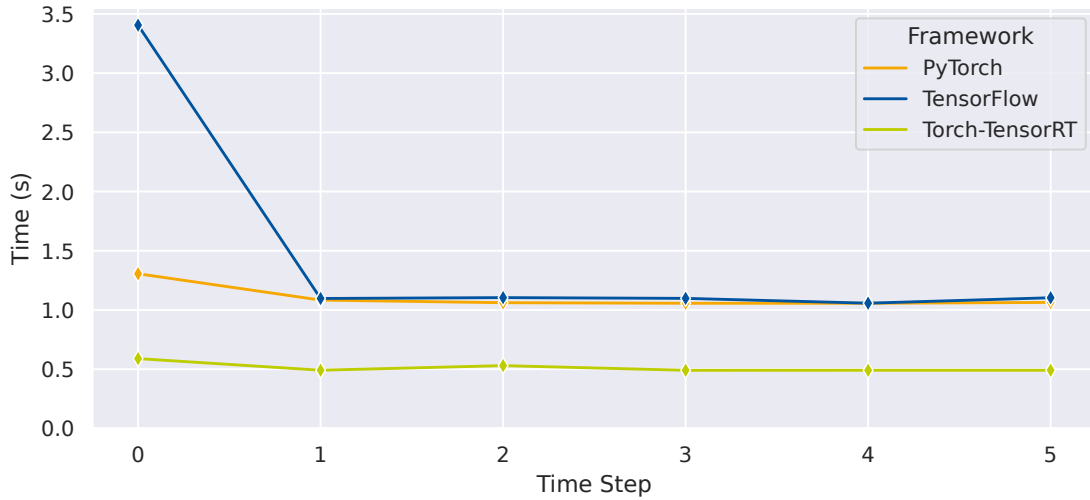
The first effect we notice is that some frameworks have a “warm-up” time, i.e., the framework only performs certain setup tasks when actually asked to perform inference for the first time, thus slowing down that first time step. The impact to the run times of this effect at different batch sizes is depicted in Figure 5.2a. All of PyTorch (Figure 5.2c), TensorFlow (Figure 5.2d), and Torch-TensorRT (Figure 5.2e)

### 5.3 Measurements with GPU-Accelerated Frameworks

take a certain amount of warm-up time, but TensorFlow takes significantly longer than the other frameworks as well as the inference itself (Figure 5.2b). We will remove the warm-up time step as an outlier from the following results where noted.



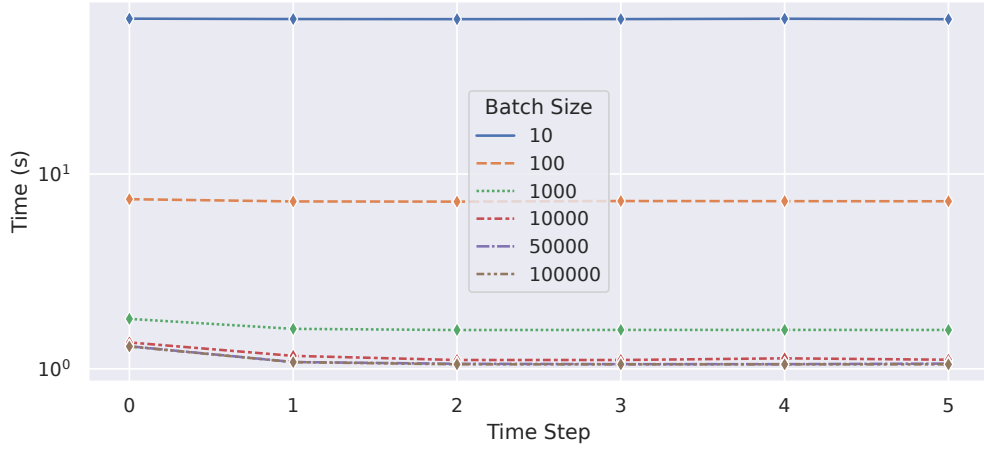
(a) Difference between actual framework time of the first (warm-up) time step and the median of the following 5 time steps at different batch sizes. All depicted frameworks have a mostly constant absolute warm-up time, but TensorFlow takes much longer than the other frameworks. PyTorch is unable to run at batch size of 10<sup>6</sup> due to memory constraints.



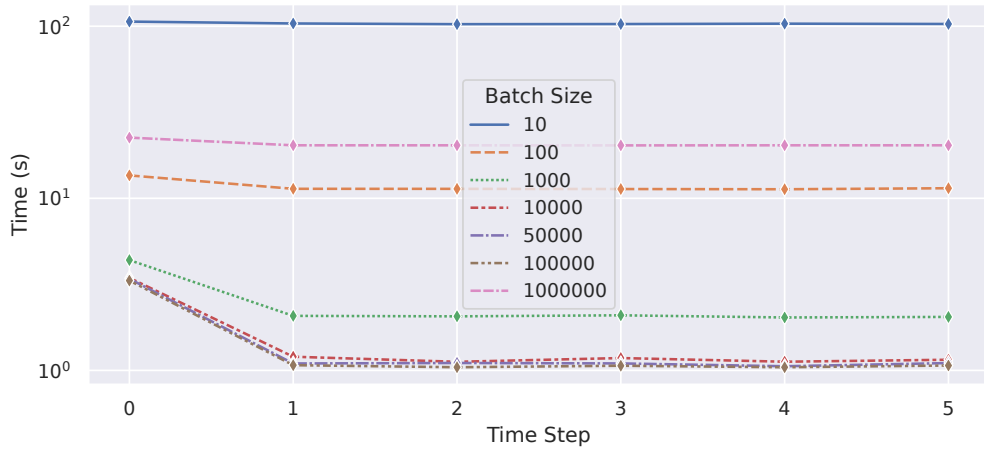
(b) Actual framework time of subsequent time steps for different frameworks at constant batch size 50 000.

Figure 5.2: Actual framework time on 2 V100 GPUs, fixed layer size of 1000 neurons.

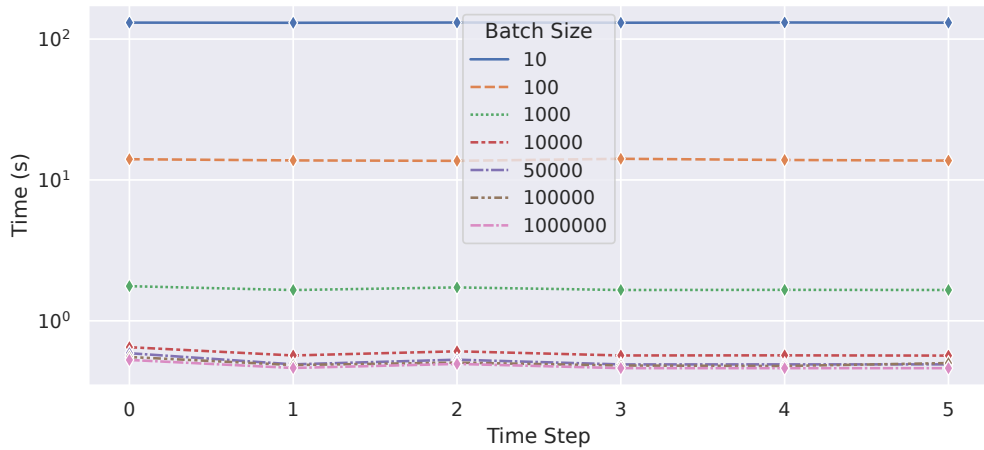
5 Results



(c) PyTorch time of subsequent time steps for different batch sizes.



(d) TensorFlow time of subsequent time steps for different batch sizes.



(e) Torch-TensorRT time of subsequent time steps for different batch sizes.

Figure 5.2: Actual framework time on 2 V100 GPUs, fixed layer size of 1000 neurons (cont.).

| Operation              | Duration        | Duration (%) |
|------------------------|-----------------|--------------|
| Memcpy HtoD            | 68.383 $\mu$ s  | 0.29%        |
| unrolled_elementwise   | 451.005 $\mu$ s | 1.93%        |
| volta_dgemm_128x64_tn  | 1.047 ms        | 4.49%        |
| vectorized_elementwise | 1.381 ms        | 5.92%        |
| unrolled_elementwise   | 453.534 $\mu$ s | 1.95%        |
| volta_dgemm_128x64_tn  | 15.720 ms       | 67.42%       |
| vectorized_elementwise | 1.384 ms        | 5.94%        |
| unrolled_elementwise   | 5.792 $\mu$ s   | 0.02%        |
| Memset                 | 3.232 $\mu$ s   | 0.01%        |
| scal_kernel            | 10.176 $\mu$ s  | 0.04%        |
| dgemm_largek           | 599.293 $\mu$ s | 2.57%        |
| Memcpy DtoH            | 61.920 $\mu$ s  | 0.27%        |
| Total (sum)            | 21.185 ms       | 90.86%       |
| Total                  | 23.317 ms       | 100%         |

Table 5.3: PyTorch batch duration per kernel as indicated by an Nvidia Nsight Systems trace on 2 V100 GPUs, fixed layer size of 1000 neurons, and batch size 50 000. The sum of the given kernel durations is only approximately 91% of the total indicated by Nsight Systems. The missing 9% are spent for unknown operations, possibly including tracing overhead.

To learn what causes the warm-up time and see the inner workings of each framework, we take a closer look at the different frameworks using the Nvidia Nsight Systems analysis tool (compare Section 5.2.1) on the same single node configuration. Thanks to Nsight Systems’ support for visualizing our NVTX annotations, the respective code regions in the trace are clearly visible, however note that the measured times may be distorted from “real” run times due to the tracing overhead, and the run times of the individual batches are very different within a single run. Despite that, we can still clearly see that the first batch takes much longer to process than all following batches, which is caused by `cudaMalloc` memory allocation calls (PyTorch, Torch-TensorRT) or `cuBLAS Create_v2` context creation (TensorFlow).

We also inspect the kernels running on the GPUs in a single batch (Tables 5.3, 5.4, 5.5). Each batch follows a similar pattern across frameworks, and the same pattern for a single framework. For all frameworks we see a “Host to Device” `Memcpy HtoD` copy operation at the start of the batch, and the reverse `Memcpy DtoH` at the end of the batch.

We can inspect the details of the operations to view how much data is copied, e.g., in our test case with 50 000 cells per batch, and 2 `double` (64 bit = 8 B) valued fields per cell, we expect  $50\,000 \text{ cells} \times 2 \frac{\text{fields}}{\text{cell}} \times 8 \text{ B} = 800\,000 \text{ B} = 800 \text{ kB}$ , which matches the amount indicated by Nsight Systems (Torch-TensorRT only uses half that due to its single precision 4 B float data type). Further we can view the throughput of the copy operations, which is indicated at around 11 GiB/s for host

| Operation             | Duration        | Duration (%) |
|-----------------------|-----------------|--------------|
| Memcpy HtoD           | 67.904 $\mu$ s  | 0.32%        |
| volta_dgemm_128x64_nn | 459.711 $\mu$ s | 2.18%        |
| BiasNHWCKernel        | 1.029 ms        | 4.89%        |
| EigenMetaKernel       | 1.013 ms        | 4.81%        |
| volta_dgemm_64x64_nn  | 15.782 ms       | 74.96%       |
| BiasNHWCKernel        | 1.029 ms        | 4.89%        |
| EigenMetaKernel       | 1.008 ms        | 4.791%       |
| Memset                | 3.457 $\mu$ s   | 0.02%        |
| scal_kernel           | 8.512 $\mu$ s   | 0.04%        |
| dgemm_largek          | 586.686 $\mu$ s | 2.79%        |
| BiasNHWCKernel        | 5.440 $\mu$ s   | 0.03%        |
| Memcpy DtoH           | 61.440 $\mu$ s  | 0.29%        |
| Total (sum)           | 21.054 ms       | 100%         |

Table 5.4: TensorFlow batch duration per kernel as indicated by an Nvidia Nsight Systems trace on 2 V100 GPUs, fixed layer size of 1000 neurons, and batch size 50 000.

| Operation                      | Duration        | Duration (%) |
|--------------------------------|-----------------|--------------|
| Memcpy HtoD                    | 35.584 $\mu$ s  | 0.35%        |
| volta_sgemm_128x64_nn          | 235.647 $\mu$ s | 2.33%        |
| eltwise                        | 643.901 $\mu$ s | 6.37%        |
| volta_sgemm_128x64_nn          | 7.811 ms        | 77.24%       |
| eltwise                        | 644.094 $\mu$ s | 6.37%        |
| volta_sgemm_32x32_sliced1x4_nn | 354.878 $\mu$ s | 3.51%        |
| eltwise                        | 5.696 $\mu$ s   | 0.06%        |
| Memcpy DtoH                    | 31.616 $\mu$ s  | 0.31%        |
| Total (sum)                    | 9.762 ms        | 96.53%       |
| Total                          | 10.113 ms       | 100%         |

Table 5.5: Torch-TensorRT batch duration per kernel as indicated by an Nvidia Nsight Systems trace on 2 V100 GPUs, fixed layer size of 1000 neurons, and batch size 50 000. The sum of the given kernel durations is only approximately 97% of the total indicated by Nsight Systems. The missing time is spent for unknown operations, possibly including tracing overhead.

to device, and 12 GiB/s for device to host, regardless of framework. In general we can follow the procedure given by the MLP model (compare Section 2.2): Each framework runs 3 volta architecture specific (matching our V100 GPUs) “General Matrix Multiply” (GEMM, [15]) kernels of the used data type (double or single precision float). PyTorch uses two  $128 \times 64$  TN kernels followed by one `largek` kernel, TensorFlow uses one  $128 \times 64$  and one  $64 \times 64$  NN kernels followed by one `largek` kernel, and Torch-TensorRT uses two  $128 \times 64$  NN kernels followed by a  $32 \times 32$  sliced  $1 \times 4$  NN kernel. We can trace other operations around each GEMM to the bias and application of the ReLU activation function of the MLP, however the specifics differ greatly between the frameworks. PyTorch uses elementwise kernels from its `at::native` namespace, TensorFlow uses its own kernels and the Eigen<sup>24</sup> library, and Torch-TensorRT uses `cuEltwise` kernels. The specific kernels chosen by the framework appear to depend on the characteristics including the model’s structure, amount of neurons per layer, batch size, and data type. It is clearly noticeable that the GEMM of the transition between the large hidden layers has the highest performance impact by far, it takes around 15 ms (7.5 ms for Torch-TensorRT) out of 21 ms (10 ms), i.e., 70 to 75% of the time.

One significant variable that affects framework and accelerator performance is the batch size. We measure the run time across frameworks with batch sizes in orders of magnitude between  $10^2$  and  $10^6$ , depicted in Figure 5.3. The coupling run time is 1 to 2 orders of magnitude higher when using batches of fewer than  $10^3$  elements compared to an optimum of 1 s when using at least  $10^4$  elements per batch. Further, at extremely high batch sizes such as  $10^6$  elements, TensorFlow slows down to 20 s and PyTorch is not able to perform inference at all as it runs out of memory (see Equation 2.1). For the following experiments, we settle on a batch size of 50 000, because it is large enough for the frameworks to perform well.

Another basic variable is the amount of neurons per hidden layer of our MLP. We measure the run time of the inference across frameworks and change the amount of neurons. The results are visualized by Figure 5.4.

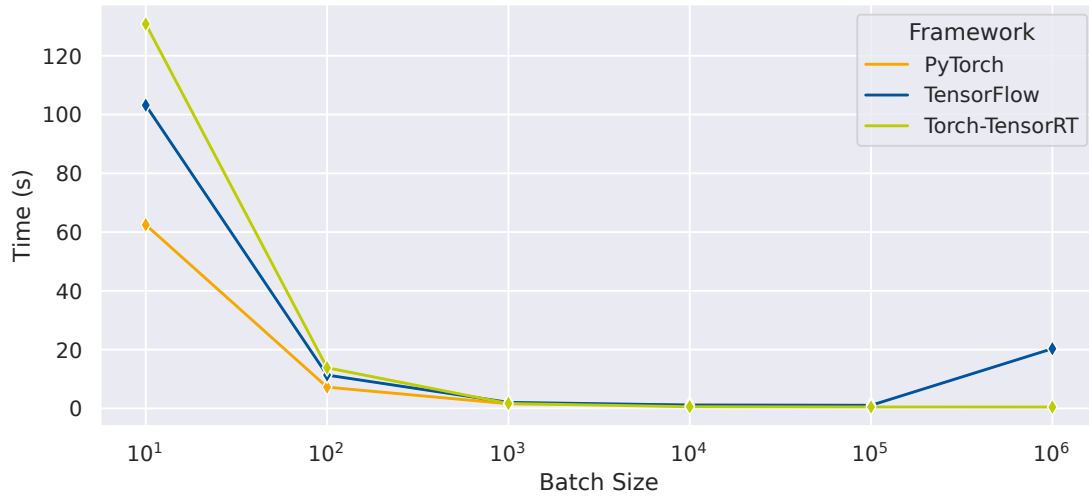
We measure a change in run time from 3 s to 8 s with PyTorch-CPU and 0.4 s to 1.1 s with PyTorch and TensorFlow. Both times the run time is increased by a factor of approximately 2.7. In the following, we use a hidden layer size of 1000 neurons unless indicated otherwise.

By comparing the three measured time ranges “fromFOAM”, “Actual Framework”, and “toFOAM” in Figure 5.5, it is evident that the processing time outweighs the communication time within our couplings by two orders of magnitude. The TensorFlow coupling is generally the slowest, especially during communication with FOAM.

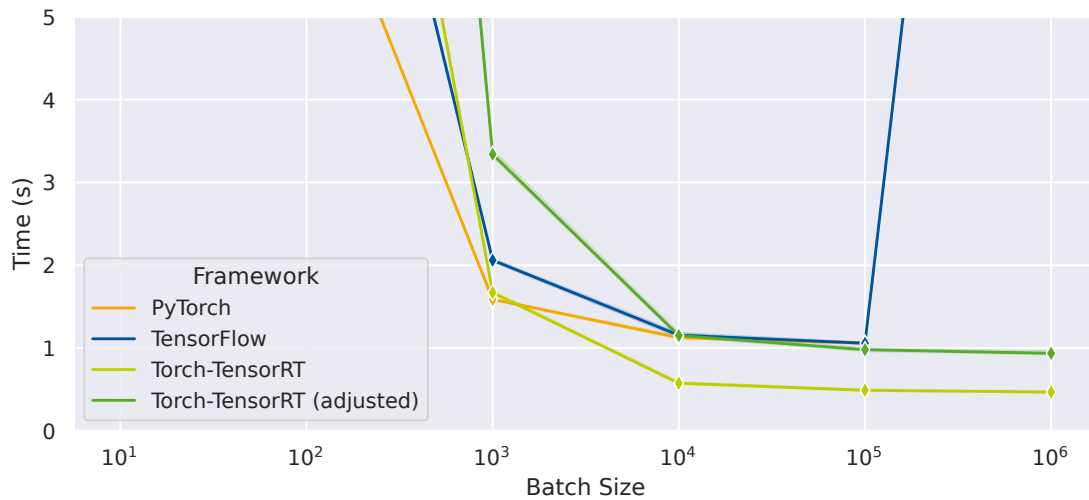
---

<sup>24</sup><https://gitlab.com/libeigen/eigen>

## 5 Results

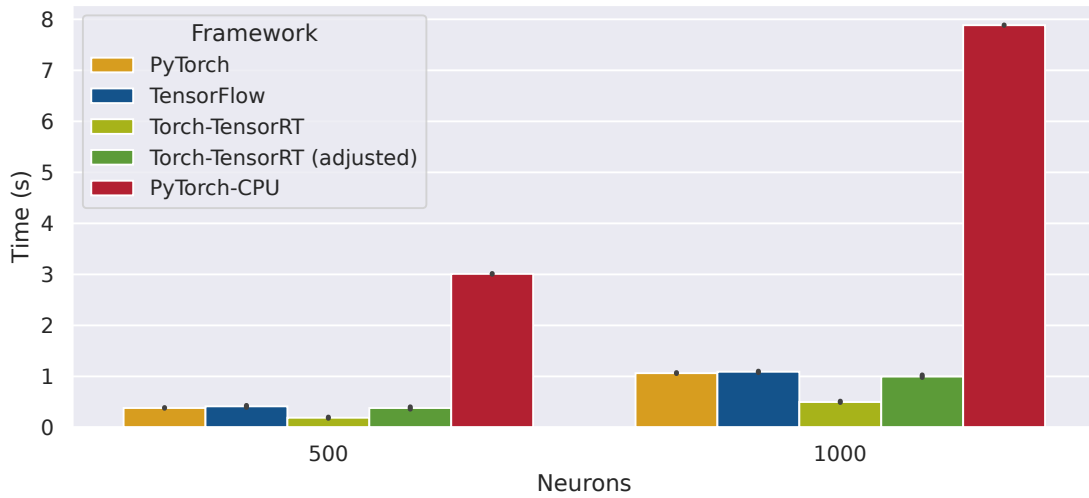


- (a) The run time increases significantly with too small batches ( $< 10^3$ ), and in some cases also for too big batches (TensorFlow,  $> 10^5$ ). Torch-TensorRT is significantly slower at small batch sizes despite running only at single precision while the other frameworks run at double precision. The data point for PyTorch at  $10^6$  is missing, because this batch size is too large for this case.

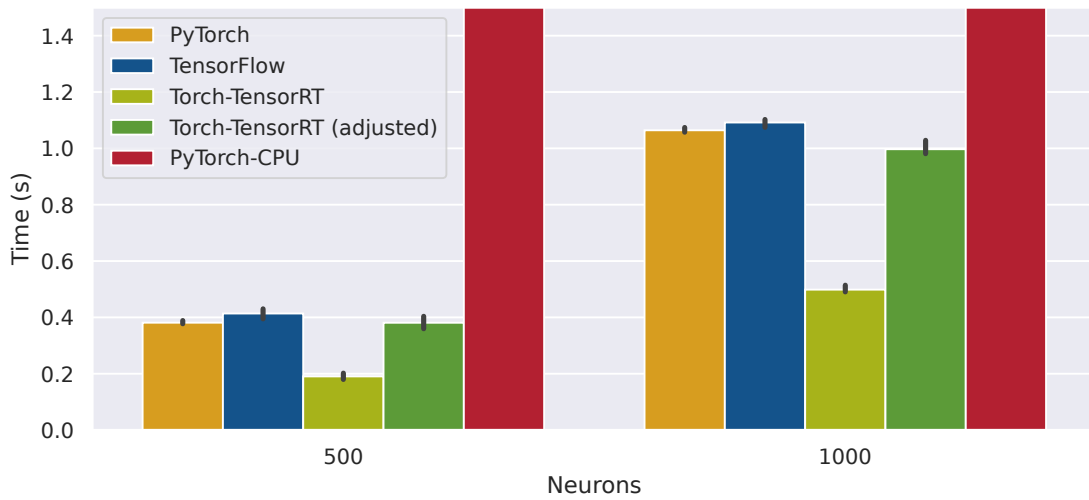


- (b) Close-up of 5.3a: All frameworks (Torch-TensorRT adjusted for double precision) are equally fast between batch sizes  $10^4$  and  $10^5$ .

Figure 5.3: Actual framework time by batch size (horizontal axis) and ML framework (color) on 2 V100 GPUs, fixed layer size of 1000 neurons. Mean of 5 time steps, excluding the warm-up time step.



(a) The run time increases significantly with bigger hidden layers. PyTorch running on all 48 CPU ranks is significantly slower than the hardware accelerated versions at this scale, even when accounting for coupling overhead. The Torch-TensorRT coupling runs twice as fast, as is expected when using only half precision compared to the other frameworks.



(b) Close-up of 5.4a: All GPU-accelerated frameworks (Torch-TensorRT adjusted for double precision) are equally fast.

Figure 5.4: Actual framework time by amount of neurons per hidden layer (horizontal axis) and ML framework (color) on two V100 GPUs, fixed layer size of 1000 neurons. Mean of 5 time steps, excluding the warm-up time step. The tips indicate the confidence interval of 95%.

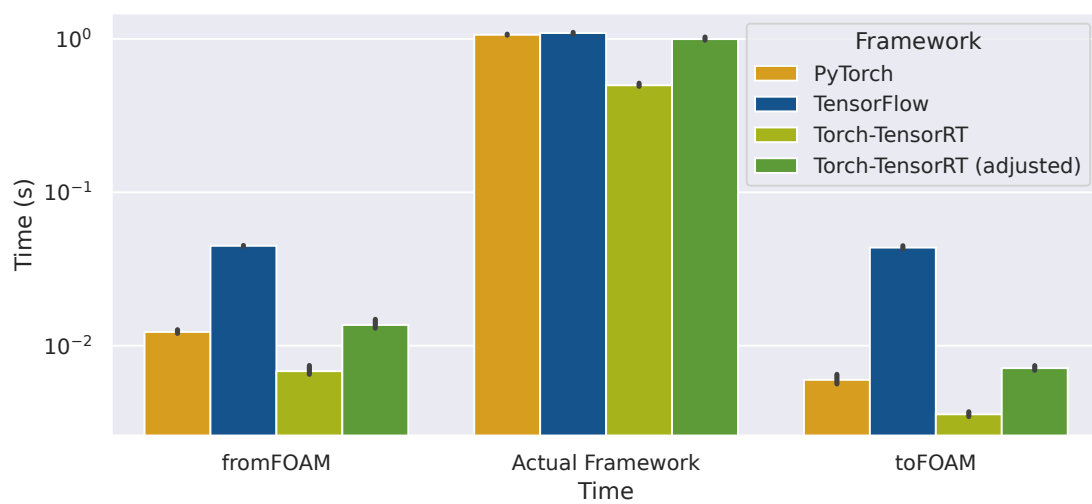


Figure 5.5: Comparison of “fromFOAM”, “Actual Framework”, and “toFOAM” times per framework. Mean of 5 time steps, excluding the warm-up time step. The tips indicate the confidence interval of 95%.

### 5.3.2 Homogeneous Scaling with GPUs

Next, we scale up the amount of GPUs “homogeneously”, i.e., we add only GPU nodes with 48 CPU cores and two V100 GPUs each. As always, we do not change the total amount of data (e.g., change the mesh size). This means that each GPU master rank has a workgroup of 24 ranks, as there are no pure CPU nodes from which to distribute more ranks.

We measure the impact on the fromFOAM and toFOAM communication times between OpenFOAM and the couplings as shown in Figure 5.6. Every time we double the amount of cluster nodes and hence workgroups, the amount of time to communicate from and to OpenFOAM is approximately halved, up to 4 nodes after which the improvement slows down. This matches the fact that by partitioning the total amount of data across more workgroups, each workgroup has less data. The toFOAM direction is consistently about twice as fast as the fromFOAM direction with PyTorch and Torch-TensorRT.

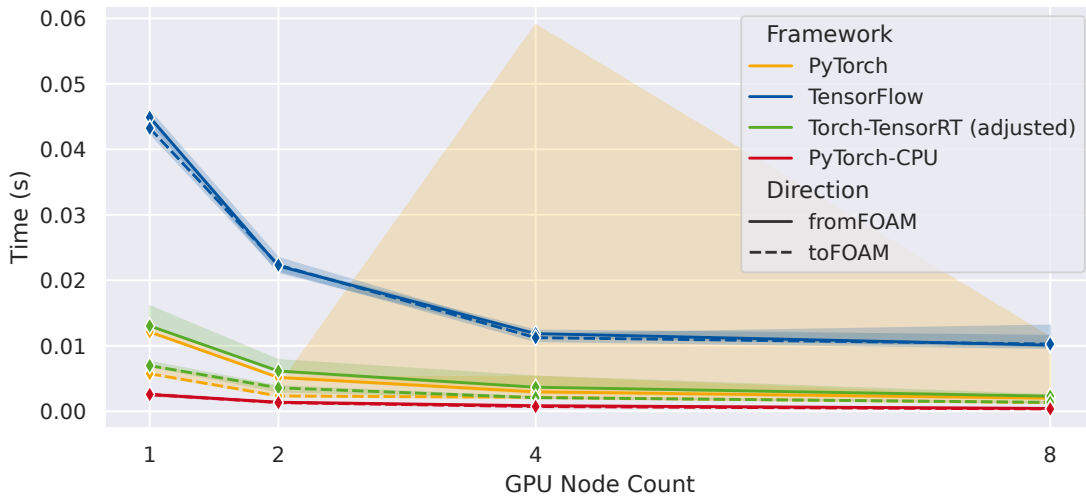


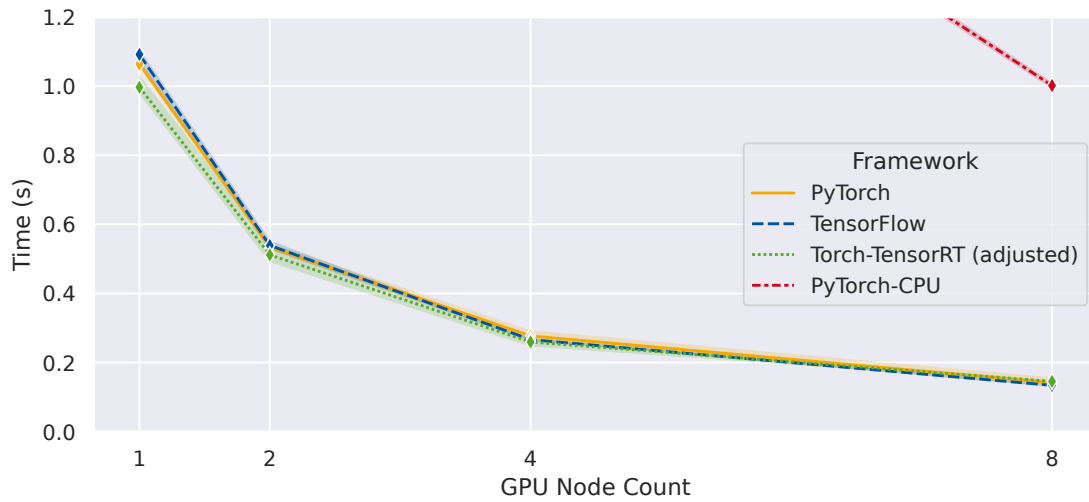
Figure 5.6: Time needed to communicate the OpenFOAM data from individual ranks of the workgroup to a central framework tensor object on the GPU master rank. Workgroups consist of only GPU nodes. Median of 5 time steps, excluding the warm-up time step. The highlighted area around each line signifies the confidence interval of 95%.

Next, we compare the actual framework time for the different couplings (Figure 5.7). The GPU-accelerated frameworks all perform very similar, only the PyTorch-CPU version is much slower by a factor of approximately 8. Similar to the communication times, the actual framework time is also halved as the amount of GPUs is doubled.

## 5 Results



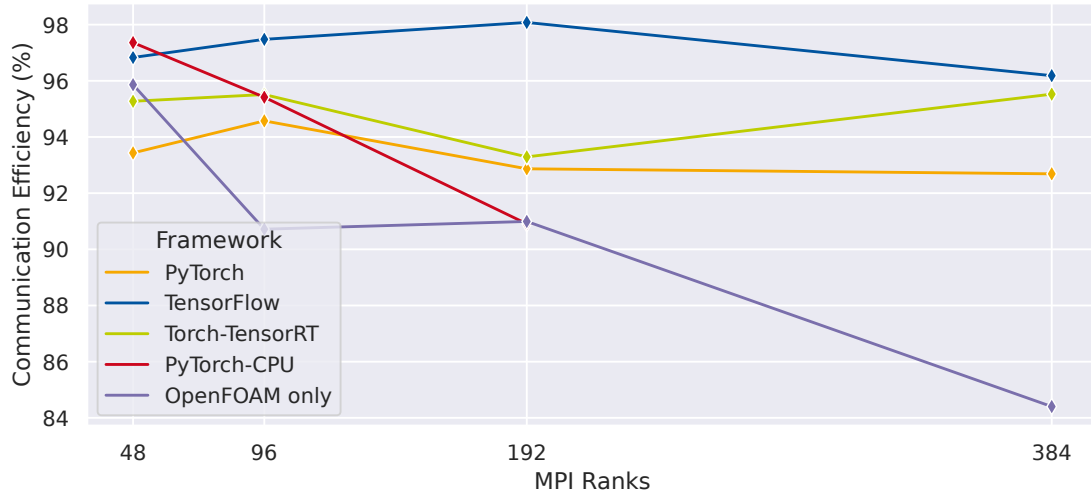
(a) Linear relation between run time and amount of workgroups. The CPU version is much slower than the GPU-accelerated versions.



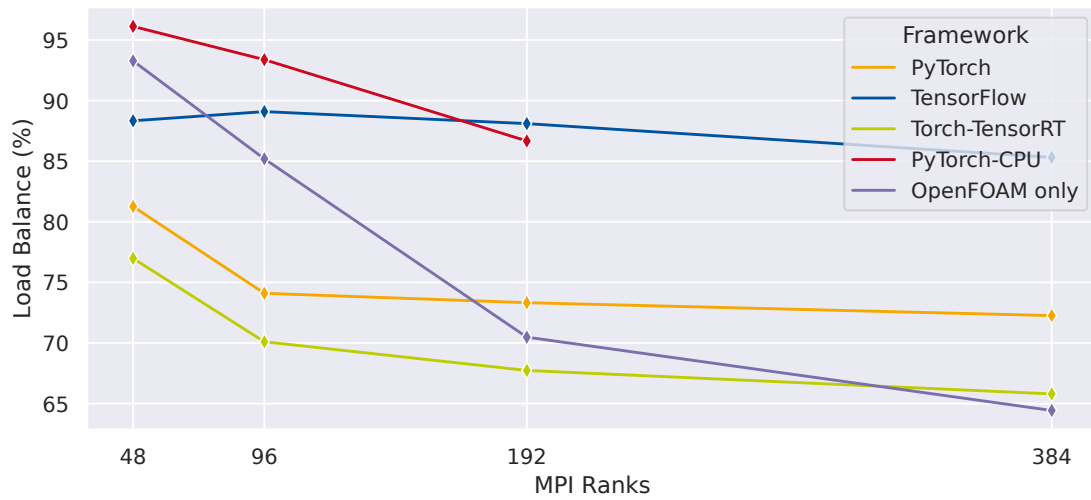
(b) Close-up of 5.7a: All GPU-accelerated frameworks (Torch-TensorRT adjusted for double precision) are equally fast.

Figure 5.7: Actual framework time when scaling up the amount of GPU nodes. The run time is halved by doubling the amount of GPUs. The CPU version given for reference scales similarly since we scale the amount of CPU cores in the same ratio as the GPUs. Mean of 5 time steps, excluding the warm-up time step. The highlighted area around each line signifies the confidence interval of 95%.

We measure POP efficiency metrics using the *Extrac* tool. To have a baseline measurement “OpenFOAM only” to compare our coupling implementations to, we first measured the application as-is but with the coupling removed, i.e., the simulation step implemented by the coupling is simply skipped. The communication efficiency is excellent ( $> 90\%$ ) for all frameworks when scaling homogeneously (Figure 5.8a). The transfer efficiency consistently introduces the smallest performance loss per framework, and almost all loss of communication efficiency is caused by the slightly dropping but still very high serialization efficiency. We note that the OpenFOAM only reference scores worse than our couplings with a downwards trend as we scale up. The load balance is generally better for a lower number of processes, and in case of PyTorch and Torch-TensorRT falls below 80% (Figure 5.8b). Again the OpenFOAM only reference has a strong downwards trend with increasing number of ranks.



(a) Communication Efficiency



(b) Load Balance

Figure 5.8: MPI POP metrics when scaling up GPU nodes homogeneously (2 GPUs per 48 CPU cores) according to POP model as measured by Extrae.

### 5.3.3 Heterogeneous Scaling with GPUs

Now we scale up the amount of cluster nodes “heterogeneously”, i.e., we use one fixed GPU node and only add pure CPU nodes. This means that the workgroup size scales by the same factor as the total amount of nodes: by limiting ourselves to a single GPU node and hence two GPUs, we also fixed the amount of workgroups to two.

We repeat the measurement of fromFOAM and toFOAM communication times for this setting and visualize the results in Figure 5.9. Since we have a constant amount of GPUs and workgroups in this setting, the amount of data per workgroup is constant but communicated over more ranks as the workgroup’s size scales up. Compared to the homogeneous scaling version, our measured times stay approximately constant as we scale up. As before, the toFOAM direction is faster than the fromFOAM direction with PyTorch and Torch-TensorRT, however for Torch-TensorRT both take longer with larger workgroups, while PyTorch’s fromFOAM stays constant and only toFOAM slows down. Both of these frameworks are also slow on a single node, and PyTorch is even slower when running on two nodes, before both reach a local minimum at 4 nodes. TensorFlow is generally slower due to additional steps necessary to create its tensor objects. It is particularly slow at 4 cluster nodes.

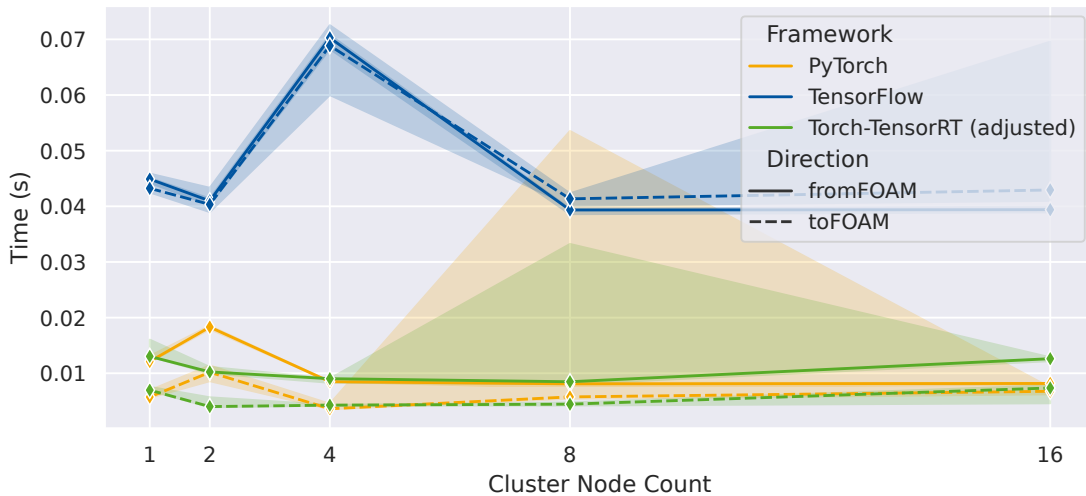
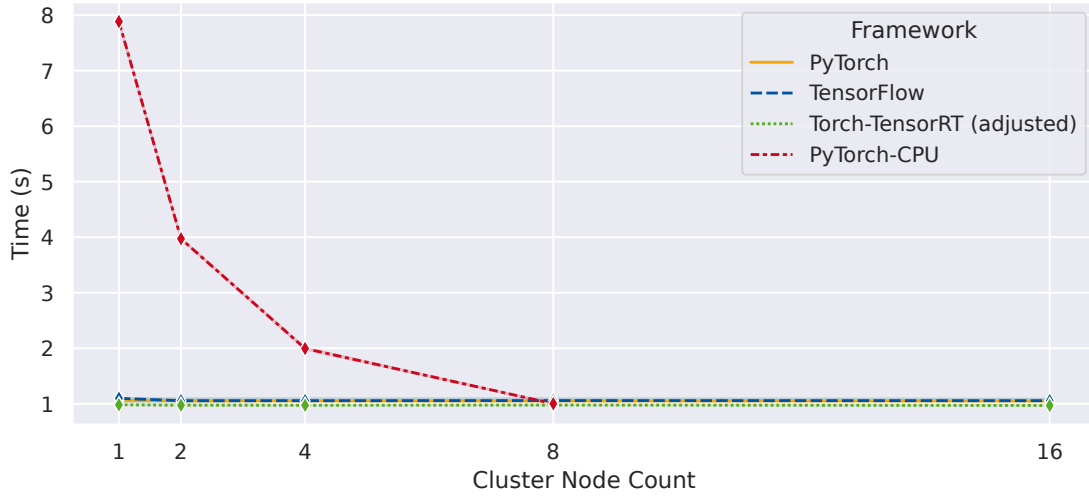


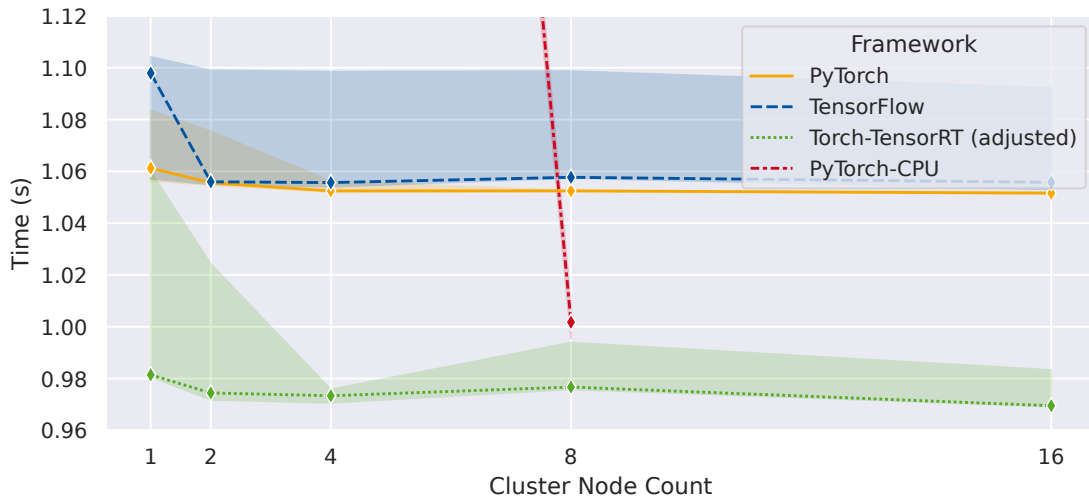
Figure 5.9: Time needed to communicate the OpenFOAM data between individual ranks of the workgroup and a central framework tensor object on the GPU master rank. Workgroups consist of 1 GPU node and otherwise only CPU nodes. Median of 5 time steps, excluding the warm-up time step. The highlighted area around each line signifies the confidence interval of 95%.

The actual framework time on the GPUs does not scale because we have a fixed amount of GPUs, however the PyTorch-CPU version does scale linearly as we add more CPU nodes and catches up to the heterogeneous run time of approximately one second with a configuration of eight cluster nodes, seven of which are pure CPU nodes (Figure 5.10a). As expected, the GPU frameworks' run time is constant regardless of the amount of CPU nodes (Figure 5.10b). The Torch-TensorRT run time which we interpolated from the single precision float run time to double precision is shown as slightly lower than the rest, however the difference is small enough to attribute this to inaccuracy due to said interpolation.

We also measure the POP efficiency metrics for heterogeneous scaling. Figure 5.11a shows a high ( $> 85\%$ ) communication efficiency for all couplings that falls as the workgroups grow. However, the OpenFOAM only reference's efficiency falls much faster. The load balance in this setting drops below 80% as we scale workgroup sizes over 48 (two total nodes), again following the trend of the OpenFOAM reference. In case of PyTorch and Torch-TensorRT we see the load balance improve again over 192 ranks per workgroup (eight total nodes).

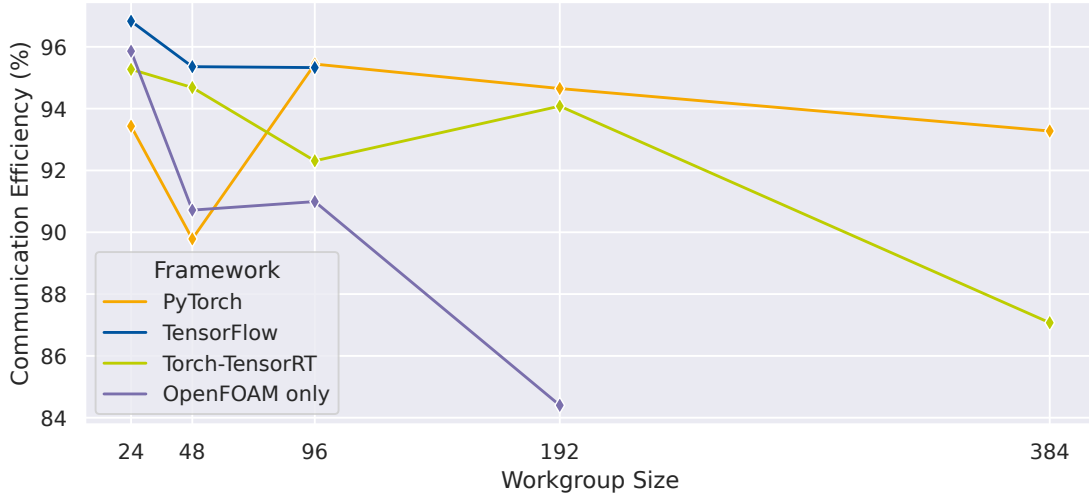


(a) The GPU run times are constant at approximately 1 as we do not add more GPUs when scaling heterogeneously. The CPU version scales linearly with the amount of nodes and reaches the performance threshold of the heterogeneous workgroups at 8 cluster nodes.

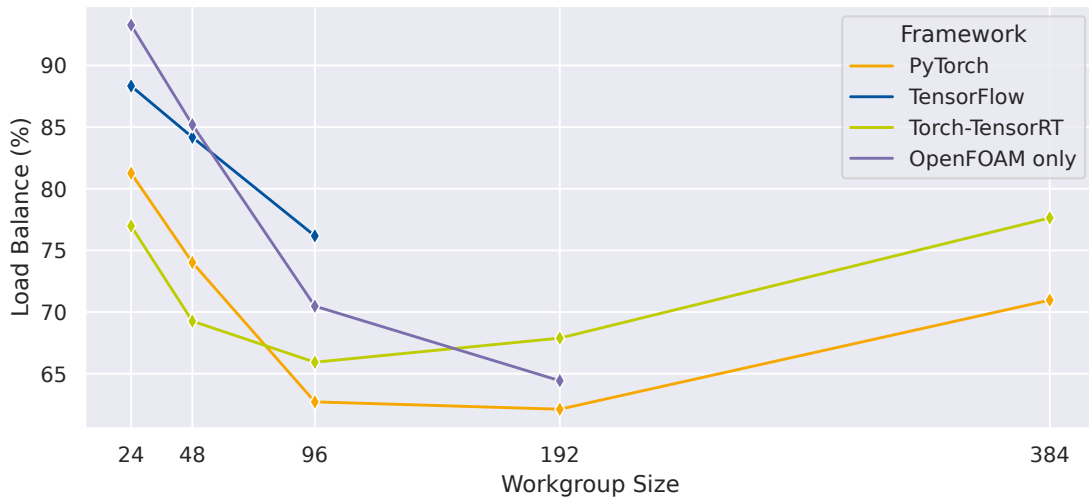


(b) Close-up of 5.10a: Run time of the GPU frameworks is approximately constant. Torch-TensorRT is depicted as slightly faster, which might be an artifact due to interpolation based on the single precision run time.

Figure 5.10: Actual framework time when scaling up the amount of CPU nodes. Mean of 5 time steps, excluding the warm-up time step. The highlighted area around each line signifies the confidence interval of 95%.



(a) Communication Efficiency



(b) Load Balance

Figure 5.11: MPI POP metrics when scaling up CPU nodes (48 CPU cores per node) with a fixed amount of 2 GPUs according to POP model as measured by Extrae.

## 5.4 Measurements with VE-accelerated SOL

### 5.4.1 VE Baseline Measurements

Because the VEs are a different type of device and use a different API, we have to repeat the ground truth experiments. In contrast to the GPU execution, we find that the VEs need no warm-up at all. Figure 5.12 depicts the difference of the actual framework time per time step to the median of that series, relative to the median. All measurements stay within a 1% margin from 0, only the series with batch size  $10^6$  has some outliers.

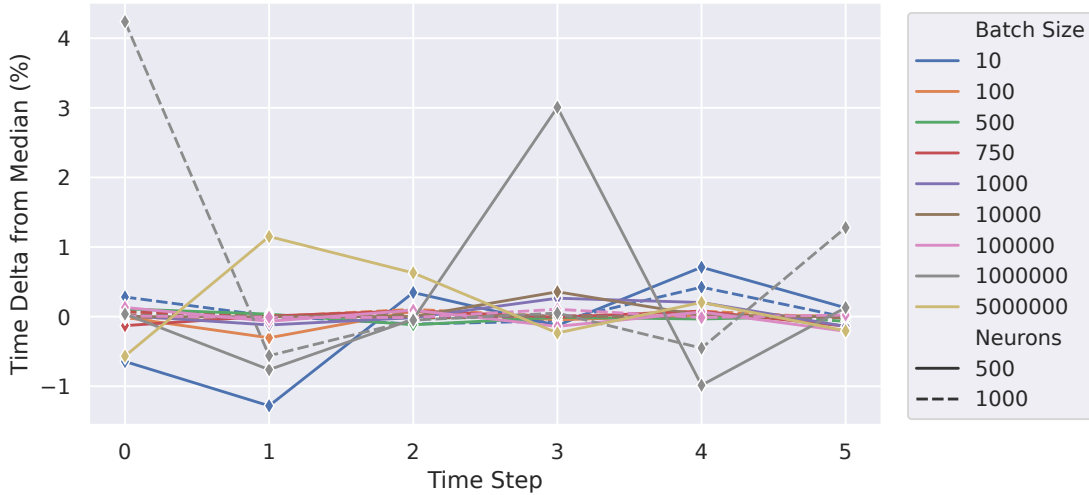


Figure 5.12: Relative difference of the actual framework time by sample (horizontal axis) to the median by batch size (color) and layer size (line style) when using a single VE.

We vary both the batch size and the amount of neurons per hidden layer and depict the results in Figure 5.13. Similar to the measured GPU performance (Figure 5.3), batch sizes of  $10^2$  and lower slow down the inference significantly. However in contrast to the indicated optimal batch size of at least  $10^4$  for GPUs, the sweet spot for the batch size on VEs at an equal layer size of 1000 neurons is in the area of  $5 \times 10^2$  to  $10^3$ . While for the small hidden layer size of 500 neurons the performance stays optimal up to batch size  $10^5$  or  $10^6$  and only explodes at  $5 \times 10^6$ , the model with larger hidden layers of 1000 neurons performs worse the larger the batch is beyond  $10^3$ . Close to the performance sweet spot at the minimum around  $10^3$ , doubling the Neurons from 500 to 1000 has a similar effect as on the GPU (Figure 5.4) and increases run time approximately by a factor of 3.1 (Table 5.6).

As in the GPU framework couplings, the actual framework time outweighs the communication overhead of the coupling in either direction by two orders of magnitude (Figure 5.14). The communication is slower than the mean of the GPU times,

## 5 Results

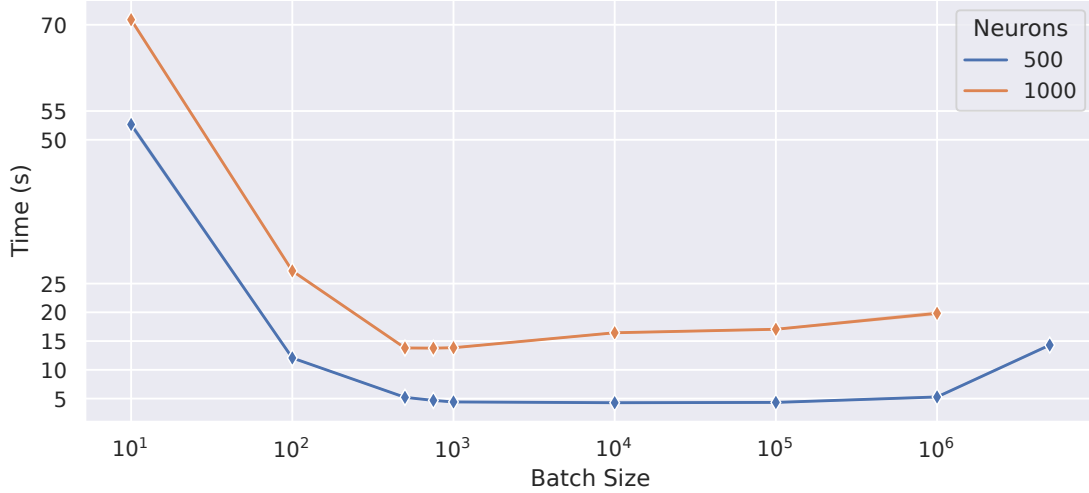


Figure 5.13: Actual framework time by batch size (horizontal axis) and layer size (color) when using a single VE. Mean of 6 time steps.

| Framework      | 500 Neurons   | 1000 Neurons | Factor |
|----------------|---------------|--------------|--------|
| PyTorch        | 0.381 190 8 s | 1.064 278 s  | 2.79   |
| TensorFlow     | 0.413 792 2 s | 1.091 866 s  | 2.64   |
| Torch-TensorRT | 0.190 473 s   | 0.498 529 s  | 2.62   |
| PyTorch-CPU    | 3.010 015 s   | 7.884 98 s   | 2.62   |
| SOL            | 4.417 234 s   | 13.844 s     | 3.13   |

Table 5.6: Actual framework time and scaling factor when increasing the hidden layer size from 500 to 1000 neurons per framework.

but not slower than TensorFlow, while the actual framework time is slower by factor seven as we have already seen earlier.

We inspect the kernel run times using `ftrace`. Running a model with 1000 neurons per hidden layer and batch size 1000 for six time steps, we obtain the results given in Table 5.7. We confirm the operations named `blas` are GEMM operations by inspecting the generated intermediate C++ code. Using Nvidia Nsight Systems, we could see that the GEMM between the two hidden layers takes approximately 70% of the run time, and the same is true for the SOL coupling. All of these operations with a significant duration have a very high ratio of vectorized operations, making use of the special hardware architecture. Still, the average vector length for the big GEMM only reaches 122.5 instead of a theoretical maximal 256 despite using this sweet spot configuration.

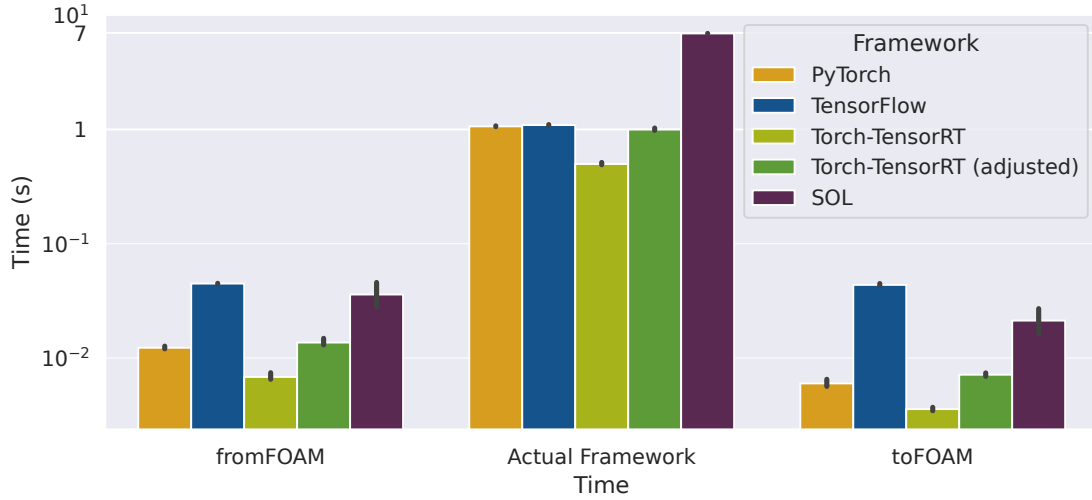


Figure 5.14: Comparison of “fromFOAM”, “Actual Framework”, and “toFOAM” times per framework. Mean of 5 time steps, excluding the warm-up time step. The tips indicate the confidence interval of 95%.

| Operation     | Duration | Dur. (%) | V. Ops (%) | Avg. V. Len. | V. Time |
|---------------|----------|----------|------------|--------------|---------|
| veblas_65     | 7.026 s  | 67.75%   | 99.13%     | 122.5        | 6.957 s |
| veda_omp_simd | 1.183 s  | 11.41%   | 99.53%     | 256.0        | 1.177 s |
| veda_omp_simd | 1.183 s  | 11.41%   | 99.53%     | 256.0        | 1.178 s |
| veblas_6C     | 0.576 s  | 5.56%    | 99.28%     | 247.8        | 0.554 s |
| veblas_5E     | 0.232 s  | 2.24%    | 95.56%     | 86.0         | 0.089 s |
| Total         | 10.371 s | 100%     |            |              |         |

Table 5.7: `ftrace` results of running the SOL coupling, sorted by duration. Batch size 1000, 1000 neurons. The given duration for each kernel is the sum over all its executions over six time steps. Kernels with insignificant duration ( $< 0.62\%$ ) are left out. All significant kernels have a high vectorization ratio.

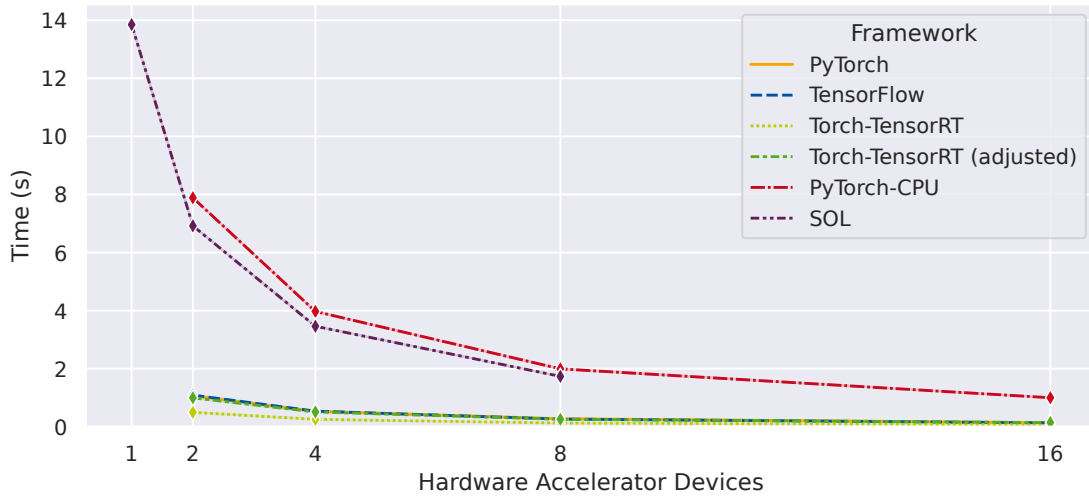
### 5.4.2 Homogeneous Scaling with VEs

We cannot scale VEs the same way we did GPUs, since we have only limited hardware available (Section 5.1) and the VE server is not connected to the batch system for heterogeneous scaling. The only possibility to practically conduct scaling experiments with VEs is to only use a subset of those built into our single server, which is comparable to the homogeneous scaling with GPUs experiment in Section 5.3.2. The scalability of the communication has not changed from our GPU experiments under the assumption that the platforms are equal apart from the difference in accelerator devices, and would not be comparable to each other otherwise.

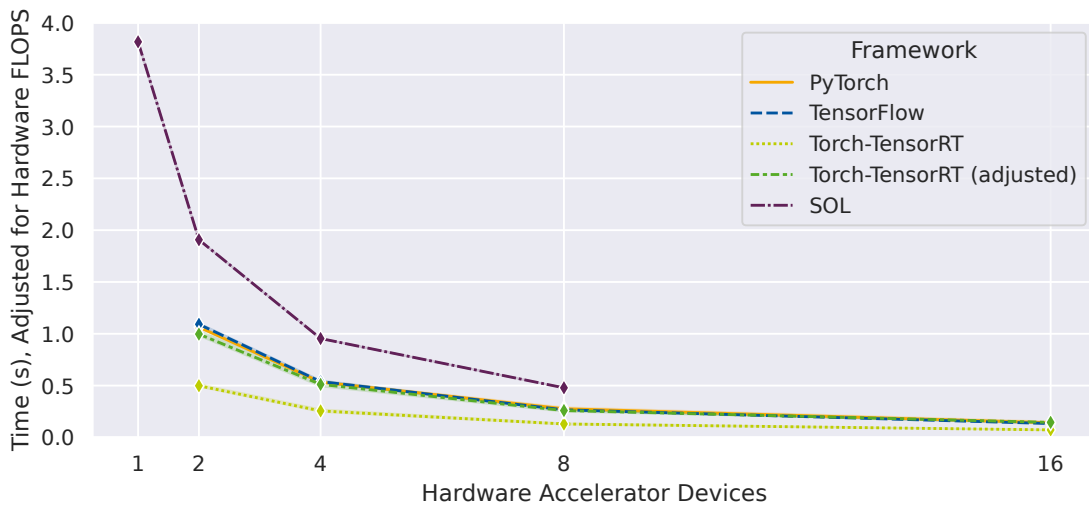
We run the VE scaling experiment by configuring an amount of devices to utilize when running the application and repeat the measurement after doubling the devices until all 8 are used. Figure 5.15a shows the actual framework time with SOL when we add more devices to the same cluster node compared to the CPU and GPU frameworks. We see the run time cut in half each time we double the amount of devices and workgroups, similar to the other frameworks. The SOL framework is slower than the GPU frameworks by a factor of approximately seven, but under the assumption of VE nodes that are equipped similar to the GPU nodes with two VEs per 48 cores still only takes approximately 87.5% as long as the CPU version. Our actual single VE server with 8 devices is 4 times faster than a single CPU node.

The V100 GPUs have a double precision peak performance of 7.8 TFLOPS while the VEs are slower by factor  $\approx 0.28$  with only 2.15 TFLOPS (see Section 5.1). To compare the performance of SOL to the GPU frameworks in another way, we scale the SOL actual framework time by that factor under the assumption that both cards are running at peak performance under the respective optimal circumstances. The result is shown in Figure 5.15b.

The Extrae tool did not work on our VE server, hence we are unable to analyze it with respect to POP metrics.



- (a) The SOL framework running on the VEs is approximately 7 times slower in computing inference than the GPU frameworks. The speedup is linear for all frameworks and devices: doubling the devices halves the run time. PyTorch-CPU in reference to the GPU setup, here two “devices” correspond to 48 cores of the GPU nodes.



- (b) The peak performance of a V100 GPU using double precision (7.8 TFLOPS) is approximately 4 times as high as that of one VE (2.15 TFLOPS). Dividing the measured VE times by this factor under the assumption that both cards run at peak performance yields this comparison. The SOL framework is still twice as slow as the GPU frameworks under this assumption.

Figure 5.15: Actual framework time by amount of accelerator devices (horizontal axis) and framework (color). Mean of 5 time steps, excluding warm-up.



# 6 Evaluation

## 6.1 GPU-Accelerated Frameworks

### 6.1.1 GPU Performance

All of the following interpretations are limited by the fact that our measurements do not include detailed traces of accelerator device internals other than a few simple samples using Nvidia Nsight Systems.

The first result we showed is the warm-up time present with all GPU frameworks and regardless of batch size, while the following time steps all take a constant amount of time. In the traces we gathered using Nvidia Nsight Systems, we could see that the first batch of the first time step is much longer and has an additional call in the CUDA device API that is related to initialization. The logical explanation is that the frameworks only initialize the device fully the first time we run our ML model which then isn't necessary for the following time steps. We could also see that TensorFlow does a different initialization call than PyTorch and Torch-TensorRT, which can explain why it takes consistently longer. This could be avoided by doing a warm-up run of the model during initialization, however it is not so significant once an application runs for more time steps than our very small sample of only six. We also noticed that the initialization of the GPU framework couplings could take very long, and this seemed to be related to the actual framework initialization of the device and underlying device API. There is potential for improvement by doing this part of the (device) initialization asynchronously with the rest of the (host) initialization.

Using Nsight Systems we also saw that the biggest part of the batch processing time is spent in the matrix-matrix multiplication from the first of our large hidden layers to the second. We can explain that easily by the MLP's structure: The amount of (scalar) multiplications of input values and weights for a two layer MLP with  $i$  inputs,  $h$  neurons per hidden layer,  $o$  outputs is calculated by

$$ih + hh + ho = h^2 + h(i + o). \tag{6.1}$$

For our MLP using  $i = o = 2$  and  $h = 500$ , we get

$$\begin{aligned} & 2 \times 500 + 500^2 + 500 \times 2 \\ & = 1000 + 250000 + 1000 \\ & = 2.52 \times 10^5 \end{aligned} \tag{6.2}$$

and when increasing the hidden layers to  $h = 1000$ , we get

$$\begin{aligned}
& 2 \times 1000 + 1000^2 + 1000 \times 2 \\
& = 2000 + 10^6 + 2000 \\
& = 1.004 \times 10^6.
\end{aligned} \tag{6.3}$$

In both cases the in- and output layers cause an insignificant amount of multiplications when compared to the total:

$$f(h)_{i,o} = \frac{h(i+o)}{h^2 + h(i+o)} = \frac{\text{i/o multiplications}}{\text{total multiplications}} \tag{6.4}$$

$$f(500)_{2,2} = \frac{2 \times 10^3}{2.52 \times 10^5} \approx 0.79\% \tag{6.5}$$

$$f(1000)_{2,2} = \frac{4 \times 10^3}{1.004 \times 10^6} \approx 0.4\% \tag{6.6}$$

Looking at an example of kernel times captured using Nvidia Nsight Systems (Table 5.3), we see the big GEMM taking up 67.42% of the time, and the other GEMMs 4.49% + 2.57%  $\approx$  7.06%, which is much more than the workload calculated in Equations 6.5 and 6.6. This is likely caused by performance overhead of these smaller operations due to lower occupancy of the device leading to less than peak performance. We also see the Memcpy operations take only 0.56% of the time, indicating a very small communication overhead between host and device compared to the whole process. This is because the small amount of 800 kB input data is very small but creates a much bigger amount of necessary computations as the layer size blows up from only two input layer neurons to 1000 hidden layer neurons. The remaining non-GEMM or memory operations take up approximately 25% of the kernel times in this case.

We found that the actual framework time depends on the chosen batch size. We can explain this with the hardware architecture of GPUs: GPUs are highly parallel devices with in this case 2560 double precision cores. Hence to achieve peak performance, matrix operations are split up into multiple smaller operations that can be computed in parallel on multiple cores. Our decision to keep the double precision from OpenFOAM prohibits us from using the mixed half and single precision specialized tensor cores that would otherwise be specialized in this operation<sup>1</sup>[22]. Such an operation needs to be sufficiently large in order to reach a threshold where the operation can occupy the whole device and use its peak performance. This threshold appears to be reached with batch sizes of  $10^4$  and larger in Figure 5.3b. In the same series of measurements we were unable to capture the sample for PyTorch at  $10^6$  and the value for TensorFlow rises also, which we can explain with the GPU's memory limit of 16 GB being reached. All GPU frameworks using double precision perform equally fast within these optimal batch sizes.

<sup>1</sup><https://developer-blogs.nvidia.com/wp-content/uploads/2019/01/Tensor-Core-Matrix-625x169.png> via <https://developer.nvidia.com/blog/tensor-cores-mixed-precision-scientific-computing/>

In our experiments, changing the amount of hidden layer neurons from 500 to 1000 increased the actual framework time approximately by a factor 2.7. We have previously seen using Nsight Systems, that the matrix multiplication step to transition from one hidden layer to the next has the biggest run time impact. If we scale  $h$  by  $n$ , then for small  $i, o \ll h$  the amount of operations scales like  $\mathcal{O}(n^2)$ :

$$\begin{aligned} & inh + nhnh + nho \\ &= n^2h^2 + nh(i + o) \\ &\approx_{i,o \ll h} n^2h^2 \end{aligned} \tag{6.7}$$

In our particular case, we scale from  $h = 500$  (Equation 6.2) to  $n \times h = 1000$  (Equation 6.3) neurons (factor  $n = 2$ ) for an increase of multiplications by factor:

$$\frac{1.004 \times 10^6}{2.52 \times 10^5} = 3.98 \approx 2^2 \tag{6.8}$$

The actual factor is lower (better) than the theoretical factor. One reason is that we modeled only the operations of the GEMMs, neglecting the remaining operations that make up 25% of the batch time. We have also seen while the small GEMMs only make up  $< 1\%$  of multiplication operations, in reality they can take 2.5% to 4.5% of the batch run time. Further, the GEMMs are optimized SIMD operations that can compute more than 1 operation per clock cycle.

We measured the time our couplings take for communication between OpenFOAM and framework, and the pure framework time. It is clear that the communication time is outweighed by the framework time by a sufficiently large factor that makes the communication overhead negligible. The TensorFlow coupling performs consistently worse than PyTorch and Torch-TensorRT, which is at least partially caused by additional steps taken to properly communicate its tensors over MPI (Listing 3.11). We lack the necessary additional trace data to determine whether something else within the TensorFlow API slows it down on top.

Homogeneous scaling of our case means running our application on a growing set of GPU nodes. Because each added node has 48 CPU cores and 2 GPU devices, this means the workgroup size stays constant because the ratio of CPU cores per GPU stays constant as each GPU has its own workgroup. At the same time the amount of work, i.e., total amount of cells (5 419 008) and fields per cell (2), stays constant. This setting is called “strong scaling”. With the above results we have learned that we can approximate our couplings’ run time as the actual machine learning time on a single cluster node. Further, we know that this part of the program is completely parallelizable in the sense of partitioning the total set of cells between workgroups. According to Amdahl’s law [11], strong scaling of a 100% parallelizable program will speed up execution by the same factor of which processors are added.

This is indeed the kind of perfect speedup we see in our couplings’ actual framework times (Figure 5.7). Remarkably, all GPU frameworks are equally fast at all sampling points. This is likely because they all implement the same or equally fast optimizations and use the hardware well. The CPU version, while scaling

equally well, is consistently slower than the GPU version by factor 8. Considering the ratio of peak GPU performance to peak CPU performance on a single node is  $\frac{15.6 \text{ TFLOPS}}{3.2 \text{ TFLOPS}} = 4.875$ , the CPU version is underperforming in comparison. In particular, Nvidia’s Torch-TensorRT optimization is not significantly faster than plain PyTorch despite the used GPUs being Nvidia hardware. Similarly the couplings’ communication (Figure 5.6) also has perfect speedup for up to four nodes. We have a total amount of  $5\,419\,008 \text{ cells} \times 2 \frac{\text{fields}}{\text{cell}} \times 8 \frac{\text{B}}{\text{field}} = 86\,704\,128 \text{ B} \approx 82.69 \text{ MiB}$  of input/output data. This data is partitioned evenly across four nodes ( $4 \text{ nodes} \times 2 \frac{\text{devices}}{\text{node}} = 8$  devices or workgroups) for  $82.69 \text{ MiB}/8 \text{ workgroups} = 10.34 \text{ MiB}$  per workgroup. Each workgroup has 24 MPI ranks, yielding  $10.34 \text{ MiB}/24 \text{ ranks} = 441.17 \text{ KiB}$  per rank. For data transfers of shrinking size, overhead times by communication initialization, link latency, and similar effects becomes ever more significant, however remains negligible in comparison to the actual framework time. We do not have further data to back that this is the sole cause for the slowed scaling of our communication, but this effect will become visible for this point in our application at some point.

Heterogeneous scaling of our case means running our application on a fixed amount of GPU nodes (one) and a growing set of additional CPU nodes. The amount of GPUs and hence workgroups is fixed, thus adding more CPU cores increases the size of the workgroups. The constant amount of work partitioned over the two workgroups means that the work per group is also constant. We do not scale the framework part of our coupling, but instead only the amount of ranks per data in our communication part.

The measured results in Figure 5.9 show mostly constant time as more nodes are added, but significantly more variance than for the similar measurement series with homogeneous scaling. A possible cause of this effect is the network architecture of the RWTH CLAIX cluster, where nodes from different cluster partitions that are also physically partitioned now have to communicate where we only used a single partition before. We were unable to run our measurements on a fixed set of nodes that fulfills certain topology constraints. The peak values of the median at two nodes for PyTorch and four nodes for TensorFlow are possibly caused by combinations of cluster nodes that are connected with worse links than those used for the other samples. There is a small drop in communication time from one to two nodes that could be explained by the amount of data being split into smaller partitions due to increase in the workgroup size and communicated asynchronously, but the improvement is not as clear as in the homogeneous setting. We can also see a trend of the communication time rising as we reach 16 cluster nodes. Similar to the homogeneous case, there are only  $220.5 \text{ KiB}$  per rank at this point, and the communication overhead becomes too significant and causes a slowdown.

The actual framework time of the GPU frameworks is constant as expected, since we don’t scale the amount of GPUs. We can see only the variant of PyTorch running on CPU scale perfectly as we still increase the amount of CPU cores. The CPU version catches up to the GPU versions at 8 cluster nodes, which is the same as the factor by which the CPU version is slower than the GPU version on a single

node as determined earlier. As long as there is enough work for this perfect scaling to continue without too much overhead, it would be faster to run the CPU version instead of a GPU version if only a single GPU node was available. In general this threshold depends on factors such as the node configurations (how many GPUs per CPU cores) and their available amounts, the amount of data, the complexity of the model, and the ratio between amount of work in the coupling to amount of work outside the coupling. In our example we have already seen the limits of the communication overhead with MPI due to our small amount of data. In comparison to deep learning models, our MLP is simple and uses mostly simple operations such as GEMM. The OpenFOAM part takes significantly longer than our coupling on a single GPU node. This example case would profit most from using more CPUs to speed up the OpenFOAM part, and hence possible to run the machine learning inference on CPUs quickly instead of introducing communication overhead and idle time by using GPUs for inference.

### 6.1.2 POP Efficiency

The shown POP metrics limited in some regards due to time constraints of this thesis. First, the measurements were taken of the whole application, which means they include the initialization part of the application and in particular that of the framework and device. This initialization step can take 30 seconds which is a significant part of the run time when the whole application runs only for two minutes. Further, it means that it includes both the coupling and the OpenFOAM part of the simulation, and since the OpenFOAM part takes much longer than the coupling part, the conclusions we can draw about our coupling are limited. Second, we were able to measure the metrics only for MPI, but not CUDA. This limits our possibility to interpret the framework part and apply the hybrid model presented in Section 4.2. Third, within our time constraints we were only able to measure these metrics once, so the data does not have the statistical significance of repeated measurements.

For homogeneous scaling (Figure 5.8), we measure very good communication efficiency for all of our couplings. Notably, the OpenFOAM only reference shows a downwards trend with increasing amount of MPI ranks in its transfer efficiency that is reflected by the communication efficiency, however the measurements of the couplings do not reflect this. We would expect a lowered serialization efficiency for small amounts of GPUs (and hence MPI ranks), because the coupling time is mostly dependent on the actual machine learning time, during which the non-GPU ranks need to wait. As more GPUs are added, the coupling is faster overall and there is less waiting. However we can not see such a trend in the data, as the measured values for the GPU couplings are approximately constant.

Similarly, we see a downwards trend in load balance in the OpenFOAM only reference, that is not strongly reflected by the couplings which again have approximately constant load balances as we scale up. As with the communication efficiency, we would expect an upwards trend as the waiting time decreases.

In both cases, the time spent in the coupling decreases as we scale up, hence these

overall metrics are impacted by the coupling less and less. However we can neither see the expected trends nor a trend of the couplings' following the reference when using more nodes for either communication efficiency nor load balance.

For heterogeneous scaling (Figure 5.11), we would expect a worse communication efficiency due to serialization problems as workgroups grow, because more ranks wait on a constantly slow GPU. Indeed loss of communication efficiency is caused by loss of serialization efficiency when it happens in the measurements with GPU couplings, but we can not see a clear trend, nor strong correlation to the OpenFOAM only reference.

For the same reason, we expect to see a load imbalance as workgroups grow. The load balance does indeed drop at first, following the trend of the OpenFOAM reference, but then rises again in case of the PyTorch and Torch-TensorRT couplings.

Inspecting the raw data, we can compare the idea runtime to the maximum useful duration (which likely corresponds to a GPU master rank). There is always a difference of multiple seconds, which is far more than the order of magnitude we measured for our couplings' communication overhead ( $10^{-2}$  seconds). This means the communication inside the OpenFOAM part is overshadowing the influence of our couplings by far.

Overall, the results of our POP metric measurements are not very conclusive.

## 6.2 VE-Accelerated Framework

We examine the SOL framework running on the VEs with regard to the same aspects as the GPU frameworks. The measurements of multiple time steps show that contrary to the GPUs, the VEs do not take additional warm-up time in the first time step, but instead perform equally fast every time for a fixed batch size and amount of neurons. A possible explanation for this different behavior is based on the fact that the deployed SOL model is very static. Not just the model structure (e.g., amount of neurons) is fixed, but additionally the batch size is fixed during deployment. This is different from even the Torch-TensorRT version where we can define the batch size at the time of executing the model, despite being compiled ahead of time as well. This could explain the need for the warm-up time with GPU frameworks and at the same time why it is missing on the VE. It is possible that the GPU frameworks have the ability to reuse more resources between batches and time steps which the SOL framework can not. In this case the initialization overhead should become visible as we scale the amount of work per device, but we see no such evidence in our scaling experiment. We conclude it is likely that the GPUs are doing an initialization step that is slow and possibly not shown and more complex than the `cudaMalloc` shown by Nsight Systems, while memory allocation is fast on the VEs.

As we inspect the device kernels using `ftrace`, we find no evidence of warm-up time and the `internalMalloc` is so fast that it has only negligible impact (0.46%). Similar to the GPU versions, the GEMM between the large hidden layers takes

the biggest part of the run time, and the smaller GEMMs take a larger part than estimated by Equations 6.5 and 6.6. In general the results of this trace mirror those of the Nsight Systems trace.

We measured the influence of both hidden layer size and batch size on the actual framework time. When we use an optimal batch size for each hidden layer size, the actual framework time scales by a similar factor as it does when using GPUs. That means VEs, too, perform better than the theoretical increase in complexity of the model (Equation 6.7). At the same time, the batch sizes that achieve the lowest run times are differently from the GPUs. While we have seen the GPUs run faster with increased batch size until exceeding the memory limit, the VEs slow down again as batches become too big despite a larger amount of memory available on the VEs than on the GPUs. The optimal batch size also depends on the amount of neurons, as the minimum run time is achieved with smaller batches on a larger model and larger batches on a smaller model. This is likely a caching effect. As the matrices are small enough to fit into the cache completely, the operation can profit from the much faster cache instead of loading from slower memory thanks to a high cache hit rate. Bigger matrices exceed the cache size and must be loaded from memory into cache multiple times, hence slowing down the operation.

Comparing the coupling’s communication times to the actual framework time, we see that again the communication is two orders of magnitude smaller and thus negligible in comparison. We also notice that the SOL communication time is a bit slower than most of the GPU frameworks. As the SOL coupling uses a simple C array for this step and no additional processing as the TensorFlow coupling, the only explanation is the different platform.

We vary the amount of VE accelerator devices to conduct a scaling experiment that is similar to the homogeneous scaling with GPUs, however we are limited to a single node as this server is not connected to the cluster in the same way the GPU nodes are. By the same reasoning as homogeneous GPU scaling, we expect and observe a perfect speedup when using SOL. Compared to the other coupling versions, the single VE server with 8 cards outperforms a single node running the CPU coupling by a factor of four, but is still twice as slow as the GPU frameworks on a single GPU node. Comparing a single VE’s run time and peak performance (14 s, 2.15 TFLOPS) to the CPUs on a single node (8 s, 3.2 TFLOPS), the VE is slower by factor  $\frac{14\text{s}}{8\text{s}} = 1.75$  despite a slower peak performance of only factor  $\frac{3.2\text{TFLOPS}}{2.15\text{TFLOPS}} \approx 1.49$ . This means the VEs are underperforming in comparison.

When we compare the same amount of devices, a single GPU takes only  $\frac{1}{7} \approx 0.14$  the amount of time of a VE, despite a difference in theoretical peak performance by factor  $\frac{2.15\text{TFLOPS}}{7.8\text{TFLOPS}} \approx 0.28$ . In other words, assuming the same peak performance, running SOL on a VE is still twice as slow as the GPU frameworks. This means that either the VE is performing additional work, or the GPUs are closer to achieving their peak performance than the VEs. More detailed knowledge about this could be gained by doing a more in depth analysis of the devices at runtime.



# 7 Conclusions

In this thesis we examined a HPC application that was modified by replacing one component of the computation with a machine learning model. Machine learning has become more feasible recently thanks to the availability of specialized hardware accelerator devices that are particularly suited to compute machine learning algorithms and is already established in some fields such as computer vision and voice recognition. The connection between the original program and an ML framework is achieved by wrapper code which we call “coupling”, and the resulting hybrid application a coupled HPC+ML application.

A reactive thermo-fluid simulation simulates both movement of a mixture of multiple substances as well as a combustion reaction of said substances. Our simplified test case is implemented with the computational fluid dynamics software framework OpenFOAM and uses an approach called flamelet generated manifold for the combustion simulation. The FGM is derived by interpolation of a set of representative flamelets, representing certain chemical relations, that are precomputed and stored in a lookup table at run time. As the amount of variables influencing the flamelets and hence dimensions of the LUT are large, it takes a great amount of memory to store the LUT. Machine learning models are self-adjusting algorithms that can be trained to capture relationships that are hard to capture as algorithm by humans by providing the correct output for some input. Despite the large amount of coefficients in such a model, it can still have a much lower memory footprint than an equivalent LUT, hence the idea of our HPC+ML application is to replace the LUT with a relatively simple machine learning model called a multilayer perceptron.

There are some well known ML frameworks, such as PyTorch and TensorFlow, that have evolved over some years now, and also constantly new approaches including client-server architectures (Nvidia Triton), hardware specific optimizers (Nvidia Torch-TensorRT), and new hardware (vector engines). We implemented equivalent couplings for these five named frameworks in order to compare their performance.

Timings for the communication and the processing part of the coupling were captured by instrumenting the application to collect timestamps at the respective parts of itself. Two important parameters that influence the network’s performance are the amount of neurons we choose for our MLP instance and the amount of data (“batch”) we submit to the network for processing in one go. We can tweak the batch size parameter to find a hardware architecture dependent sweet spot at which the framework achieves the best performance for a certain hidden layer size. The of input data and size of the layer-to-layer transitions in the model directly influence the size of the matrix multiplications that implement the MLP. These matrix multiplications can be broken down into smaller partial matrix multiplications. They need to be

## 7 Conclusions

large enough to in the best case occupy all processing units of the device, but small enough to still fit any intermediate results into memory. The GPU frameworks achieve the highest speedup with the highest possible batch sizes, but the VEs have a different hardware architecture that is less efficient calculating with matrices that are too large.

Over half the time of each model execution is spent in the big matrix multiplication that implements the transition between the two large hidden layers of the MLP. As we increase the size of these layers by  $n$ , we scale up the theoretical amount of multiplications necessary by  $\mathcal{O}(n^2)$ , however we measure the runtime increase by a much smaller factor thanks to the efficient implementation of the matrix multiplication operations.

Comparing the measured communication overhead in the coupling introduced by translating the data from OpenFOAM to the respective framework and back to the actual framework run time, we see this overhead is two orders of magnitude smaller than the useful computation time and thus negligible.

We examine the coupling performance with regard to strong scaling.

The run time spent in the coupling part of the application consists almost completely of the machine learning inference running on the accelerator devices. Because there are no dependencies between the devices, the parallel part of our coupling is approximately 100%, and we see a perfect speedup according to Amdahl's law as we add more accelerator devices. All GPU frameworks are equally fast, using the same amount of VEs is slower by factor 7, and using the same amount of CPUs (24 cores each) is slower by factor 8 than using GPUs. If we take into account the theoretical peak performance of these device types, the GPUs perform closest to their peak performance by far, followed by the CPUs, while the VEs are the furthest from reaching it. We can see a similar scaling in the communication overhead as we partition the fixed total amount of data over ever more workgroups, reducing the amount that is communicated within the workgroup, until we reach the latency limit of the connection.

If instead we fix the amount of devices and increase the workgroup size by adding more CPUs, the machine learning time predictably only improves in the CPU version of the coupling, undercutting the time needed by the GPU frameworks as we reach the scaling factor 8 we measured earlier. We remark that while the GPUs perform the machine learning task very fast, depending on factors including hardware configuration and ratio of coupling to total run time, it can be more efficient to run the machine learning on the CPUs than introducing idle time and communication overhead by using GPUs. Again the coupling's communication time is unaffected until we split the amount of data over so many ranks that the overhead of sending such small amounts of data becomes significant enough to have a measurable negative impact.

The POP metrics we captured are limited and we can recognize neither a correlation to the reference captured without any ML coupling, nor a the trends we expect due to the architecture of the coupling. A large contribution to this inconclusiveness is the fact that our measurements include the initialization phase of application and

coupling, which makes up a significant part of the total run time, and also include the time spent outside the coupling, which has a much greater run time by two orders of magnitude and thus has a much greater impact on the metrics than the couplings.



## 8 Future Work

Some aspects we planned to examine for this thesis had to be left out due to issues with frameworks and tools that could not be solved within our time constraints.

To interpret the POP metrics conclusively, a more in depth analysis is required. This includes capturing the different parts of the application independently from each other, such as the initialization step and the coupling part. We presented a POP model for hybrid applications, that could be applied when the tracing tool is able to capture the CUDA runtime as well as the MPI runtime.

We inferred some indication of the occupancy of the accelerator devices based on theoretical limits. An analysis of the kernels in detail with regard to the hardware structure could indicate better how to adapt the algorithm for more efficient processing. To increase the efficiency with which devices are used, HPC clusters could offer a central service that adjusts automatically to requested workloads with techniques such as multi instance GPU and automated batching. While we studied some of the involved parameters, a deeper understanding is required to be able to model a heuristic for such an automatism.

We explored the idea of different MPI workgroup assignments and architectures, but were unable to implement any within the time constraints of this thesis. We already mentioned a possibility to take rank locality into account when assigning workgroups in Section 2.2. Another possibility would be an MPI+OpenMP hybrid approach that spawns only one MPI process per cluster node and utilizes the whole CPU by multithreading with OpenMP. Then MPI operations would be limited to far fewer participants through multi-tiered communication. However to implement this approach in our case, it would need to be adapted in the OpenFOAM code while the coupling could potentially remain largely unmodified.

The examined frameworks are generally able to access multiple GPUs directly. This possibly has performance implications to the inference itself, but at the same time reduces the amount of workgroups needed and thus changes the MPI behavior as well.

Because of the high level HTTP abstraction that Nvidia Triton uses for communication between client and servers, it is particularly flexible with regards to the communication architecture. Our proposed implementation (Section 3.5) uses a two-level communication scheme that mirrors the other couplings: First the data is gathered centrally using MPI and then it is sent to the Triton server over HTTP. One obvious possibility is to instead implement the coupling in such a way that each rank individually communicates its data directly to the server, removing the intermediate MPI communication as depicted in Figure 8.1.

The use of HTTP for communication offers another opportunity. Load balancers

for HTTP traffic are a standard practice for large scale cloud services. By inserting a central load balancer in the communication between clients and servers, the work can be evenly distributed across all Triton servers in case the workgroups generate different workloads. This approach is visualized in Figure 8.2.

TensorFlow also has some capability for distributed processing<sup>1</sup> that can be examined with similar communication architectures.

In our case there is a dependency between all ranks in the OpenFOAM part of the application, synchronizing the execution of the ML in the coupling despite the workgroups running the couplings being independent from each other. If this were not the case, and the different workgroups run asynchronously, they could share a smaller amount of GPUs more effectively by “taking turns”.

We have pointed out that it is not trivial to create the exact same ML network in different frameworks. The correctness of different implementations is of course important and should be verified before using this approach for a real application.

Additionally, the ML approach should deliver results with an accuracy that is comparable to the results of the reference implementation using a LUT. Hardware accelerator devices are faster when using lower precision floating point data types, and in particular modern GPUs offer tensor cores that can perform the large tensor operations in our ML model much faster at the cost of floating point precision. In our current application structure, the machine learning step is executed in alternation with the rest of the algorithm, leaving one processing device idle as the other runs. For circumstances where both parts take approximately equal amounts of time, the result of one ML inference could be approximated by its input. This way both parts can run simultaneously while losing a small amount of accuracy due to the approximation. The resulting accuracy of implementing either approach remains to be researched.

---

<sup>1</sup><https://www.tensorflow.org/tutorials/distribute/input>

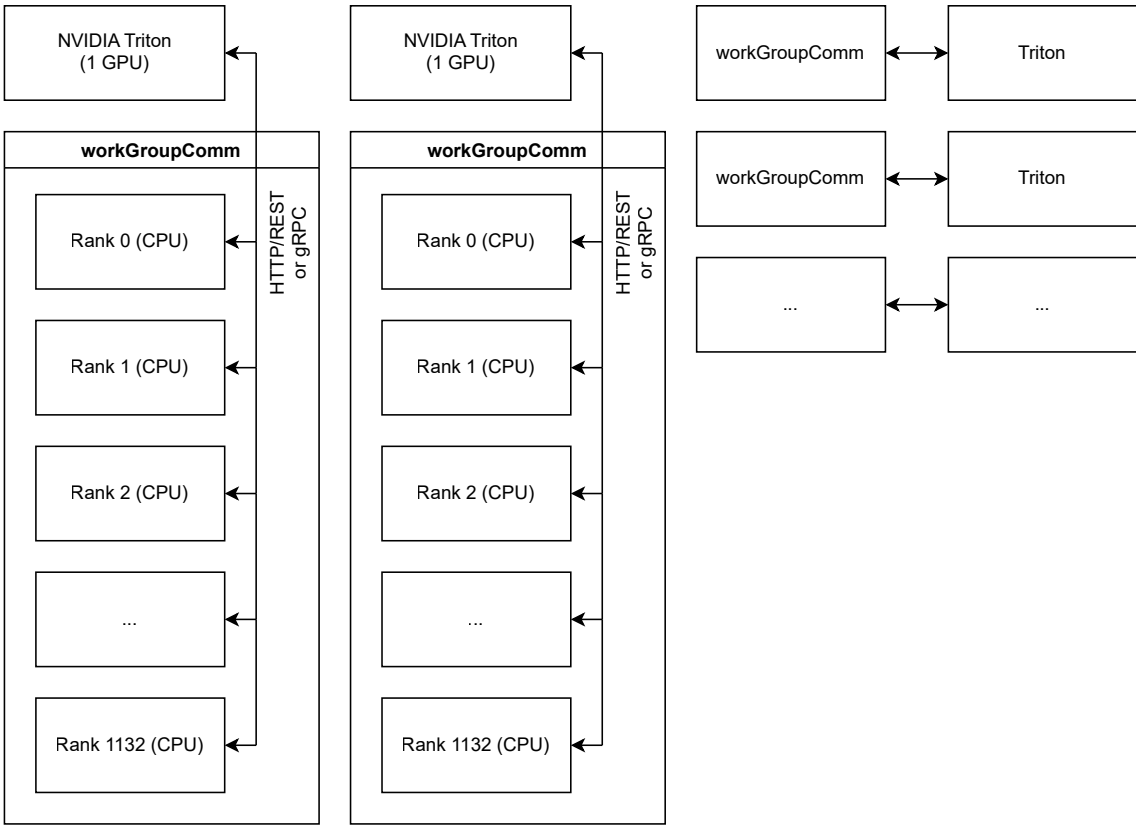


Figure 8.1: The need for MPI communication is eliminated due to Triton’s HTTP interface.

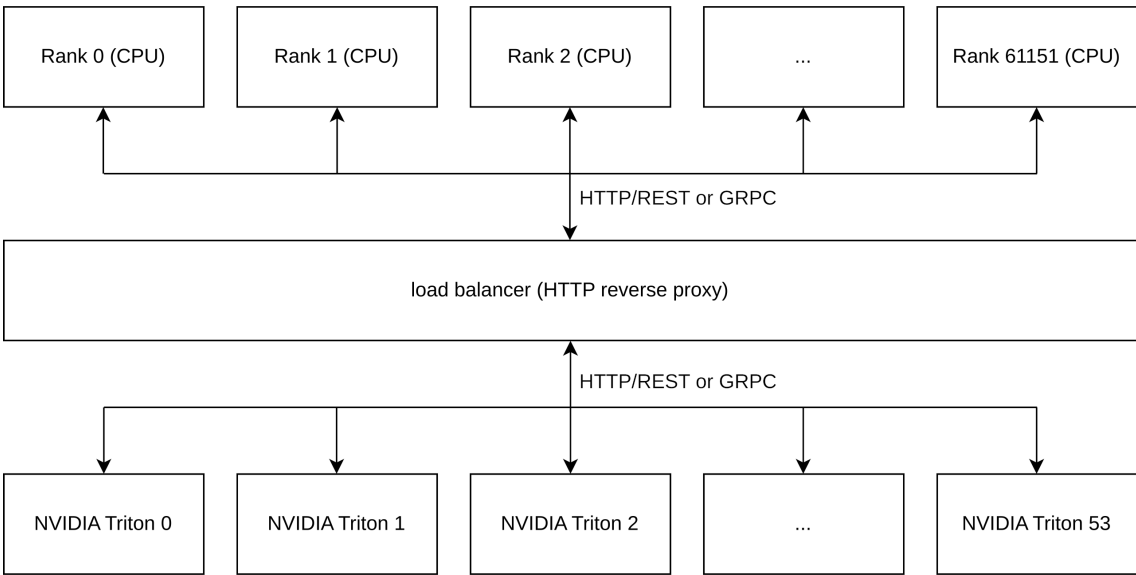


Figure 8.2: A load balancer between Triton clients and servers may balance the uneven workloads across different workgroups.



# Acknowledgements

Simulations were performed with computing resources granted by RWTH Aachen University under project thes1099.



# Bibliography

- [1] cuBLAS. <https://developer.nvidia.com/cublas>. Accessed: 2022-05-13.
- [2] Exrae. <https://tools.bsc.es/exrae>. Accessed: 2022-05-09.
- [3] Nvidia TensorRT. <https://developer.nvidia.com/tensorrt>. Accessed: 2022-05-09.
- [4] Nvidia Triton inference server. <https://developer.nvidia.com/nvidia-triton-inference-server>. Accessed: 2022-05-09.
- [5] ONNX. <https://onnx.ai>. Accessed: 2022-05-09.
- [6] OpenFOAM website. <https://www.openfoam.com>. Accessed: 2022-05-09.
- [7] PyTorch 1.10: `torch.nn.Linear`. <https://pytorch.org/docs/1.10/generated/torch.nn.Linear.html>. Accessed: 2022-05-09.
- [8] TensorFlow 2.6.0: `tf.keras.layers.Dense`. [https://www.tensorflow.org/versions/r2.6/api\\_docs/python/tf/keras/layers/Dense](https://www.tensorflow.org/versions/r2.6/api_docs/python/tf/keras/layers/Dense). Accessed: 2022-05-09.
- [9] Torch-TensorRT. <https://nvidia.github.io/Torch-TensorRT/>. Accessed: 2022-05-09.
- [10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [11] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.
- [12] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

## Bibliography

- [13] V. Bozheniuk, A. Zeyer, R. Schlüter, and H. Ney. A comprehensive study of residual cnns for acoustic modeling in asr. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 7674–7678, Barcelona, Spain, May 2020.
- [14] F. Christo, A. Masri, and E. Nebot. Artificial neural network implementation of chemistry with pdf simulation of h<sub>2</sub>/co<sub>2</sub> flames. *Combustion and Flame*, 106(4):406–427, 1996.
- [15] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, mar 1990.
- [16] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- [17] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17*, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery.
- [19] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller. Data mining-based analysis of hpc center operations. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 766–773, 2017.
- [20] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18*. IEEE Press, 2018.

- [21] G. Malivenko. pytorch2keras. <https://github.com/gmalivenko/pytorch2keras>. Accessed: 2022-01-19.
- [22] M. Martineau, P. Atkinson, and S. McIntosh-Smith. Benchmarking the nvidia v100 gpu and tensor cores. In G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R. R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J. D. Garcia Sanchez, and S. L. Scott, editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 444–455, Cham, 2019. Springer International Publishing.
- [23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.
- [24] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10*, page 807–814, Madison, WI, USA, 2010. Omnipress.
- [25] L. A. Ochoa. Multilayer perceptron with tikz, Feb. 2019.
- [26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [27] EU Center of Excellence for Performance Optimization and Productivity. POP. <https://pop-coe.eu>. Accessed: 2022-05-26.
- [28] NEC Corporation. VEDA. <https://github.com/SX-Aurora/veda>. Accessed: 2022-05-09.
- [29] NEC Corporation. *PROGINF/FTRACE User's Guide*, 7 edition, Dec 2021. Accessed: 2022-04-26.
- [30] M. Riedel, R. Sedona, C. Barakat, P. Einarsson, R. Hassanian, G. Cavallaro, M. Book, H. Neukirchen, and A. Lintermann. Practice and experience in using parallel and scalable machine learning with heterogenous modular supercomputing architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 76–85, 2021.
- [31] C. Rosas, J. Giménez, and J. Labarta. Scalability prediction for fundamental performance factors. *Supercomputing frontiers and innovations*, 1(2):4–19, 2014.

## *Bibliography*

- [32] J. Van Oijen, A. Donini, R. Bastiaans, J. ten Thije Boonkkamp, and L. De Goey. State-of-the-art in premixed combustion modeling using flamelet generated manifolds. *Progress in Energy and Combustion Science*, 57:30–74, 2016.
- [33] N. Weber and F. Huici. SOL: effortless device support for AI frameworks without source code changes. *CoRR*, abs/2003.10688, 2020.