Hybrid Inference Optimization for AI-Enhanced Turbulent Boundary Layer Simulation on Heterogeneous Systems

Fabian Orland
orland@itc.rwth-aachen.de
RWTH Aachen University
High Performance Computing
Aachen, Germany

Tom Hilgers
tom.hilgers@rwth-aachen.de
RWTH Aachen University
Aachen, Germany

Fabian Hübenthal
f.huebenthal@aia.rwth-aachen.de
RWTH Aachen University
Chair of Fluid Mechanics and
Institute of Aerodynamics
Aachen, Germany

Rakesh Sarma
r.sarma@fz-juelich.de
Forschungszentrum Jülich GmbH
Jülich Supercomputing Centre
Jülich, Germany

Andreas Lintermann
a.lintermann@fz-juelich.de
Forschungszentrum Jülich GmbH
Jülich Supercomputing Centre
Jülich, Germany

Christian Terboven
terboven@itc.rwth-aachen.de
RWTH Aachen University
High Performance Computing
Aachen, Germany

Abstract

Active drag reduction (ADR) using spanwise traveling surface waves is a promising approach to reduce drag of airplanes by manipulating the turbulent boundary layer (TBL) around an airfoil, which directly translates into power savings and lower emission of greenhouse gases harming the environment. However, no analytical solution is known to determine the optimal actuation parameters of these surface waves based on given flow conditions. Data-driven deep learning (DL) techniques from artificial intelligence (AI) are a promising alternative approach, but their training requires a huge amount of high-fidelity data from computationally expensive computational fluid dynamics (CFD) simulations. Previous works proposed a TBL-Transformer architecture for the expensive time-marching of turbulent flow fields and coupled it with a finite volume solver from the multi-physics PDE solver framework m-AIA to accelerate the generation of TBL data. To accelerate the computationally expensive inference of the TBL-Transformer, the AIxeleratorService library was used to offload the inference task to GPUs. While this approach significantly accelerates the inference task, it leaves the CPU resources allocated by the solver unutilized during inference. To fully exploit modern heterogeneous computer systems, we introduce a hybrid inference method based on a hybrid work distribution model and implement it into the AIxeleratorService library. Moreover, we present a formal model to derive the optimal hybrid work distribution. To evaluate the computational performance and scalability of hybrid inference, we benchmark the coupled m-AIA solver from previous work on a heterogeneous HPC system comprising Intel Sapphire Rapids CPUs and NVIDIA H100 GPUs. Our results show that hybrid inference achieves a performance speedup, that grows as the ratio of allocated CPU cores to GPU devices increases. We further demonstrate that the runtime improvement by hybrid inference also increases the energy efficiency of the coupled solver application. Finally, we highlight that the theoretical hybrid work distribution derived from our formal model yields near optimal results in practice.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; **Machine learning**; **Massively parallel algorithms**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Applied computing** → *Aerospace*; *Engineering*.

Keywords

CFD, TBL, ADR, AI, ML, DL, Transformer, hybrid inference, energy efficiency, AIxeleratorService

ACM Reference Format:
Fabian Orland, Tom Hilgers, Fabian Hübenthal, Rakesh Sarma, Andreas Lintermann, and Christian Terboven. 2026. Hybrid Inference Optimization for AI-Enhanced Turbulent Boundary Layer Simulation on Heterogeneous Systems. In *SCA/HPCAsia 2026 Workshops: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops (SCA/HPCAsiaWS 2026), January 26–29, 2026, Osaka, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3784828.3785255

1 Introduction

The CO_2 emissions from the aviation sector are a significant contributor to the greenhouse effect, which is the main driver of today's global climate change. Thus, research in aerospace engineering is investigating techniques to reduce an airplane's energy consumption. One promising approach is ADR using spanwise traveling surface waves to manipulate the near-wall TBL around an airfoil, which has been shown to result in significant drag reductions of up to -31% and net power savings of up to -10% [2]. In this approach a surface wave is parameterized by its wavelength λ, amplitude A, and time-period T. Given the flow conditions characterized by the Reynolds number Re, the Mach number M, and the angle of attack α, the challenging problem is to find the best actuation parameters λ^*, A^*, T^* that minimize the drag experienced by the airfoil. While a comprehensive analysis of this highly complex parameter space was done by Albers [1], no analytical solution of the

SCA/HPCAsiaWS 2026, Osaka, Japan

problem $f_{\text{ADR}} : \mathbb{R}^3 \rightarrow \mathbb{R}^3, (Re, M, \alpha) \mapsto (\lambda^*, T^*, A^*)$ is known. A promising approach to solve the ADR problem may be to utilize universal function approximation methods from AI, in particular from DL, and train a deep neural network to learn a function $f_{\text{ADR}}^{\text{AI}} \approx f_{\text{ADR}}$. Training a DL model requires a huge amount of high-fidelity data that samples the complex parameter space of flow conditions and actuation parameters. Due to the turbulent nature of the boundary layer under investigation, the generation of training data necessitates direct numerical simulations (DNSs), which often leads to infeasible computational requirements for realistic scenarios due to fine mesh resolution requirements to capture the wide range of turbulent scales. Previous work by Sarma et al. [27] proposed a method to speed up the time-marching of turbulent flow fields by coupling a Transformer architecture with a CFD solver. The Transformer architecture was successfully employed before by Wu et al. [30] to forecast time series of influenza-like illness. During a numerical TBL simulation, a CFD solver generates a time series $(U^t)_{t=0}^{t_n} = (U^0, U^1, \ldots, U^{t_n-1}, U^{t_n})$, where $U^t \in \mathbb{R}^3$ is the three-dimensional velocity field at a discrete point in time $t \in \{0, 1, \ldots, t_n\}$ and each $U^{t+1} = U^t + \frac{\partial U^t}{\partial t}\Delta t$ is obtained from U^t by integrating the solution of the Navier-Stokes momentum equation. Sarma et al. [27] adapted the Transformer architecture by Wu et al. [30] to predict the time-marching of turbulent flow fields in TBL simulations by transforming an input sequence of turbulent flow fields $(U^{t-m}, \ldots, U^{t-1}, U^t)$ into an output sequence of time-marched fields $(U^{t+1}, U^{t+2}, \ldots, U^{t+n})$.

In recent years, the development of AI-enhanced simulations by coupling CFD solvers with trained DL models has gained a lot of traction in many engineering applications dealing with turbulent or reactive flows. In these applications, DL models are used to predict interpolation weights for convective fluxes in finite volume methods [15], to speedup iterative methods solving the Poisson equation by predicting good initial guesses [12], or as closure models in large eddy simulation (LES) predicting different subfilter-scale quantities [4–6, 9, 16, 21, 31]. AI-enhanced simulation poses a computational challenge because many CFD solvers have been optimized over decades for central processing unit (CPU) architectures but the inference of a DL model mainly consists of matrix operations, that can be significantly accelerated by current GPU architectures. Driven by this increasing demand for deploying trained DL models into CFD solvers, different software packages have been developed including SmartSim [25], NNPred [19], PhyDLL [28], and the AIxeleratorService [24]. While these coupling libraries differ in the level of abstraction provided to theirs users, all of them, except NNPred, support offloading the inference task to graphics processing units (GPUs). However, if the coupled simulation application does not offer the potential to overlap the inference task with other computations, the allocated CPU cores are idle, which constitutes an inefficient resource utilization of the heterogeneous hardware.

Since the inference task is highly data-parallel, it offers the potential for a hybrid worksharing approach involving CPUs and GPUs cooperatively. Hybrid worksharing is not a new concept and was, for example, applied successfully by Luk et al. [20] in 2009, where the workload of different computation kernels was adaptively mapped between CPUs and GPUs. For DL workloads, however, hybrid approaches are still not broadly established. Some recent hybrid approaches focus on optimizing the training of graph neural networks [18, 32] or large models in general [26], but hybrid inference approaches are often only considered for mobile or edge devices [14, 17].

In this work, we propose a hybrid inference method to optimize the heterogeneous hardware utilization of AI-enhanced simulations. We demonstrate its applicability to productive applications by the example of the finite volume solver from the multi-physics PDE solver framework m-AIA, that was recently coupled with the TBL-Transformer of Sarma et al. [27] using the AIxeleratorService library [24] by Hilgers et al. [11]. We define a formal parameterized hybrid work distribution model adapted from Luk et al. [20] and derive the optimal hybrid work distribution ratio between CPUs and GPUs. Moreover, we extend the publicly available AIxeleratorService library with a general hybrid inference implementation based on our proposed hybrid work distribution model.

This paper is structured as follows: Section 2 describes the deployment of the TBL-Transformer [27] into the m-AIA solver framework and the resulting coupled workflow to accelerate the inference task on GPUs by using the AIxeleratorService embedded into a general machine learning (ML)-module as proposed by Orland et al. [24]. Section 3 introduces our proposed hybrid work distribution model for the inference task and elaborates on the implementation of this model into the AIxeleratorService library. Section 4 evaluates the proposed hybrid inference method with the coupled m-AIA solver. We demonstrate that both speedup and energy efficiency improve as the ratio of allocated CPU cores to GPU devices increases. Moreover, we evaluate the accuracy of the optimal CPU fractions derived from our formal hybrid work distribution model compared to the real observed optima. Finally, Section 5 discusses future work and Section 6 summarizes our conclusions.

2 Transformer-Enhanced TBL Simulation

In this work we use the finite volume solver on structured grids (FVStructuredSolver) of the multi-physics PDE solver framework m-AIA [22], that was recently coupled by Hilgers et al. [11] with the TBL-Transformer model trained by Sarma et al. [27] to speed up the generation of actuated TBL data. Since the setup of the coupled solver is an important foundation for this work, we first summarize the deployment of the TBL-Transformer into the m-AIA solver and then elaborate on the acceleration of the inference task using the AIxeleratorService library [23].

2.1 TBL-Transformer Deployment into m-AIA

Since the TBL-Transformer was trained on simulation snapshots with a temporal resolution of $\Delta\tau = 24$ time steps [27], it is ideally inferred once every $(m-1)\Delta\tau$ time steps of the coupled simulation, where m denotes the context window size, for physical consistency. To maintain a feasible hidden dimension d_{model} [29] of the TBL-Transformer's embedding space and to control the computational cost, the model was trained on individual cubic subdomains as input rather than the full velocity field. This means the full velocity field is mathematically decomposed according to

$$\begin{aligned} &f_{\text{decomp}} : \mathbb{R}^{n_x \times n_y \times n_z \times 3} \rightarrow \mathbb{R}^{3 \times n_c \times d_c^3}, U^t \mapsto \hat{U}^t \\ &= \left((\hat{u}_1^t, \ldots, \hat{u}_{n_c}^t), (\hat{v}_1^t, \ldots, \hat{v}_{n_c}^t), (\hat{w}_1^t, \ldots, \hat{w}_{n_c}^t) \right), \end{aligned} \tag{1}$$

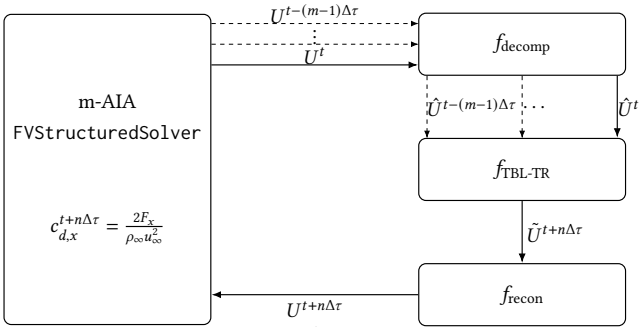

Figure 1: Deployment of the TBL-Transformer model into the `FVStructuredSolver` of the m-AIA solver framework.

where each $\hat{u}_i^t, \hat{v}_i^t, \hat{w}_i^t \in \mathbb{R}^{d_c^3}, i \in \{1, \ldots, n_c\}$ is a cubic subdomain of size $d_c \times d_c \times d_c$ of one scalar velocity component and $n_c = \left\lceil \frac{n_x}{d_c} \right\rceil \cdot \left\lceil \frac{n_y}{d_c} \right\rceil \cdot \left\lceil \frac{n_z}{d_c} \right\rceil$ is the resulting total number of subdomains. The inverse transformation that reconstructs the full velocity field from a decomposed set of cubic subdomains is denoted by $f_{\text{recon}} = f_{\text{decomp}}^{-1}$. Consequently, the time-marching for a forecasting window size n learned by the TBL-Transformer based on a context window size m is defined as:

$$f_{\text{TBL-TR}} : \left(\mathbb{R}^{m\times 3\times d_c^3}, \mathbb{R}^{n\times 3\times d_c^3}\right) \to \mathbb{R}^{n\times 3\times d_c^3},$$
$$((\hat{c}^{t-m+1}, \ldots, \hat{c}^t), (\hat{c}^t, \ldots, \hat{c}^{t+n-1})) \mapsto (\tilde{c}^{t+1}, \ldots, \tilde{c}^{t+n}), \qquad (2)$$

where $\hat{c}_i^t \in \{\hat{u}_i^t, \hat{v}_i^t, \hat{w}_i^t\}_{i=1}^{n_c}$ is an arbitrary cubic subdomain at time instance t, $(\hat{c}^{t-m+1}, \ldots, \hat{c}^t)$ is the input sequence for the encoder, and $(\hat{c}^t, \ldots, \hat{c}^{t+n-1})$ is the target sequence input for the decoder.

The resulting workflow of the simulation coupled with the TBL-Transformer is illustrated in Figure 1. Three different types of time steps need to be distinguished as defined by Hilgers et al. [11]: i) regular solver step, ii) coupling step, and iii) inference step. In a regular solver step, the TBL-Transformer is not used and the usual finite volume method solution of the governing equations is advanced in time using the Runge-Kutta method. In a coupling step, the solver decomposes the velocity field according to the definition of f_{decomp} in Equation (1) in preparation for a future inference step. In an inference step, the whole coupling loop, shown in Figure 1 is executed. Suppose at time step t of the coupled simulation, the TBL-Transformer should be inferred. This means at the previous time steps $(t - k\Delta\tau)_{k=1}^m$, the solver already needs to decompose the velocity field into cubic subdomains $\hat{U}^{t-k\Delta\tau} = f_{\text{decomp}}(U^{t-k\Delta\tau})$ and keep them stored until time step t is reached. At time step t, the current velocity field is also decomposed into cubic subdomains $\hat{U}^t = f_{\text{decomp}}(U^t)$ first, but then the whole coupling loop can be executed. Since all required inputs for the TBL-Transformer are available, the model is inferred n-times autoregressively on batches of time series of individual cubic subdomains yielding the full decomposed velocity field $\tilde{U}^{t+n\Delta\tau} = f_{\text{TBL-TR}}\left((\hat{U}^{t-(m-1)\Delta\tau}, \ldots, \hat{U}^t), \tilde{U}^{t+(n-1)\Delta\tau}\right)$. Subsequently, the time-marched velocity field $U^{t+n\Delta\tau} = f_{\text{recon}}(\tilde{U}^{t+n\Delta\tau})$ is reconstructed. The Runge-Kutta time integration step is skipped and the internal variables to keep track of the simulated physical time are incremented manually by $n\Delta\tau\Delta t$, where Δt is the time step size of the solver constrained by the CFL condition [8]. Finally the solver computes the new streamwise drag coefficient $c_{d,x}^{t+n\Delta\tau}$ based on the time-marched velocity field. Further information regarding the computation of the drag coefficient and related physical analysis can be found in [2].

2.2 Accelerating the TBL-Transformer Inference on GPUs

While the m-AIA solver is an MPI-parallel program running on CPUs, the inference of the trained TBL-Transformer is computationally expensive and suitable to be accelerated by GPUs since it involves many matrix operations. To achieve GPU acceleration of the inference task in this work, we build upon the implementation of the coupled m-AIA solver by Hilgers et al. [11], in which the Fortran-based ML-module developed by Orland et al. [24] was extended and partially ported to C++ for easier integration into the C++-based m-AIA framework. Using this modular coupling approach also enables separation of concerns allowing the actual `FVStructuredSolver` code to stay mostly intact. The main logic and computations of the coupling workflow described in Section 2.1 are encapsulated in the ported ML-module. Only a few lines were added to the solver code implementing the interaction with the ML-module. The interaction between the solver and our ML-module implementation is illustrated by the sequence diagram shown in Figure 2. A new class `MLCouplingMAIA` was derived from the abstract class `MLCoupling`, formerly `ml_coupling_t` in Fortran. The abstract parent class `MLCoupling` exposes a single method `ml_step()` to the solver to initiate the 3-step coupling workflow comprising `preprocess_input`, `inference`, and `postprocess_output`. The child class `MLCouplingMAIA` provides concrete implementations for these abstract methods. Moreover, it manages internal memory buffers representing an input tensor $\mathcal{T}_{\text{in}} \in \mathbb{R}^{3n_c\times m\times d_c^3}$ and an output tensor $\mathcal{T}_{\text{out}} \in \mathbb{R}^{3n_c\times n\times d_c^3}$ for the inference of the TBL-Transformer. The `preprocess_input` method implements the decomposition of the given full velocity field U^t into the cubic subdomains according to the definition of f_{decomp} in Equation (1) by filling the correct indices of the input tensor $\mathcal{T}_{\text{in}}$.

The `MLCouplingMAIA` class also implements the logic to distinguish between the three different types of time steps. In the coupling or inference step, the m-AIA solver calls `ml_step()` and provides a memory pointer to the velocity field. In a regular solver step, no further computations are performed by the `MLCouplingMAIA` class. In a coupling step, only the `preprocess_input` method is called inside the `MLCouplingMAIA` class. In an inference step, the `MLCouplingMAIA` class knows, that the input tensor $\mathcal{T}_{\text{in}}$ has been completely filled, and therefore calls into the `inference` method.

The implementation of the `inference` method follows the popular strategy design pattern [10] and is encapsulated in a general, modular `MLCouplingStrategyAix` class. This class was omitted from Figure 2 for visual clarity as it is essentially a wrapper around our AIxeleratorService library[1] and the concept is very similar to our previous work [24]. For this work, it is important to highlight the steps performed inside the AIxeleratorService library. First the

[1] https://github.com/RWTH-HPC/AIxeleratorService

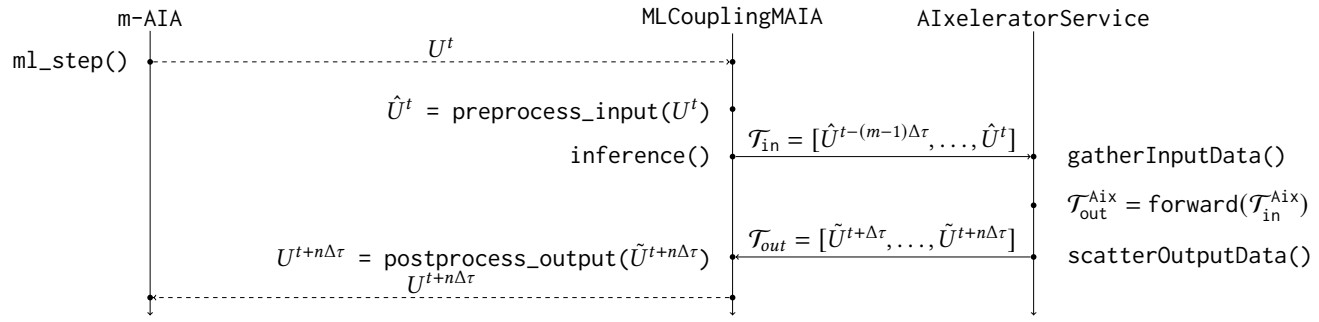

Figure 2: Sequence diagram illustrating the implemented modular coupling workflow from Orland et al. [24].

AIxeleratorService gathers the input tensors $\mathcal{T}_{\text{in}}$ from all m-AIA solver processes via MPI and concatenates them along the first dimension. This means, the AIxeleratorService internally deals with a huge tensor $\mathcal{T}_{\text{in}}^{\text{Aix}} \in \mathbb{R}^{3N_c \times m \times d_c^3}$, where $N_c = \sum_{p=0}^{N_p} n_{c,p}$ is the total number of cubic subdomains aggregated over all solver processes $p \in \{0, \ldots, N_p\}$. The gathered tensor is then moved to the GPU and the inference of the TBL-Transformer is performed using PyTorch's C++API from libtorch. To enable inference of the TBL-Transformer in C++we used the TorchScript JIT compiler to create a compiled version of the TBL-Transformer's `run_encoder_decoder_inference` Python function, that is available in the public AI4HPC repository[2]. After the forward pass through the TBL-Transformer is performed on the GPU, the predicted time-marched cubic subdomains are scattered back to their corresponding solver process. Inside the `postprocess_output` method of the `MLCouplingMAIA` class, the full velocity field is reconstructed, according to the definition of $f_{\text{recon}} = f_{\text{decomp}}^{-1}$, and passed back to the solver process.

3 Hybrid Inference Optimization

The inference of the TBL-Transformer is a highly data-parallel task since the TBL-Transformer learned the time-marching of an individual cubic subdomain, see Equation (2). In deep learning it is a common practice to train and infer neural networks on batches of data samples according to a defined batch size because the computations inside a neural network then become efficient matrix-matrix operations. Offloading the inference task to GPUs as described in Section 2.2 introduces a load imbalance between CPU and GPU resources, as all samples are inferred by the GPUs. In this work, the coupled simulation creates a huge number of constant sized samples. This opens the possibility for a hybrid inference scheme, that distributes the number of data samples to be inferred between CPU and GPU resources to optimize the utilization of the allocated heterogeneous hardware. In this section, we first highlight the load balance problem introduced by offloading the inference of the TBL-Transformer to GPUs. Secondly, we define a formal hybrid work distribution model to formulate and solve an optimization problem to find the optimal work distribution resulting in the most balanced execution times between CPU and GPU resources. Finally, we describe our general hybrid inference extension implemented into the AIxeleratorService library based on the formal hybrid work distribution model.

3.1 Load Imbalance of the GPU-Offloaded TBL-TR Inference

The internal `distributionStrategy` component of the AIxeleratorService library implements a distributed client-server architecture [24], which partitions the `MPI_COMM_WORLD` communicator into one workgroup subcommunicator per allocated GPU device in a (heterogeneous) HPC job. Inside each workgroup one MPI process running on a GPU node acts as a designated server while all remaining processes become clients. Each server process is responsible to gather the preprocessed input data for the TBL-Transformer from the clients within its workgroup communicator, offload the inference of the TBL-Transformer to the GPU via libtorch's C++ API, and finally scatter the resulting predictions of the model back to the client processes. This distributed client-server architecture inherently leads to a load imbalance between clients and servers. Figure 3 shows a trace of one exemplary `ml_step` execution during an inference step highlighting the load imbalance problem. In this example the coupled m-AIA solver was executed on a single GPU node of the CLAIX-2023 cluster with 1 GPU device and 96 CPU cores allocated. The whole `ml_step` takes approximately 7.68 seconds. The pre-processing routines at the beginning and the post-processing routines at the end of the `ml_step` are barely visible. The `gatherInputData` step is highlighted by the orange colored trace events. Most of the time is spent for the inference of the TBL-Transformer, which is managed by process 0 as a server and indicated by the dark green bar in the timeline of this example. The remaining 95 processes are clients and only 9 of them are visualized in the trace for visual clarity as they all execute the same code. The clients waiting inside the `MPI_Scatterv` collective are effectively unutilized during this time frame, which constitutes inefficient usage of the allocated heterogeneous hardware resources. To reduce the time to solution, we propose a hybrid inference scheme, which also involves the CPU cores of the client processes.

3.2 Hybrid Work Distribution Model

The challenging part of the hybrid inference scheme is to split the total number of samples to be inferred by the CPU and GPU resources respectively. In this work, the total number of samples

[2]https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/ai4hpc/ai4hpc/-/blob/master/Cases/src/networks.py

Figure 3: Vampir trace visualization of one exemplary `ml_step` execution highlighting the load imbalance between CPU and GPU during offloaded TBL-Transformer inference. The client processes 1 to 95 execute the same code, but only 9 of them are shown in the timeline. The pre- and post-processing is barely visible at the beginning and end of the trace. The `gatherInputData()` and scatterOutputData() step is highlighted in orange and red, respectively. The GPU-accelerated inference of the TBL-Transformer is highlighted in dark green on server process 0 taking more than 7 seconds of runtime.

depends on the resolution of the discrete computational grid. The decomposition of the full velocity field on a discrete grid with $n_x \times n_y \times n_z = 732 \times 131 \times 250$ cells according to Equation (1) results in a total number of $N_{\text{total}} = 3 \times n_c = 3 \times \frac{n_x}{d_c} \times \frac{n_y}{d_c} \times \frac{n_z}{d_c} \approx 140467$ samples. Note that this does not consider any domain decomposition among MPI processes in a parallel execution of the solver. In a parallel solver execution the decomposition into cubic subdomains for the TBL-Transformer is performed by each process locally. If in any spatial dimension the length of the local domains is not perfectly divisible by the cube length d_c additional cubes compared to the non-parallel case are created. Thus, the total number of samples may increase in a parallel execution. To distribute the total number of samples between CPU and GPU resources, we introduce a parameter $\alpha \in \mathbb{R}, 0 \leq \alpha \leq 1$ to control the fraction of all samples that get assigned to the CPU resources. This yields the parameterized hybrid work distribution

$$N_{\text{total}} = N_{\text{CPU}} + N_{\text{GPU}} = \alpha N_{\text{total}} + (1-\alpha) N_{\text{total}}, \tag{3}$$

where $N_{\text{CPU}} = \alpha N_{\text{total}}$, $N_{\text{GPU}} = (1-\alpha) N_{\text{total}}$ denote the number of samples assigned to the CPU and GPU resources, respectively. The hybrid optimization problem is now to find α^* such that all CPU cores and GPU devices finish processing their assigned workload at the same time. To solve this problem, a formalization of the resulting runtime depending on the number of assigned samples for CPU cores and GPU devices is required. The runtime of the whole `ml_step()` can be defined as

$$T_{\texttt{ml\_step}}(N_{\text{total}}) = T_{\text{pre}}(N_{\text{total}}) + T_{\text{inf}}(N_{\text{total}}) + T_{\text{post}}(N_{\text{total}}) \tag{4}$$

where T_{pre}, T_{post} denote the runtime of the `preprocess_input()` and `postprocess_output()` calls. It should be noted that the gathering and scattering of the input and output data for the TBL-Transformer is only required for the N_{GPU} samples inferred by the GPU resources. The N_{CPU} samples for the CPU resources can be directly inferred by each solver process locally on the CPU core it is running on without requiring any communication. Thus, the runtime of the hybrid `inference()` call is defined as

$$T_{\texttt{inf}}(N_{\text{total}}) = \max \begin{cases} T^{\text{CPU}}_{\texttt{inf}}(N_{\text{CPU}}) = T^{\text{CPU}}_{\texttt{fwd}}(N_{\text{CPU}}) \\ T^{\text{GPU}}_{\texttt{inf}}(N_{\text{GPU}}) = T^{\text{GPU}}_{\texttt{fwd}}(N_{\text{GPU}}) + T_{\texttt{OH}}(N_{\text{GPU}}), \end{cases} \tag{5}$$

where $T^{\text{CPU}}_{\texttt{inf}}(N_{\text{CPU}})$ denotes the runtime of inferring N_{CPU} samples on the CPU resources and $T^{\text{GPU}}_{\texttt{inf}}(N_{\text{GPU}})$ denotes the runtime of inferring N_{GPU} samples on the GPU resources. The time of the `forward()` pass through the TBL-Transformer on CPU and GPU resources is denoted by $T^{\text{CPU}}_{\texttt{fwd}}$ and $T^{\text{GPU}}_{\texttt{fwd}}$, respectively. In case of GPU inference, the communication overhead is summarized by

$$\begin{aligned} T_{\texttt{OH}}(N_{\text{GPU}}) = T_{\text{MPIG}}(N_{\text{GPU}}) + T_{\texttt{H2D}}(N_{\text{GPU}}) \\ + T_{\texttt{D2H}}(N_{\text{GPU}}) + T_{\text{MPIS}}(N_{\text{GPU}}), \end{aligned} \tag{6}$$

where T_{MPIG} and T_{MPIS} denote the runtime of `gatherInputData()` and `scatterOutputData()`, respectively. Similarly, the time spent to transfer the gathered input data from the host to the device is denoted by $T_{\texttt{H2D}}$ and the time to move the predicted output from the device back to the host is denoted by $T_{\texttt{D2H}}$, respectively. As both the CPU and the GPU resources infer their assigned number of samples in parallel, the resulting hybrid inference runtime $T_{\texttt{inf}}(N_{\text{total}})$ is constrained by the maximum of $T^{\text{CPU}}_{\texttt{inf}}(N_{\text{CPU}})$ and $T^{\text{GPU}}_{\texttt{inf}}(N_{\text{GPU}})$. Based on this formulation, the optimization problem to find the optimal CPU fraction α^* becomes

$$\alpha^* = \underset{0 \leq \alpha \leq 1}{\operatorname{argmin}} \, T_{\texttt{ml\_step}}(N_{\text{total}}). \tag{7}$$

It should be noted, that the pre- and post-processing is always done by each solver process locally on its allocated CPU core, so it does not depend on α. Minimizing Equation (7) means minimizing $T_{\texttt{ml\_step}}(N_{\text{total}})$ given by Equation (4). The maximum operator in the term $T_{\texttt{inf}}(N_{\text{total}})$, see Equation (5), becomes minimal iff

$$T^{\text{CPU}}_{\texttt{inf}}(N_{\text{CPU}}) = T^{\text{GPU}}_{\texttt{inf}}(N_{\text{GPU}}). \tag{8}$$

This means to find α^*, minimizing $T^{\alpha}_{\texttt{ml\_step}}(N_{\text{total}})$ from Equation (4), one needs to solve

$$T^{\text{CPU}}_{\texttt{fwd}}(N_{\text{CPU}}) = T^{\text{GPU}}_{\texttt{fwd}}(N_{\text{GPU}}) + T_{\texttt{OH}}(N_{\text{GPU}}). \tag{9}$$

Since this is a single equation, that needs to be solved for a single variable α^*, a unique solution exists. Applying the definition of the hybrid work distribution from Equation (3) yields

$$T^{\text{CPU}}_{\texttt{fwd}}(\alpha N_{\text{total}}) = T^{\text{GPU}}_{\texttt{fwd}}((1-\alpha) N_{\text{total}}) + T_{\texttt{OH}}((1-\alpha) N_{\text{total}}). \tag{10}$$

To be able to solve this equation for α we assume that all three terms obey ideal strong scaling such that for any scalar $\gamma \in \mathbb{R}$ the

following relations hold:

$$\begin{aligned} T_{\text{fwd}}^{\text{CPU}}(\gamma N_{\text{total}}) &= \gamma T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}}), \\ T_{\text{fwd}}^{\text{GPU}}(\gamma N_{\text{total}}) &= \gamma T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}), \\ T_{\text{OH}}(\gamma N_{\text{total}}) &= \gamma T_{\text{OH}}(N_{\text{total}}). \end{aligned} \tag{11}$$

For the computational terms $T_{\text{fwd}}^{\text{CPU}}$ and $T_{\text{fwd}}^{\text{GPU}}$ this assumption should be reasonable as the computational work per cubic subdomain sample is constant and the most expensive operations inside a deep Transformer are matrix-matrix multiplications, which are compute-bound. A single cubic sub-domain sample contains $m \times d_c^3 = 5{\times}8^3 = 2560$ single-precision floating point elements requiring approximately 10 KB of memory. Even at the largest scale of solver processes used in this work, each process holds hundreds of those samples. This means the message sizes communicated via MPI or PCIe are large enough, such that also the overhead terms should scale asymptotically linear [13]. Applying these assumptions from Equation (11) to Equation (10) yields

$$\alpha T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}}) = (1-\alpha) T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) + (1-\alpha) T_{\text{OH}}(N_{\text{total}}), \tag{12}$$

which can be rearranged and finally solved for α^* yielding

$$\alpha^* = \frac{T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) + T_{\text{OH}}(N_{\text{total}})}{T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}}) + T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) + T_{\text{OH}}(N_{\text{total}})}. \tag{13}$$

Based on the optimal CPU fraction α^* we can also define the expected speedup of using hybrid inference compared to the naive fully GPU-offloaded inference with $\alpha = 0$ as

$$\begin{aligned} S_{\alpha*} &= \frac{T_{\texttt{ml\_step}}^{\alpha=0}(N_{\text{total}})}{T_{\texttt{ml\_step}}^{\alpha=\alpha^*}(N_{\text{total}})} \\ &= \frac{T_{\text{pp}}(N_{\text{total}}) + T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) + T_{\text{OH}}(N_{\text{total}})}{T_{\text{pp}}(N_{\text{total}}) + (1-\alpha^*) T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) + (1-\alpha^*) T_{\text{OH}}(N_{\text{total}})}, \end{aligned} \tag{14}$$

where $T_{\texttt{ml\_step}}^{\alpha}(N_{\text{total}})$ denotes the evaluation of Equation (4) for a chosen α and $T_{\text{pp}}(N_{\text{total}}) = T_{\text{pre}}(N_{\text{total}}) + T_{\text{post}}(N_{\text{total}})$ for brevity.

As a result, the practical evaluation of the optimal CPU fraction α^* and the related speedup requires two executions of the coupled solver on the same allocated heterogeneous hardware configuration. In one execution, the TBL-Transformer inference is performed purely on the CPU resources by setting $\alpha = 1$ to determine $T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}})$, $T_{\text{pre}}(N_{\text{total}})$, and $T_{\text{post}}(N_{\text{total}})$ from profiling. In a second execution, the inference is then performed purely on the allocated GPU resources by setting $\alpha = 0$ to determine $T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}})$ and $T_{\text{OH}}(N_{\text{total}})$ from profiling.

3.3 AIxeleratorService Extension

The modular software architecture of the AIxeleratorService, developed in previous work by Orland et al. [24], based on the strategy design pattern [10] allows to extend the library with a hybrid inference approach as described in Section 3.2. Each process of the coupled CFD solver creates an instance of an AIxeleratorService object during runtime. Each of these objects stores a unique pointer to the internal `InferenceStrategy` class, that encapsulates the inference of a given input tensor through a given deep learning model on a specified device meaning CPU or GPU. If GPUs are allocated in a parallel job executing the coupled simulation, only those processes, that get internally designated as server processes, instantiate this unique pointer with a concrete `InferenceStrategy` object currently. To enable the hybrid inference scheme described in Section 3.2, we modified the code to also allow every client process to instantiate a concrete `InferenceStrategy` object.

The implementation of the hybrid work distribution defined in Equation (3) is more challenging. First we introduced a parameter `host_fraction`, that users can specify when initializing an AIxeleratorService object in their application, to define the CPU fraction α. The internal communication mechanism of the AIxeleratorService library allows to gather data from the client ranks to the server ranks within a work group sub-communicator. Thus, we decided to implement the hybrid work split at the granularity of individual work groups. Each work group contains a number of MPI processes $N_{\text{ranks}} = N_{\text{S}} + N_{\text{C}}$, where N_{S} denotes the number of servers and N_{C} denotes the number of clients. Since there is always a single server, this implies $N_{\text{C}} = N_{\text{ranks}} - 1$. Further, let N_p denote the number of samples of process with rank p. The hybrid work distribution depending on α is determined within each work group during initialization of the AIxeleratorService library by each server process using Algorithm 1. First the server computes the

Algorithm 1 Hybrid Work Split per AIxeleratorService Workgroup

Require: number of ranks N_{ranks}, number of clients $N_{\text{C}} = N_{\text{ranks}} - N_{\text{S}}$, samples of process p N_{p}, CPU fraction α

1: $N_{\text{total}} = \sum_{p=0}^{N_{\text{C}}} N_p$
2: $N_{\text{CPU}} = \lfloor \alpha N_{\text{total}} \rfloor$
3: $\bar{N}_{\text{CPU}} = N_{\text{CPU}} / N_{\text{C}}$
4: **for** p = 1 to N_{C} **do**
5: $\quad N_{\text{CPU},p} = \min(N_p, \bar{N}_{\text{CPU}})$
6: **end for**
7: **if** $N_{\text{CPU}} \bmod N_{\text{C}} = N_{\text{R}} \wedge N_{\text{R}} > 0$ **then**
8: $\quad$ **for** p = 1 to N_{R} **do**
9: $\quad\quad N_{\text{CPU},p} += 1$
10: $\quad$ **end for**
11: **end if**
12: $N_{\text{CPU},0} = 0 \wedge N_{\text{GPU},0} = N_0$
13: **for** p = 1 to N_{C} **do**
14: $\quad N_{\text{GPU},p} = N_p - N_{\text{CPU},p}$
15: **end for**
16: **return** Hybrid work distribution $(N_{\text{CPU},p}, N_{\text{GPU},p})$ per process p.

total number of samples N_{total} across all processes within the work group (line 1). Then it determines the number of samples N_{CPU}, that should be inferred on CPU resources by all client processes based on the CPU fraction α (line 2). These samples need to be distributed equally among all client processes, so that each client should get $\bar{N}_{\text{CPU}}$ samples (line 3). If the simulation domain is not equally divisible among the solver processes, the resulting number of cubic subdomain samples N_p per process p also varies across processes. In those cases it may happen that the number of samples N_p is smaller than the average of number of samples $\bar{N}_{\text{CPU}}$, that should be inferred by each client in the hybrid setting. Since the internal communication mechanism of the AIxeleratorService library only allows to gather samples from the clients to the server

but not vice versa and also not between clients, the server can only assign $N_{\text{CPU},p} = \min(N_p, \bar{N}_{\text{CPU}})$ samples to process p for hybrid inference (line 5). Moreover, if N_{CPU} cannot be equally divided among all clients, some remainder samples $N_{\text{R}} < N_{\text{C}}$ may be left over (line 7). In this case, the server assigns the remaining samples in a round robin fashion among the clients (line 9). Since the server process ($p = 0$) itself is responsible to perform the inference of the gathered samples on the GPU via a blocking libtorch C++API, all samples of the server process need to be defined for GPU inference by setting $N_{\text{CPU},0} = 0 \wedge N_{\text{GPU},0} = N_0$ (line 12). Finally, the server needs to determine how many samples $N_{\text{GPU},p}$ each client process will send to the server during the `gatherInputData()` step, see Figure 2. Theoretically, it should be possible to also involve the server processes to perform part of the inference on their allocated CPU core. However, this requires an asynchronous launch of the GPU inference, which we have not investigated further yet because we already expect a significant performance improvement from involving the client processes in the hybrid inference approach.

4 Results

To demonstrate the applicability to real world applications, we evaluate the computational performance of our hybrid inference approach introduced in Section 3 with the coupled m-AIA + TBL-Tranformer solver on the TBL case. First we describe the heterogeneous hardware architecture used to perform our experiments with the coupled solver. Second, we analyze the relationship between the CPU fraction α and the resulting runtime of the `ml_step` on two exemplary resource allocations using 1 and 4 GPU devices, respectively. Next, we also demonstrate strong scalability of our hybrid inference approach with increasing CPU resources and identify the observed real optimal CPU fraction $\tilde{\alpha}$. After that, we highlight that runtime reductions of the `ml_step` also result in corresponding reductions of the energy consumed for the simulation. Finally, we evaluate the accuracy of our hybrid work distribution model explained in Section 3.2 by comparing the theoretical optimum α^* to the observed real optimum $\tilde{\alpha}$.

4.1 Experimental Setup

In this section, we first describe the heterogeneous hardware setup, that was used to run the coupled m-AIA solver. Second, we describe the configuration of the coupled m-AIA solver with respect to the hyper-parameters of the TBL-Transformer.

4.1.1 Heterogeneous Hardware Setup. All experiments were conducted on the CLAIX-2023 cluster at RWTH Aachen University, which consists of two segments. The traditional HPC segment consists of compute nodes equipped with two Intel Xeon 8468 Sapphire Rapids CPUs and a total of 96 cores. Three different memory configurations are available: i) 256 GB, ii) 512 GB, and iii) 1024 GB. The Machine Learning (ML) segment provides compute nodes with the same CPU configuration, that are additionally equipped with 4 Nvidia H100 GPUs. These nodes always provide 512 GB of main memory and 96 GB of HBM2e on each GPU device. The nodes from both segments are interconnected via a single Nvidia/Mellanox NDR InfiniBand fat-tree network. Communication between nodes is routed through at most two switches and the unidirectional communication bandwidth is 25 GB/s. The single node experiments in this work were executed on CPU nodes with 512 GB of main memory. For experiments involving two or more CPU nodes 256 GB of main memory per node were sufficient to instantiate a copy of the TBL-Transformer in each process. In all cases heterogeneous hardware resources were allocated via a heterogeneous SLURM job, that allocates one GPU node from the ML segment and up to $X \in \{0, 1, 3, 7\}$ CPU nodes from the HPC segment such that the total number of compute nodes ranges from 1 to 8. While the number of allocated GPU devices was varied, always all CPU cores were allocated on each node.

4.1.2 Coupled Simulation Setup. The m-AIA solver coupled with the TBL-Transformer was executed for 3000 time steps in total. Every 5 time steps an inference of the TBL-Transformer, using the previous $m = 5$ time steps as input, is performed that advances the velocity fields $n = 2$ times by $\Delta\tau/2 = 12$ time steps each. This means, the inference step triggered by calling the `ml_step()` method is executed 103 times in total. In a real simulation run, where the TBL-Transformer should be used consistent with its training, which implies a context windows size $m = 3$ and a forecasting window size $n = 2$. In such a case the inference would be executed once every $24(m - 1) = 48$ time steps and the physical time would be advanced by $24n = 48$ time steps as explained in Section 2.1. Since the focus of this work is to evaluate the computational performance of the hybrid inference approach, these parameters were chosen to keep the required core-hours for performing these experiments reasonable. A physical evaluation of the TBL-Transformer using the correct time stepping scheme according to training can be found in previous work by Hilgers et al. [11].

4.2 Hybrid Work Split Evaluation

Figure 4 shows the effect of the CPU fraction α on the resulting runtime of the `ml_step()` call. In both cases the m-AIA solver coupled with the TBL-Transformer was executed on 2 nodes comprising 1 GPU node and 1 CPU node. The left plot shows the results if only one of the four available GPU devices on the GPU node are allocated. In the right plot all four GPU devices were allocated. The horizontal axis samples the range of possible values for the CPU fraction $\alpha \in [0, 1]$ in steps of 10%. Around the real optimal CPU fraction $\tilde{\alpha}$ additional samples with a granularity of 1% steps were added. The vertical axis represents the maximum absolute runtime over all processes in seconds for different parts of the `ml_step` execution. The red line indicates the runtime of the whole `ml_step` function. The blue line and the orange line indicate the time to infer N_{CPU} samples on the CPU resources and N_{GPU} samples on the GPU resources, which corresponds to the terms $T_{\text{fwd}}^{\text{CPU}}(N_{\text{CPU}})$ and $T_{\text{fwd}}^{\text{GPU}}(N_{\text{GPU}})$ introduced in Section 3.2. Similarly, the green line shows the measured overhead denoted by $T_{\text{OH}}(N_{\text{GPU}})$.

Both plots verify our assumption made in Equation (11) that the time to perform a forward pass through the TBL-Transformer scales linearly with the number of samples to be inferred for both the CPU as well as the GPU resources. Especially the GPU resources follow this trend closely while for the CPU resources a few outliers can be observed. For example, in the left plot the runtime of the CPU inference at $\alpha = 0.4$ is higher than expected. In the right plot, the outliers for the CPU inference are less pronounced but a change in slope can be noticed for $\alpha > 0.1$. However, for the

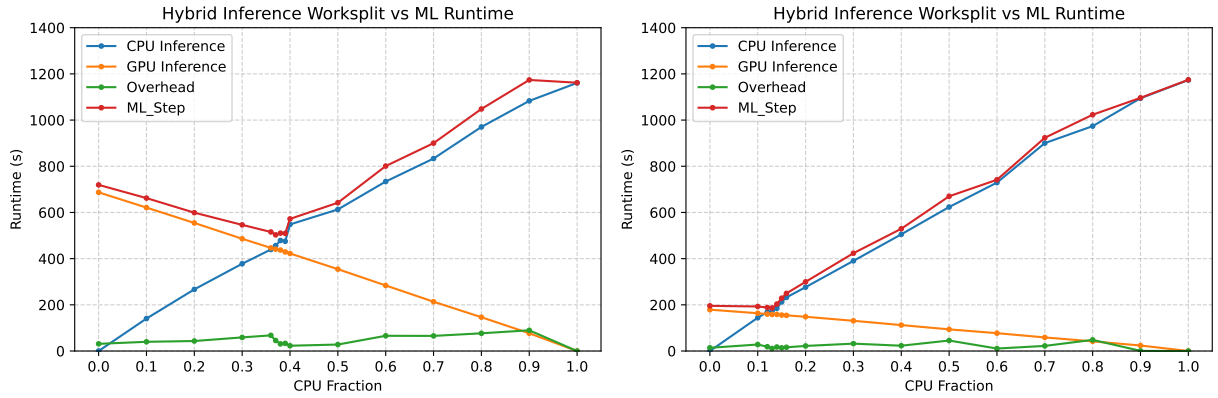

Figure 4: Work split evaluation on 2 nodes (1 GPU + 1 CPU node) with 1 GPU device (left) and 4 GPU devices (right) allocated.

overhead term our assumption cannot be verified. By increasing the CPU fraction α we would expect the overhead to go down as less samples need to be communicated from the clients to the servers and from the servers' CPU cores to the GPU devices. Instead the observed overhead increases approximately linearly between $\alpha = 0$ and $\alpha = 0.36$ and between $\alpha = 0.6$ and $\alpha = 0.9$. However, the overhead only accounts for at most 17% of the whole `ml_step` execution such that we still expect useful results from our modeling approach.

Most importantly, both plots highlight that there is indeed an optimal CPU fraction $\tilde{\alpha}$. In the left plot, where 1 GPU device was allocated, the observed optimal CPU fraction is $\tilde{\alpha} = 0.37$. At this point the hybrid inference approach results in 503 seconds runtime for the whole `ml_step`. Compared to the naive GPU offloading at $\alpha = 0$, for which a runtime of 719 seconds is observed, the hybrid inference approach yields a speedup of 1.43x. In the right plot, where 4 GPU devices were allocated, the observed optimal CPU fraction is $\tilde{\alpha} = 0.13$. At $\alpha = 0$ the `ml_step` runtime is approximately 195 seconds while the runtime at the optimum $\tilde{\alpha} = 0.13$ is approximately 186 seconds yielding a speedup of 1.05x. In this case the observable speedup by hybrid inference is much smaller, because the ratio of CPU cores to GPU devices is four times lower compared to the left plot. As four Nvidia H100 GPU devices are significantly more powerful than 92 CPU cores, it is expected that the benefit of hybrid inference becomes less significant.

4.3 Scalability & Speedup

The previous results shown in Figure 4 indicate that the speedup gained by our hybrid inference approach depends on the ratio between allocated CPU cores and GPU devices. Higher ratios of CPU cores per allocated GPU device are supposed to result in a higher speedup. Thus, we scaled the coupled m-AIA solver from 1 node to 2, 4, and 8 nodes. This experiment should not be considered a strong scaling experiment in the sense of Amdahl's law [3], because the total number of cubic subdomain samples to be inferred increases with an increasing number of processes due to different simulation domain decomposition. For example, using 8 nodes the total number of samples is 7% higher compared to using 1 node. This means, by doubling the number of nodes we do not expect

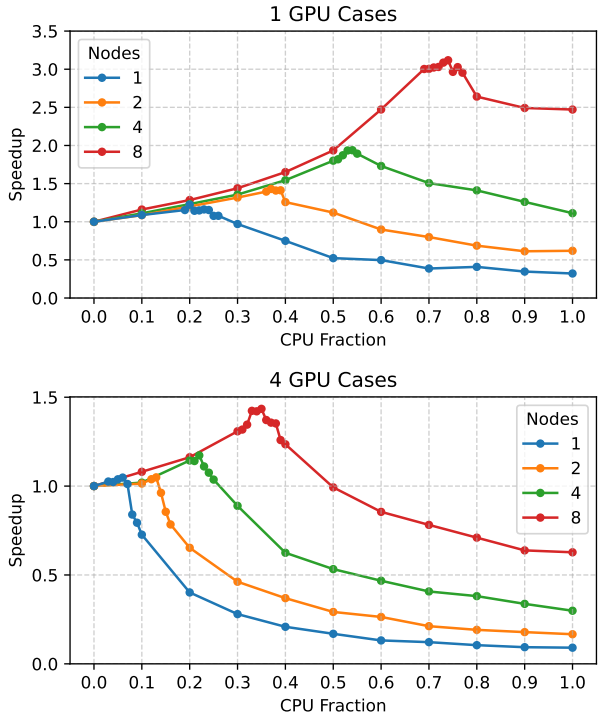

Figure 5: Scalability of hybrid inference on 1 to 8 nodes with 1 GPU device (top) and 4 GPU devices (bottom) allocated.

a two-fold speedup. Instead we want to highlight, that for each individual scale our proposed hybrid inference approach yields a speedup.

The results of this scalability experiment are shown in Figure 5. In all cases the spectrum of possible CPU fractions α on the horizontal axis was sampled as done in Figure 4 to find the optimum $\tilde{\alpha}$. The reported speedup on the vertical axis is again defined as the runtime of the `ml_step()` using fully GPU offloaded inference with $\alpha = 0$ compared to the runtime of the `ml_step()` using hybrid inference with $\alpha > 0$, i.e. $T^{\alpha=0}_{\texttt{ml\_step}} / T^{\tilde{\alpha}}_{\texttt{ml\_step}}$.

The top plot shows the speedups, that are obtained when 1 GPU device is allocated. In case of 1 node execution the optimal hybrid work split is $\tilde{\alpha} = 0.2$ yielding a speedup of 1.22x. Using 2 nodes, the optimum is at $\tilde{\alpha} = 0.37$ with a speedup of 1.43x, as reported in Section 4.2. On 4 nodes, the observed optimal hybrid work split is $\alpha = 0.54$ yielding a speedup of 1.938x, which is slightly higher than the speedup of 1.934x obtained at $\alpha = 0.53$. So the true optimum $\tilde{\alpha}$ is in between these two values. For the measurements with 8 nodes, the optimum is found at $\tilde{\alpha} = 0.74$ yielding a speedup of 3.12x. The bottom plot illustrates the case with 4 GPU devices allocated. For 1, 2, 4, and 8 nodes the observed optima $\tilde{\alpha}$ are 0.06, 0.13, 0.22, and 0.35 yielding speedups of 1.05x, 1.05x, 1.17x, and 1.43x, respectively.

In summary, the trend that higher ratios of CPU cores to GPU devices result in higher optimal CPU fractions and yield higher speedups can be verified. One might hypothesize that doubling the number of CPU cores should also shift the optimal CPU fraction by a factor of two. Since in this work the inference workload does not stay constant with increasing number of processors, this trend cannot be verified. Further investigation with a synthetic benchmark that ensures a constant workload over increasing processor counts could clarify this hypothesis.

4.4 Energy Efficiency

Energy efficiency has become an important research aspect due to the high energy demand of HPC-clusters and the rising energy prices driven by a growing energy demand worldwide. As demonstrated in Section 4.3 our hybrid inference approach results generally in reduced execution times of the `ml_step()`. Moreover, the total amount of computational work, i.e. the number of samples inferred by the TBL-Transformer, does not change for a fixed hardware allocation. As a result, we expect the hybrid execution to be more energy efficient because the same amount of work is performed by the same amount of hardware in less time.

Users of the CLAIX-2023 cluster have access to a performance monitoring[3] of their jobs. For each job running on the cluster, the monitoring system reads hardware performance counters once every minute. The energy consumption of the CPU cores and DRAM modules is measured via Intel's Running Average Power Limit (RAPL) counters [7]. For each GPU device the energy consumption is queried via the NVIDIA Management Library (NVML)[4]. In both cases the counters provide energy values integrated over time, such that no energy information is lost between two sampling points of the performance monitoring system.

Figure 6 illustrates the total energy consumed by different coupled m-AIA solver runs using the same hardware configurations, that were also used to assess scalability. For a fixed hardware configuration, let E_α denote the total energy consumed by the coupled solver if the inference is executed using the CPU fraction α. We denote the absolute energy saved by using hybrid inference with the empirically optimal $\tilde{\alpha}$ compared to full GPU-offloaded inference and full CPU-based inference by $\Delta E_{\alpha=0} = E_{\alpha=0} - E_{\alpha=\tilde{\alpha}}$ and $\Delta E_{\alpha=1} = E_{\alpha=1} - E_{\alpha=\tilde{\alpha}}$, respectively.

[3]https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/3a11a76fdf30476bb4b1a8b30661dab3/

[4]https://docs.nvidia.com/deploy/nvml-api/group__nvmlDeviceQueries.html#group__nvmlDeviceQueries_1g732ab899b5bd18ac4bfb93c02de4900a

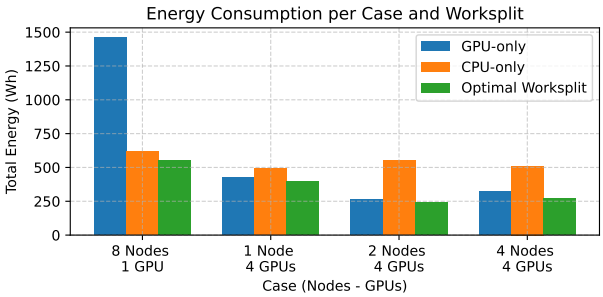

Figure 6: Energy consumption of hybrid inference with optimal $\tilde{\alpha}$ versus $\alpha = 0$ and $\alpha = 1$.

Nodes	GPUs	$\Delta E_{\alpha=0}$ [Wh]	$\Delta E_{\alpha=1}$ [Wh]
8	1	906.50 (62.1%)	64.52 (10.5%)
1	4	28.16 (6.6%)	90.80 (18.6%)
2	4	19.23 (7.4%)	307.99 (56.0%)
4	4	50.57 (15.7%)	239.90 (46.9%)

Table 1: Energy savings of hybrid inference with optimal $\tilde{\alpha}$ versus $\alpha = 0$ and $\alpha = 1$.

These energy savings are shown in Table 1. Due to technical problems, the monitoring system was missing energy measurements from the GPUs in half of the coupled solver runs. However, in all complete cases a reduction in energy consumption can be noted. Especially, in the case of 8 nodes and 1 GPU the most energy can be saved. In this case the fully GPU-offloaded inference with $\alpha = 0$ consumed approximately 1459 Wh. Compared to the other runs, the energy consumption appears exceptionally high. However, on the CLAIX-2023 cluster, the CPUs are configured to not clock down. Consequently, while the GPU performs the inference of all input samples and the clients wait inside the scatter collective, their corresponding CPU cores still run at maximum frequency, which consumes a significant amount of energy due to high number of CPU cores allocated in this run. In contrast the optimal hybrid run only consumed approximately 552 Wh. This means the hybrid run required approximately 907 Wh less energy, which is a relative energy saving of approximately 68%. In case of 4 GPUs allocated, the energy saved by hybrid inference is much lower but still significant and ranges from 6.6% on 1 node up to 15.7% on 4 nodes. Compared to the CPU-only runs with $\alpha = 1$ the hybrid inference with $\alpha = \tilde{\alpha}$ also saves between 10.5% and 56.0% of energy. It should be noted that these runs were performed exclusively on CPU nodes from the HPC-segment of CLAIX-2023 because no GPU is needed such that also no idle energy consumption of the GPUs is included in the data. Moreover, the resulting energy saving $\Delta E_{\alpha=0}$ correlates well with the yielded speedups discussed in Section 4.3. For example the optimal hybrid run on 4 nodes with 4 GPUs allocated yielded a speedup of 1.17x and required 15.7% less energy. In summary, we can conclude that the speedup obtained by our proposed hybrid inference approach also results in a proportionally more energy efficient execution of the coupled solver.

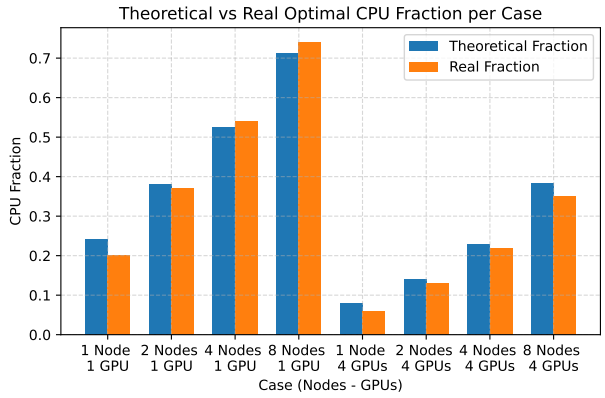

Figure 7: Comparison of the theoretical optimum to the observed real optimum for CPU fraction.

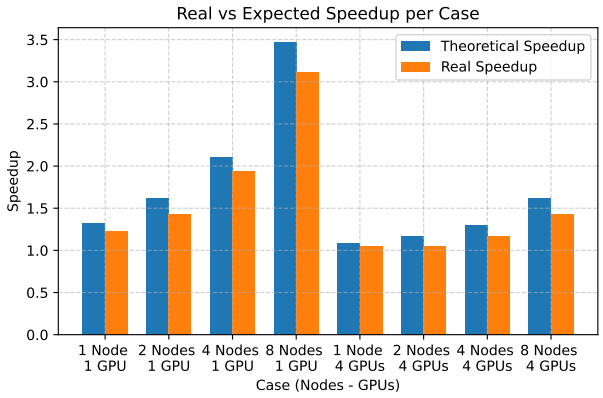

Figure 8: Comparison of the theoretical optimum to the observed real optimum for speedup.

4.5 Performance Model Accuracy

So far we have analyzed the advantages of our hybrid inference approach based on an empirically found optimal CPU fraction $\tilde{\alpha}$. Since the optimal CPU fraction depends on the given deep learning model and heterogeneous hardware allocation, it is infeasible to always evaluate the whole spectrum $0 \leq \alpha \leq 1$ to empirically find an optimal $\tilde{\alpha}$ in practice. In Section 3.2 we derived a formal model to determine the optimal α^* analytically. In this section, we evaluate the accuracy of this model by comparing the theoretical optimal α^* with the empirically found optimal $\tilde{\alpha}$.

For the different heterogeneous hardware configurations investigated in this work, Figure 7 shows the comparison of the theoretically determined optimal CPU fraction α^* compared to the empirically found optimal CPU fraction $\tilde{\alpha}$ and Figure 8 shows the related theoretical speedup S_{α^*}, see Equation (14), compared to the empirically found actual speedup on the right. Determining the theoretical optimal CPU fraction α^* requires the term $T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}})$ from profiling, see Equation (13). Due to the technical limitation of the AIxeleratorService library, that only allows clients and not servers to perform hybrid inference on their allocated CPU core, this term needs modification. In this work, a normalized term $\hat{T}_{\text{fwd}}^{\text{CPU}}(N_{\text{total}}) = \frac{N_{\text{ranks}}}{N_{\text{C}}} T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}})$, that scales the profiled runtime according to the number of clients N_{C} and total number of solver processes N_{ranks}, was used to determine α^*.

In most cases our hybrid model derived in Section 3.2 overestimates the optimal CPU fraction α^* compared to the empirically found optimum $\tilde{\alpha}$. In the 4 and 8 node cases with 1 GPU, the real optima $\tilde{\alpha} = 0.54$ and $\tilde{\alpha} = 0.74$ are underestimated by the theoretical $\alpha^* = 0.526$ and $\alpha^* = 0.712$ respectively. Over all cases the highest absolute error is 0.04 for the 1 node with 1 GPU case and the lowest absolute error is 0.008 for the case with 4 nodes and 4 GPUs.

The speedup is generally overestimated by our theoretical model, because our assumptions from Equation (11) regarding the overhead term do not hold in reality, see Section 4.2. The lowest absolute error of 0.04x occurs in the case with 1 node and 4 GPUs while the highest absolute error of 0.35x occurs in the case with 8 nodes and 1 GPU, respectively. In relative terms the error in the predicted speedup ranges from 3.7% up to 13.2%.

In practice the accuracy of the theoretical optimal CPU fraction α^* is the important one, as users of the AIxeleratorService library need to provide an accurate estimate of this value to profit from the hybrid inference approach and avoid extensive manual tuning. In most cases the predicted optimal CPU fraction α^* is only off by 0.01 or 0.02. Especially in cases with a lot of potential for hybrid inference due to a high ratio of CPU cores per allocated GPU device, the actual optimal CPU fraction $\tilde{\alpha}$ does not need to be hit exactly. For example in the 8 node and 4 GPU case the whole range $0.3 \leq \alpha \leq 0.4$ around the optimal fraction $\tilde{\alpha} = 0.35$ still yields a speedup of at least 1.25x compared to the naive GPU-offloaded execution, see Figure 5. Thus, we claim that our theoretical performance model to derive a near optimal work split fraction α^* is accurate enough for practical real world applications.

5 Future Work

Future work is dedicated towards improving the accuracy and practical usability of the theoretical model. We suspect, that the real optimal CPU fraction $\tilde{\alpha}$ can be found fully automated within the AIxeleratorService library by the following iterative procedure. Let α_k denote the CPU fraction, that was chosen for the k-th invocation of the hybrid inference call. If $0 < \alpha_k < 1$, the runtime terms $T_{\text{fwd}}^{\text{CPU}}(N_{\text{CPU}})$, $T_{\text{fwd}}^{\text{GPU}}(N_{\text{GPU}})$, and $T_{\text{OH}}(N_{\text{GPU}})$ could be measured within the AIxeleratorService library to derive

$$T_{\text{fwd}}^{\text{CPU}}(N_{\text{total}}) = T_{\text{fwd}}^{\text{CPU}}(N_{\text{CPU}})/\alpha_k \tag{15}$$

$$T_{\text{fwd}}^{\text{GPU}}(N_{\text{total}}) = T_{\text{fwd}}^{\text{GPU}}(N_{\text{GPU}})/(1-\alpha_k) \tag{16}$$

$$T_{\text{OH}}(N_{\text{total}}) = T_{\text{OH}}(N_{\text{GPU}})/(1-\alpha_k) \tag{17}$$

based on Equation (11) yielding a current estimate α_k^* according to Equation (13). For the next hybrid inference invocation $k+1$, the previous estimate α_k^* can be used to update $\alpha_{k+1} = \alpha_k + \Delta\alpha_k$, where $\Delta\alpha_k = \alpha_k^* - \alpha_k$ in the simplest case. Preliminary results indicate, that this iterative update procedure might converge to the real optimum $\tilde{\alpha}$. Table 2 shows the derived α_k^* values for each individual sample α_k obtained from executing the coupled solver on 8 nodes with 4 GPUs allocated, see Figure 5. In this case the observed optimal CPU fraction was $\tilde{\alpha} = 0.35$, see Figure 5. One may recognize, that the closer α_k gets to $\tilde{\alpha}$, the closer the estimate α_k^* gets to $\tilde{\alpha}$ as well.

α_k	α_k^*	α_k	α_k^*	α_k	α_k^*
0.10	0.1820	0.34	0.3473	0.4	0.3443
0.20	0.2887	0.35	0.3473	0.5	0.3474
0.30	0.3228	0.36	0.3413	0.6	0.3460
0.31	0.3319	0.37	0.3435	0.7	0.3601
0.32	0.3345	0.38	0.3534	0.8	0.3816
0.33	0.3473	0.39	0.3452	0.9	0.4158

Table 2: Theoretical optimal CPU fraction α^* evaluated on single samples from the 8 node 4 GPU case.

In this example, the iterative update rule starting with $\alpha_0 = 0.10$ converges to a fixpoint already after 6 iterations:

$$\begin{aligned}
&(\alpha_0 = 0.10, \alpha_0^* = 0.1820 \approx 0.20)\\
\rightarrow&(\alpha_1 = 0.20, \alpha_1^* = 0.2887 \approx 0.30)\\
\rightarrow&(\alpha_2 = 0.30, \alpha_2^* = 0.3228 \approx 0.32)\\
\rightarrow&(\alpha_3 = 0.32, \alpha_3^* = 0.3345 \approx 0.33)\\
\rightarrow&(\alpha_4 = 0.33, \alpha_4^* = 0.3473 \approx 0.35)\\
\rightarrow&(\alpha_5 = 0.35, \alpha_5^* = 0.3473 \approx 0.35)\\
\rightarrow&(\alpha_6 = 0.35, \alpha_6^* = 0.3473 \approx 0.35).
\end{aligned}$$

These results encourage further investigation to find the best update rule yielding the fastest convergence up to a small tolerance to fully eliminate the current user's burden and improve the practical usability of hybrid inference.

6 Conclusions

In this work we optimized the computationally expensive inference of a Transformer model, that was previously coupled with the highly parallel finite-volume solver on structured grids from the multi-physics PDE solver framework m-AIA using the open-source AIxleratorService library, to accelerate the costly time-marching of turbulent flow fields in a turbulent boundary layer simulation case. In this workflow, we identified a load imbalance between the allocated heterogeneous GPU and CPU resources caused by offloading of all data samples to the GPU. We proposed a hybrid inference approach, that splits the work between the available GPU and CPU resources to circumvent this issue and optimize the utilization of the allocated heterogeneous hardware, and derived an analytical model to determine the optimal hybrid work distribution. The proposed hybrid inference scheme was implemented into the AIxeleratorService library. The AIxeleratorService[5] library, the partially ported C++-ML-Module[6], and the coupled m-AIA solver code are publicly available on GitHub[7].

The applicability of our proposed hybrid inference approach to productive real-world application cases and its scalability was demonstrated by the example of the coupled m-AIA solver. The speedup yielded by our hybrid inference approach was found to depend on the ratio of allocated CPU cores to GPU devices and the highest investigated ratio of 768 CPU cores to 1 GPU device resulted in a speedup of more than 3x. Even in cases with the smallest possible ratio of 24 CPU cores per 1 GPU device on the given heterogeneous cluster architecture, a speedup of 1.05x was achieved. Of course the actual numbers might differ on different hardware, but we suspect the general trend to hold independent of the actual hardware as the hybrid work split depends mostly on the achieved inference performance of the employed heterogeneous hardware. Moreover, a reduction in energy consumption was found to directly correlate with the runtime reduction by our hybrid inference approach rendering the hybrid execution more energy efficient because a constant amount of worked is inferred in less runtime. Considering that highly parallel CFD simulation cases are potential candidates to target upcoming exascale supercomputers, even small reductions in the order of 5% regarding runtime and energy consumption may pay off in absolute terms for users and operators of these systems. The theoretical optimal hybrid work split derived by our analytical model was found to yield accurate estimates that in the worst case missed the real optimum by up to 4%. In most cases, users can still expect significant near optimal runtime improvements by using a hybrid work split, that is 1% or 2% lower than the analytically derived optimum. To derive the analytical estimate for their given application, users currently need to perform only two exemplary runs of their coupled solver application. In future work, the AIxeleratorService library may automatically find the best hybrid work distribution iteratively during the first few time steps of a long running simulation.

Finally, the results of this work encourage HPC users with similar AI-enhanced CFD simulation cases to reconsider their allocation of heterogeneous hardware resources. The results shown in Figure 5 demonstrate that in our case the same workload can be equally solved by fully offloading the inference workload to a single GPU or by performing the inference fully on 4 CPU nodes. In general GPUs are an expensive and limited resource, especially on university and national clusters like CLAIX-2023. In practice, users might reduce the scheduling time of their jobs on HPC clusters by allocating more CPU resources, that are plentiful available compared to GPU resources. Moreover, if GPUs cannot be utilized for near 100% of the runtime because the remaining CFD solver parts are still executed on CPUs, a sensible resource usage might be to limit the allocation to a few GPU devices compared to the number of CPU cores and use a hybrid inference approach as demonstrated in this work.

[5]https://github.com/RWTH-HPC/AIxeleratorService/tree/MMCP_2026
[6]https://github.com/RWTH-HPC/CPP-ML-Interface
[7]https://github.com/RWTH-HPC/MMCP_2026_Artifact_Hybrid_Inference

Acknowledgments

The authors gratefully acknowledge the computing time provided to them on the high-performance computer CLAIX-2023 at the NHR Center RWTH Aachen University. This is funded by the German Federal Ministry of Research, Technology and Space and the state government of North Rhine-Westphalia. The computations for this research were performed using computing resources under projects rwth1859 and p0025821.

References

[1] Marian Albers. 2021. *Numerical Analysis of Active Drag Reduction for Turbulent Airfoil Flow* (1. auflage ed.). Verlag Dr. Hut, München.
[2] Marian Albers, Pascal S. Meysonnat, Daniel Fernex, Richard Semaan, Bernd R. Noack, and Wolfgang Schröder. 2020. Drag Reduction and Energy Saving by Spanwise Traveling Transversal Surface Waves for Flat Plate Flow. *Flow, Turbulence and Combustion* 105, 1 (June 2020), 125–157. doi:10.1007/s10494-020-00110-8

[3] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*. ACM Press, Atlantic City, New Jersey, 483. doi:10.1145/1465482.1465560

[4] Geveen Arumapperuma, Nicola Sorace, Matthew Jansen, Oliver Bladek, Ludovico Nista, Shreyans Sakhare, Lukas Berger, Heinz Pitsch, Temistocle Grenga, and Antonio Attili. 2025. Extrapolation Performance of Convolutional Neural Network-Based Combustion Models for Large-Eddy Simulation: Influence of Reynolds Number, Filter Kernel and Filter Size. *Flow, Turbulence and Combustion* (March 2025). doi:10.1007/s10494-025-00643-w

[5] Andrea Beck and Marius Kurz. 2023. Toward Discretization-Consistent Closure Schemes for Large Eddy Simulation Using Reinforcement Learning. *Physics of Fluids* 35, 12 (Dec. 2023), 125122. doi:10.1063/5.0176223

[6] Julian Bissantz, Jeremy Karpowski, Matthias Steinhausen, Yujuan Luo, Federica Ferraro, Arne Scholtissek, Christian Hasse, and Luc Vervisch. 2023. Application of Dense Neural Networks for Manifold-Based Modeling of Flame-Wall Interactions. *Applications in Energy and Combustion Science* 13 (March 2023), 100113. doi:10.1016/j.jaecs.2023.100113

[7] Intel Corporation. 2024. *4th Gen Intel Xeon Processor Scalable Family, Codename Sapphire Rapids.* Data Sheet Volume 1 Rev. 1.0.

[8] R. Courant, K. Friedrichs, and H. Lewy. 1928. Über die partiellen Differenzengleichungen der mathematischen Physik. *Math. Ann.* 100, 1 (Dec. 1928), 32–74. doi:10.1007/BF01448839

[9] Benet Eiximeno, Marcial Sanchis-Agudo, Arnau Miró, Ivette Rodriguez, Ricardo Vinuesa, and Oriol Lehmkuhl. 2025. On Deep-Learning-Based Closures for Algebraic Surrogate Models of Turbulent Flows. *Journal of Fluid Mechanics* 1020 (Oct. 2025), A36. doi:10.1017/jfm.2025.10610

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., USA.

[11] Tom Hilgers, Fabian Orland, Fabian Hübenthal, Rakesh Sarma, Andreas Lintermann, and Christian Terboven. 2025. Evaluating the Computational Performance and Accuracy of a Coupled CFD Solver-ML Workflow. In *36th Parallel CFD International Conference 2025.* Merida, Yucatan, Mexico, accepted for publication.

[12] Ekhi Ajuria Illarramendi, Michaël Bauerheim, and Bénédicte Cuenot. 2022. Performance and Accuracy Assessments of an Incompressible Fluid Solver Coupled with a Deep Convolutional Neural Network. *Data-Centric Engineering* 3 (Jan. 2022), e2. doi:10.1017/dce.2022.2

[13] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. 2001. LogGPS: A Parallel Computational Model for Synchronization Analysis. *SIGPLAN Not.* 36, 7 (June 2001), 133–142. doi:10.1145/568014.379592

[14] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: Efficient CPU-GPU Co-Execution for Deep Learning Inference on Mobile Devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22).* Association for Computing Machinery, New York, NY, USA, 209–221. doi:10.1145/3498361.3538932

[15] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. 2021. Machine Learning–Accelerated Computational Fluid Dynamics. *Proceedings of the National Academy of Sciences* 118, 21 (May 2021), e2101784118. doi:10.1073/pnas.2101784118

[16] Marius Kurz, Andrea Beck, and Benjamin Sanderse. 2025. Harnessing Equivariance: Modeling Turbulence with Graph Neural Networks. arXiv:2504.07741 [physics] doi:10.48550/arXiv.2504.07741

[17] Nan Li, Alexandros Iosifidis, and Qi Zhang. 2022. Collaborative Edge Computing for Distributed CNN Inference Acceleration Using Receptive Field-Based Segmentation. *Computer Networks* 214 (Sept. 2022), 109150. doi:10.1016/j.comnet.2022.109150

[18] Yi-Chien Lin, Gangda Deng, and Viktor Prasanna. 2024. A Unified CPU-GPU Protocol for GNN Training. In *Proceedings of the 21st ACM International Conference on Computing Frontiers (CF '24).* Association for Computing Machinery, New York, NY, USA, 155–163. doi:10.1145/3649153.3649191

[19] Weishuo Liu, Ziming Song, and Jian Fang. 2023. NNPred: Deploying Neural Networks in Computational Fluid Dynamics Codes to Facilitate Data-Driven Modeling Studies. *Computer Physics Communications* 290 (Sept. 2023), 108775. doi:10.1016/j.cpc.2023.108775

[20] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 45–55.

[21] Ludovico Nista. 2024. Influence of Adversarial Training on Super-Resolution Turbulence Reconstruction. *Physical Review Fluids* 9, 6 (2024). doi:10.1103/PhysRevFluids.9.064601

[22] Institute of Aerodynamics. 2024. M-AIA. Zenodo. doi:10.5281/zenodo.13350586

[23] Fabian Orland, Kim Sebastian Brose, Julian Bissantz, Federica Ferraro, Christian Terboven, and Christian Hasse. 2022. A Case Study on Coupling OpenFOAM with Different Machine Learning Frameworks. In *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S).* 7–12. doi:10.1109/AI4S56813.2022.00007

[24] Fabian Orland, Ludovico Nista, Nick Kocher, Joris Vanvinckenroye, Heinz Pitsch, and Christian Terboven. 2025. Efficient and Scalable AIxeleration of Reactive CFD Solvers Coupled with Deep Learning Inference on Heterogeneous Architectures. In *Proceedings of the 2025 International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPC Asia '25 Workshops).* Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/3703001.3724386

[25] Sam Partee, Matthew Ellis, Alessandro Rigazzi, Andrew E. Shao, Scott Bachman, Gustavo Marques, and Benjamin Robbins. 2022. Using Machine Learning at Scale in Numerical Simulations with SmartSim: An Application to Ocean Climate Modeling. *Journal of Computational Science* 62 (July 2022), 101707. doi:10.1016/j.jocs.2022.101707

[26] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21).* 551–564.

[27] Rakesh Sarma, Fabian Hübenthal, Eray Inanc, and Andreas Lintermann. 2024. Prediction of Turbulent Boundary Layer Flow Dynamics with Transformers. *Mathematics* 12, 19 (Jan. 2024), 2998. doi:10.3390/math12192998

[28] Anass Serhani, Victor Xing, Dorian Dupuy, Corentin Lapeyre, and Gabriel Staffelbach. 2024. Graph and Convolutional Neural Network Coupling with a High-Performance Large-Eddy Simulation Solver. *Computers & Fluids* 278 (June 2024), 106306. doi:10.1016/j.compfluid.2024.106306

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[30] Neo Wu, Bradley Green, Xue Ben, and Shawn O'Banion. 2020. Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case. arXiv:2001.08317 [cs] doi:10.48550/arXiv.2001.08317

[31] Victor Xing, Corentin Lapeyre, Thomas Jaravel, and Thierry Poinsot. 2021. Generalization Capability of Convolutional Neural Networks for Progress Variable Variance and Reaction Rate Subgrid-Scale Modeling. *Energies* 14, 16 (Jan. 2021), 5096. doi:10.3390/en14165096

[32] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU Training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22).* Association for Computing Machinery, New York, NY, USA, 4582–4591. doi:10.1145/3534678.3539177