Influence of Mixed Precision on Performance and Accuracy of DNN Training for AI-Accelerated CFD Simulations on NVIDIA Multi-GPU System

Kamil Halbiniak[1][0000−0001−9116−8981], Krzysztof Rojek[1][0000−0002−2635−7345], Roman Wyrzykowski[1][0000−0003−1724−1786], Paweł Gepner[2][0000−0003−0004−1729], and Norbert Meyer[3][0000−0003−4020−5329]

[1] Czestochowa University of Technology, Poland
{khalbiniak, krojek, roman}@icis.pcz.pl
[2] Warsaw Technical University, Poland, pawel.gepner@pw.edu.pl
[3] Poznan Supercomputing and Networking Center, Poland, meyer@man.poznan.pl

Abstract. CFD has become a vital tool for understanding and optimizing fluid flow phenomena in engineering. The recent incorporation of AI into CFD has opened new prospects for faster and more reliable simulations. This work delves into the accuracy of integrating CFD with AI and evaluates its performance on an HPC system using multiple NVIDIA HG200 chips - one of the most powerful GPU-based accelerators for AI.

This research explores the potential of mixed precision techniques with diverse data formats to accelerate the distributed data-parallel training of our DNN model proposed for CFD motorBike simulations. Among the considered formats are *BF16*, *TF32*, and *FP32*. Especial emphasis is given to validating and tuning the accuracy of training concerning the impact of mixed precision methods and partitioning a large training dataset into smaller batches. We aim to understand better how various number formats impact the performance-accuracy trade-off in training DNN models for CFD simulations on modern HPC platforms with multiple GPUs and nodes.

Keywords: HPC · CFD · AI/ML · DNN · mixed precision · GPU · data parallel training · NVIDIA GH200 · performance · accuracy

1 Introduction

Computational fluid dynamics (CFD) has become a keystone for understanding and perfecting fluid flow phenomena in various engineering areas. The recent incorporation of artificial intelligence (AI) into CFD has opened new possibilities for accelerated simulations and has promised enhanced accuracy. This work discusses a key intersection of topics at the core of this field - emerging new number formats, leveraging mixed precision methods in AI, and utilizing high-performance parallel architectures.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

This work is based on recent hardware developments that have dramatically increased the scale of parallelism available for CFD AI-driven simulations, including neural network training [21] leveraging the data parallel (DP) training method. It allows us to accelerate training deep neural networks (DNNs) used in CFD simulations by dividing a large dataset into smaller batches processed simultaneously across multiple computing units like GPUs [22, 20]. Each GPU holds a copy of the model and executes all training steps on its data subset. This technique makes it possible to train large models faster by taking advantage of multiple GPUs. Distributed data-parallel (DDP) training is an advanced form of DP training, assuming computations across multiple nodes and allowing training to scale beyond a single machine [22].

At the same time, additional data parallelism in training can come at certain costs, among which lower accuracy is particularly significant. This effect depends heavily on the specific AI model and is related to the increase in the effective batch size, corresponding to the size of the whole dataset processed by all GPUs of the computing platform [21]. This paper focuses on unraveling the intricacies of performance-accuracy trade-off for DDP training of DNN models using as a use case an AI model based on the variational autoencoder architecture we propose for CFD motorBike simulations. Notably, among a number of aspects influencing this trade-off, we deliberately concentrate on the impact of mixed precision techniques using emerging floating-point number formats such as bfloat16 (BF16) and TensorFloat-32 (TF32). By directing our attention to these techniques and formats, we aim to contribute a distinctive perspective on mixed precision computations, recognizing potentially enormous capabilities integral to their implementation in the state-of-the-art graphics accelerators such as those provided by NVIDIA GH200 chips considered in this paper.

The paper is organized as follows. Related works are discussed in Section 2. Section 3 introduces the AI-accelerated CFD motorBike simulation, including the training dataset and architecture of the DNN model we propose for the simulation. Section 4 outlines key issues of implementing DDP training, along with performance-accuracy trade-off. The overview of HPC hardware and software for DDP training of DNN models, including programming of the training in PyTorch, is provided in Section 5. Section 6 presents the results of the performance and accuracy evaluation of DDP training of our DNN model using mixed precision based on the BF16 format, while Section 8 discusses the performance-accuracy trade-off for the TF32 format versus the BF16 and FP32 formats. Section 8 concludes the paper.

2 Related Works

Recently, there has been a remarkable growth in research exploring integrating AI models within CFD simulations. These efforts aim to expand the efficiency and precision of simulations, thus enabling handling more complicated and realistic problems [26, 11, 2, 18]. Modern AI frameworks, like TensorFlow [1] and Pytorch [10], play a key role in supporting AI-driven simulations on multiple

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

computing platforms, providing code portability with minimum additional effort.

A widespread approach involves using machine learning algorithms to model fluid behavior [16]. Recent works [14, 19] have addressed the increasing computation demand of CFD simulations by implementing generalized AI models to simulate various use cases. It allows achieving lower costs of experiments and faster prototyping/parametrization. For instance, neural networks have been successfully employed to simulate turbulence [4], forecast drag and lift forces on aircraft, and optimize the design of turbulent flow control devices [23]. Furthermore, researchers have delved into leveraging AI models for optimizing CFD simulation parameters and settings [18]. These models are trained on large amounts of data and used to make accurate and efficient predictions, which can be incorporated into HPC simulations [14].

In the realm of CFD simulations the choice of number formats significantly influences both the accuracy and performance of computations, along with memory utilization and overall computational efficiency. A prominent step to optimize the performance of numerical computations has been the reinvention of the mixed precision technique - a combination of lower and higher-precision number formats [9]. It has become a powerful optimization to speed up AI computations, especially deep learning training and inference [12]. Utilizing a mixture of (16-bit| (BF16, TF32) and 32-bit floating-point number formats (FP32) in a model during training aims to run it faster and reduce memory utilization. Accelerating computations and reducing the execution time with lower-precision formats also allows decreasing energy consumption. However, reduced precision may lead to numerical instability in certain computations. Therefore, the optimal choice of floating-point precision depends on the specific application and the trade-off between precision needs, memory constraints, and computational efficiency.

The impact of mixed precision techniques on training and inference efficiency of deep neural networks has been studied in papers [6] and [8]. Regardless of whether the first work uses NVIDIA GPU for experiments and the second one focuses on Intel CPU and GPU, both papers do not consider TF32 format, and what is more significant, they leverage only a single accelerator and do not consider any method of training parallelization such as data parallelism, model parallelism, or pipeline one [3]. This paper delves into the intricacies of incorporating mixed precision computing into the multi-GPU implementation of DNN training based on data parallelism. The reason for engaging parallelization across multiple GPUs in our AI-driven simulatio is that, unlike previous work [8], where a simpler model was used with fewer degrees of freedom, this study on the CFD motorBike simulation scenario adopts an enhanced AI (and larger) model tailored to the increased complexity introduced by the expanded parameter space. The inclusion of three variable parameters - freestream velocity U_∞, yaw angle ψ, and ground clearance h_g - significantly broadens the diversity of the aerodynamic scenarios simulated, necessitating a more robust deep learning architecture capable of capturing the intricate interplay between these factors and the resulting flow fields.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

3 Deep Neural Network for AI-Accelerated CFD Simulation

3.1 CFD simulation of steady flow around a motorcycle and rider

This work investigates the performance and accuracy of training DNN models used in AI-accelerated CFD simulations. It leverages a practical application: the simulation of steady flows around a motorcycle and rider geometry available in OpenFOAM [15] – a versatile open-source CFD software widely used for simulating fluid flows. This tool provides a comprehensive set of solvers for a variety of flow problems, including steady-state and transient simulations.

For this study, we employ the steady-state *simpleFoam* solver, based on the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) [25] algorithm, which is suitable for incompressible, turbulent flows. The solver performs steady-state, incompressible Reynolds-Averaged Navier-Stokes calculations (RANS) over the mesh. In fluid dynamics, the Navier-Stokes equations describe the motion of fluid, taking into account viscosity, pressure, and velocity. The RANS-based approach is a widely used approach in CFD for simulating turbulent flows.

In the considered application, by integrating AI methodologies [17, 8] into the simulation, we aim to streamline the evaluation of crucial fluid dynamic parameters – velocity U and pressure p. This study's motorcycle and rider geometry is a standard benchmark provided in OpenFOAM. It realistically represents the motorbike scenario, and is often used for aerodynamic and drag analysis.

3.2 Training Dataset

The AI model is trained on extensive datasets from RANS simulations, designed to infer and optimize the estimation of flow velocity U and pressure p in a motorcycle aerodynamic analysis. The datasets are constructed through a parametric series of simulations with key physical variables - rider's freestream velocity U_∞ and motorcycle's yaw angle ψ, systematically varied to capture a broad range of operational conditions. An additional parameter, the ground clearance h_g, is introduced to account for the motorcycle's proximity to the ground, influencing boundary layer interactions and wake dynamics. The simulation domain is discretized into a structured mesh with 350,000 hexahedral cells, ensuring sufficient resolution of the near-field flow structures around the rider-motorcycle geometry. Each simulation is evolved over 100 timesteps with duration of $\Delta t = 1{\times}10^{-3}$ s each, using the OpenFOAM finite-volume solver to achieve numerical convergence of the turbulent flow field, modeled with the $k-\omega$ SST turbulence model.

In this study, 25 distinct simulations are performed, each representing a unique combination of U_∞, ψ, and h_g. The velocity U_∞ is varied between 10 m/s and 30 m/s, the angle ψ ranges from $-15°$ to $+15°$ relative to the longitudinal axis, and the clearance h_g is adjusted between 0.05 m and 0.15 m. A sliding window approach is adopted for each simulation with five consecutive timesteps per window. The dataset for a simulation is constructed by designating the flow field data (velocities u_x, u_y, u_z, and pressure p) from the first four timesteps as

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

input features, with the fifth timestep serving as the target output. This temporal sequencing yields 96 samples per simulation (calculated as $100 - 4 = 96$ windows), resulting in a total of $96 \times 25 = 2400$ samples across all simulations.

Each sample contains spatially-resolved field data across the 350k-cell domain, with four variables (u_x, u_y, u_z, p) stored in FP32 format, giving the size of a single sample approximately $350k \times 4 \times 4$ bytes = 5.6 MB, and the total dataset size 2400×5.6 MB ≈13.44 GB. The dataset is divided into training and validation subsets with a ratio of 90% to 10% of the samples to facilitate efficient model training. Before training, the data is preprocessed by normalizing the velocity and pressure fields concerning their global maxima across the dataset, ensuring numerical stability and improving the deep learning framework's convergence.

3.3 Architecture of Deep Neural Network used in CFD Simulations

Our AI model for predicting iteration results in motorBike CFD simulations leverages a variational autoencoder (VAE) architecture [24], chosen for its flexibility and scalability in handling complex computational tasks. The VAE's capabilities are particularly well-suited for CFD for two primary reasons. First, its efficient and scalable design facilitates the processing of extensive datasets, a common requirement in CFD simulations involving intricate geometries like motorbikes. Second, VAEs' unsupervised learning capability enables them to capture latent patterns and structures in CFD data, making them adept at modeling the nonlinear and dynamic nature of fluid flows.

In this work, the VAE is used to predict critical CFD quantities such as pressure and velocity around a motorbike geometry. Our approach employs a ML pipeline with four iterations of CFD simulation, which produces the input data for the VAE and generates a single predicted output. Each input corresponds to q quantities (e.g., pressure, velocity) across n domain cells, represented as an array of size $timesteps \times n \times q$. The output is represented as an array of size $1 \times n \times 1$, predicting the last time step's pressure distribution. The VAE model includes an encoder with four Conv2D and a decoder with four Conv2DTranspose layers. The encoder compresses the high-dimensional input into a latent representation, and the decoder reconstructs the predicted output (Algorithm 1). This setup efficiently models the dynamic interactions within fluid flows around the motorbike geometry, enabling accurate predictions over time. In Algorithm 1, the Kullback-Leibler (KL) divergence quantifies the difference between two probability distributions. In variational autoencoders, it acts as a regularization term, ensuring the latent distribution aligns with a prior one (e.g., standard normal), promoting a structured and efficient latent space for inference. Hyperparameters like the number of filters, kernel sizes, and latent space dimensionality were determined empirically through grid search, optimizing for validation loss.

4 Distributed Data Parallel Training of DNN

4.1 Key Issues of Implementing DDP Training

In DDP training, multiple processes are launched on various machines with each process usually assigned to a single GPU. Training DNN models rely on

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

Algorithm 1 VAE Architecture for Motorbike CFD Simulations

Input: Data (timestep, n, q)
Output: Predicted output $(1, n, 1)$
1 **Encoder:**
Conv2D: filters=8, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2D: filters=16, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2D: filters=32, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2D: filters=64, kernel_size=(3,3), strides=(2,2), activation='relu'
Flatten output to 1D
Dense Layer: units=latent_dim, activation='linear'
2 **Latent Representation:**
z = mean + std_dev
3 **Decoder:**
Dense Layer: units=(encoded size), activation='relu'
Reshape back to 3D
Conv2DTranspose: filters=32, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2DTranspose: filters=16, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2DTranspose: filters=8, kernel_size=(3,3), strides=(2,2), activation='relu'
Conv2DTranspose: filters=1, kernel_size=(3,3), strides=(2,2), activation='linear'
4 **Loss Function:**
Loss = Reconstruction Loss (MSE) + KL Divergence

executing iteratively three steps: (i) the forward pass to compute loss, (ii) the backward pass to compute gradients, and (iii) the optimizer step to update parameters [22]. Each process performs the forward pass independently on separate partitions of the training dataset to calculate the local gradients. Since DDP training and traditional local training must be mathematically equivalent, models for all processes have to be synchronized at each training iteration by sharing and averaging the local gradients in order to compute the global gradient used by each process to update the model [22, 27]. As a result, all processes have the same model state after each training iteration.

DDP training is implemented by many AI frameworks, including PyTorch and Horovod [20, 22]. While they rely on gradient synchronization across processes, their implementations differ. Horovod performs gradient averaging after all processes complete the backward pass [20], implementing *AllReduce* operation between gradient calculations and model update. The original optimizer is wrapped with a new one called Distributed Optimizer, which delegates the gradient computation to the local optimizer, and averages gradients based on *AllReduce* operation [27]. The averaged gradients are used to update the model for each process. Opposite to Horovod, the gradient averaging in PyTroch is performed during the local backward pass using *AllReduce* operation. PyTorch uses a technique called *layer bucketing* [22] to optimize the gradient synchronization performance. It groups parameters from several layers into a larger *bucket*. Once the backward pass for the layers in a bucket is performed, this bucket is syn-

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

chronized in a single AllReduce operation, so *AllReduce* operation on gradients can start before the local backward pass finishes. Bucketing allows overlapping communication and computations [22]. While the backward computations for bucket *i+1* are performed, the gradients of bucket i are exchanged.

4.2 Performance Versus Accuracy for DDP Training

For the DDP training, we will distinguish between the batch size B_S corresponding to the size of subsets processed by each GPU and the effective batch size B_{Ef}, which corresponds to the size of the whole dataset processed by all p GPUs of the computing platform, where $B_{Ef} = pB_S$. When using only a single GPU, the rule of thumb for selecting the first parameter is to choose the maximum value at which all calculations are still performed in GPU memory. This way, we can maximize hardware efficiency (utilization) and training performance. For DDP training on multiple GPUs, the statistical accuracy of the model training has to be considered in addition to the performance when selecting the second parameter. Following work [3], the size B_{Ef} should not be too small to harness inherent concurrency in the evaluation of the loss function, nor should they be too large, as the quality of the result decays once increased beyond a certain point. Typical batch sizes, where the training error is stably close to the minimum, lie in a range between the orders of 10 and 10,000 [3]. The boundary points of this range depend heavily on the specific model.

Consequently, setting the batch size B_{Ef} results in a complex optimization space. Increasing B_{Ef} allows us to escalate the parallelism of computations by expanding the GPU number. At the same time, large batch sizes can negatively affect training convergence. Among the methods of overcoming this problem, the most popular is adjusting learning rates statically or adaptively, including the usage of specific learning-rate schedules [3]. In particular, when multiplying the batch size by p, it was recommended to multiply the learning rate by $\sqrt{p}$. Linear scaling is also frequently used in practice. However, the efficiency of these large-batch methods is again highly dependent on the specific model.

5 Overview of HPC Hardware and Software for Distributed Data Parallel DNN Training

5.1 Overview of Multi-GPU System with NVIDIA GH200

NVIDIA GH200 Superchip was designed for large-scale AI and HPC tasks, based on heterogenous Grace Hooper architecture [13]. It combines the high-performance capabilities of the NVIDIA Hopper H100 GPU with the versatility of the NVIDIA Grace CPU based on Neoverse V2 Armv9 architecture. It has 72 cores and up to 480GB of LPDDR5X memory. The CPU and GPU are connected via the high-bandwidth NVIDIA NVLink-C2C delivering up to 900GB/s of bandwidth (about 7 times higher than PCIe Gen5 connections). All GH200 devices within a node are coupled through NVIDIA NVLink Switch System, which enables accessing peer memory using direct loads, stores, and atomic operations.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

The GPU chip includes 16,896 FP32 cores providing up to 67 TFlop/s of theoretical peak performance. Depending on the version, the GPU has 96GB of HBM3 or 144 GB of HBM3e memory with a bandwidth of up to 4.9 TB/s. Additionally, it includes 528 fourth-generation Tensor Cores designed to accelerate matrix operations essential for AI and HPC. The cores support various precisions, including FP64, FP32, BF16, TF32, and newly introduced FP8 [13]. The GPU achieves remarkable performance across different precision levels. The theoretical peak performance (without sparsity) for TF32 and BF16 formats equals 494 and 990 TFlop/s, respectively. Significant performance gains of BF16 and TF32 over FP32 justify using mixed precision techniques [6, 8], where certain operations, such as weight updates in deep learning, leverage FP32 precision to maintain stability and accuracy. Their presence explains why the actual performance gain over FP32 is lower than the theoretical maximum.

5.2 Programming DDP Training in PyTorch

PyTorch is a machine learning framework that provides tools for building and training AI/ML models [10]. Developed originally by Meta AI, since 2022 it has been developed by PyTorch Foundation. PyTorch is one of the most popular deep learning frameworks, alongside others such as TensorFlow or Keras. The framework ensures modules and classes including `torch.nn`, `torch.optim`, `Dataset`, and `DataLoader` that allow create and train neural networks [10]. While the `torch.nn` provides components to define neural network architectures with layers, loss functions, and other components, the `torch.optim` implements optimization algorithms (like SGD, Adam, ...) that adjust the model's parameters based on gradients computed during training. The `Dataset` and `DataLoader` modules provide efficient data handling during model training.

The basic building blocks for deep learning and numerical computations in PyTorch are tensors [10]. They are specialized data structures that are very similar to arrays and matrices. PyTorch tensors can be used on both CPUs, GPUs and other accelerators, enabling fast computations.

PyTroch includes `torch.nn.parallel.DistributedDataParallel` module for DDP training on machines with multiple computing units. Listing 1.1 presents a code snippet implementing DDP training in PyTorch. It begins by initializing distributed training. Next, it prepares the dataset by using `DatasetCFD` class and splitting it into the training and validation datasets. The `DatasetCFD` inherits from PyTorch `Dataset` class. In lines 6-7, the DNN model (`NetCFD` class) is created and wrapped with `DistributedDataParallel`, which enables gradient synchronization between processes. The loss function (`MSELoss`) and optimizer (`Adam`) are defined to guide the training process. The data loader (`DataLoader`) is set up with `DistributedSampler` (Lines 11-12), which ensures that data are evenly distributed across multiple GPUs. The training loop (Lines 13-21) iterates over epochs, adjusting the sampler per epoch (Line 14), and performing the training steps (Lines 15-21), in which the model processes inputs, computes the loss, backpropagates gradients, and updates weights using the optimizer.

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

```
# import and initialize required PyTorch packages
# ...
dataset = DatasetCFD()
train_dataset,valid_dataset = random_split(dataset, ...)
# ...
model = NetCFD()
model = DistributedDataParallel(model)
loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# ...
distSampler = DistributedSampler(train_dataset, ...)
trainLoader = DataLoader(train_dataset, batch_size=bs,
    sampler=distSampler)
for epoch in range(epochs):
   trainLoader.sampler.set_epoch(epoch)
   for inputs, targets in trainLoader:
        inputs, targets = inputs.to(...), targets.to(...)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()
    # Rest of code including calculaton of loss curve
```

Listing 1.1: Distributed Data Parallel Training in PyTorch

6 Performance and Accuracy Evaluation of Distributed Data Parallel DNN Training: Using BF16 Format

6.1 Evaluation Metodology

The evaluation is performed on the Helios supercomputer installed at ACC Cyfronet AGH [5]. The tests are executed on four nodes with four NVIDIA Grace Hopper GH200 devices each (16 in total). The Slingshot interconnect with a bandwidth of 200 Gb/s provides connections between nodes (using dragonfly topology) and inside them. The NVLink Switch System connects GH200 devices within a single node, based on NVLink 4 and NVLink-C2C.

The whole application's code is written in Python (version 3.11.5), while the DNN training is implemented in PyTroch 2.3.1. Since the GH200 device is based on the ARM architecture, we have to compile PyTorch from the source. The software stack also contains NVIDIA CUDA SDK 12.4.0.

This paper is focused on training the model with the VAE architecture of Section 3.3. The model includes 93,319 trainable parameters in FP32 format, so over 10 GB of memory is required to hold the whole model. This model is more than 5.5 times larger than a simpler model (with only 16,583 parameters), considered in our work [8], trained on a single H100 GPU. For improving the performance of training while preserving the required accuracy, the mixed precision approach is leveraged based on either BF16 format (Section 6) or TF32

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

format (Section 7). PyTorch Automatic Mixed Precision (`torch.amp`) package is employed to enable mixed-precision computations with BF16.

In this section, the performance and accuracy of parallel DNN training is evaluated in the following three scenarios:

1. *Scenario 1: Benchmarking the scalability without considering accuracy:* the fixed number $N_E = 50$ of training epochs is executed on various GPU numbers, with the batch size $B_S = 16$ set corresponding to the maximum portion of the dataset that ensures all computations fit into the GPU memory.
2. *Scenario 2: Investigating the scalability of training with considering accuracy*: unlike the previous scenario, training is performed for different numbers of epochs until the loss reaches a value equal to that obtained for a single GPU (with a tolerance of 10%).
3. *Scenario 3: Analyzing the scalability of training when batch sizes B_S and B_{Ef} are selected concerning the performance and accuracy trade-off.*

6.2 Performance and Accuracy Results

Scenario 1: Scalability without considering accuracy
Table 1 shows the execution times T_p for this scenario, obtained on different numbers p of GPUs, as well as the speedup achieved against a single GPU and the loss value after executing all training epochs.

The performance results achieved for Scenario 1 shows that increasing the number of GPUs significantly reduces the execution time of DNN training, with nearly linear scaling and efficiency remaining high across different GPU configurations. Using all GPUs allows us to accelerate the training 14.6 times (89% of ideal scalability). However, while training performance increases significantly, the accuracy decreases when employing more GPUs. The final loss remains relatively stable when using up to 4 GPUs, but it rises significantly for 8 and 16 GPUs (0.211 and 0.278, respectively). This behavior suggests that, in our case, selecting a large size B_{Ef} negatively affects the training convergence.

The DNN model is trained with a learning rate $\lambda = 0.001$ in the tests. For overcoming the drop in training accuracy on 8 and 16 GPUs, we investigated the methods for adjusting learning rates mentioned in Section 4, including multiplying by $\sqrt{p}$, as well as PyTorch Linear, and Step schedulers. However, they do not provided the intended effect and even drop the accuracy of the training.

Scenario 2: Scalability with considering accuracy
In this scenario, training is performed only for 8 and 16 GPUs as yielding noticeably lower accuracy in Scenario 1. The tests show that $N_E = 74$ and $N_E = 94$ epochs are required to reach the loss value achieved for a single GPU. However, increasing N_E decreases the training performance significantly.

Scenario 3: Seting batch size concerning performance and accuracy trade-off
This scenario starts with assuming the same effective batch size $B_{Ef} = 16$ as B_S for a single GPU, regardless of the number of GPUs. This allows us to perform training for the fixed number $N_E = 50$ of epochs with the batch size $B_S = B_{Ef}/p$ varied across GPU configurations. The obtained results (Table 1)

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

Table 1: Performance and accuracy obtained for BF16

	1×GH200	2×GH200	4×GH200	8×GH200	16×GH200
Scenario 1					
T_p [s]	1274	657	336	175	90
Final loss	0.141	0.149	0.148	0.211	0.278
S_p	1	1.94	3.79	7.28	14.16
Scenario 2					
T_p [s]	-	-	-	260	172
S_p	-	-	-	4.9	7.41
Scenario 3: $B_{Ef} = 16$					
T_p [s]	1274	764	396	217	129
Final loss	0.141	0.14	0.144	0.142	0.146
S_p	1	1.67	3.22	5.87	9.88
Scenario 3: $B_{Ef} = 64^*$					
T_p [s]	-	-	-	199	110
Final loss	-	-	-	0.15	0.152
S_p	-	-	-	6.4	11.58

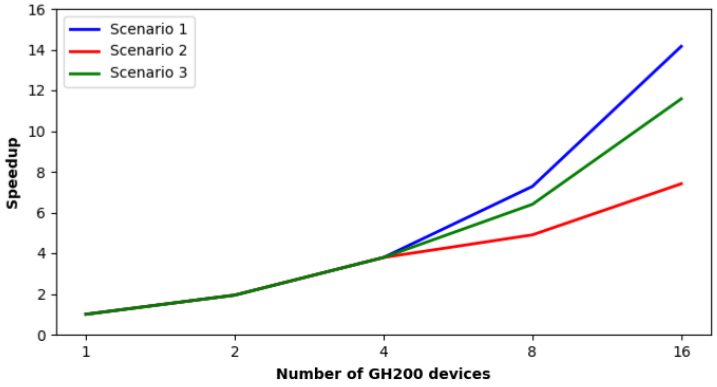

Fig. 1: The comparison of scalability achieved for different scenarios with BF16 on the multi-GPU system using NVIDIA GH200 chips.

indicate a better scalability than Scenario 2 while providing the desired accuracy. For example, using all GPUs permits us to accelerate training by 9.88 times.

At the same time, the results achieved for Scenario 1 show a relatively stable loss for up to 4 GPUs with the effective batch size $B_{Ef} \leq 4 * 16 = 64$. So, it is rational to set $B_{Ef} = 64$ for 8 and 16 GPUs as well, improving scalability considerably with the speedup of 6.4 and 11.58 for 8 and 16 GPUs, respectively. These performance gains come at slightly higher loss but within the tolerance.

Fig. 1 compares scalability achieved for all three scenarios. While Scenario 1 provides the best scalability, it delivers lower accuracy for configurations with 8 and 16 GPUs. The opposite is true for Scenario 2, which provides the desired

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

accuracy but at the cost of performance. Finally, Scenario 3 achieves a reasonable speedup for all numbers of GPUs while maintaining the desired accuracy.

7 Performance-Accuracy Trade-off for TF32 Format versus BF16 and FP32 Formats

Table 2 shows the execution time and accuracy obtained for mixed precision training with TF32 format for $N_E = 50$ epochs. This table also includes the achieved scalability, which is further visualized in Fig. 2. It is worth noting that for implementing mixed precision computations with TF32 format, we have to employ `torch.backends` module instead of `torch.amp`. This module contains a set of routines and variables allowing for control of the code execution on the CUDA platform, including `torch.backends.cuda.matmul.allow_tf32` and `torch.backends.cudnn.allow_tf32` flags.

For the TF32 format, the maximum batch size B_S that guarantees all computations fit into GPU memory equals 8. However, the shortest execution time and best accuracy for a single GPU correspond to $B_S = 2$. Besides good scalability (speedup of about 14 times for 16 GPUs), the TF32 format provides better accuracy than BF16 for all numbers of GPUs. For example, on a single GPU and four GPUs with TF32, loss values equal, respectively, 0.118 and 1.12, against 0.141 and 1.148 with BF16, i.e., are 1.19 and 1.23 times lower.

However, the TF32 format requires more computational overheads, which results in 1.42 and 1.22 times longer execution time, compared to Scenario 2 for the BF16 format, on a single GPU and four GPUs, respectively. At the same time, these overheads are counterbalanced by better training accuracy. Consequently, the performance gap between TF32 and BF16 decreases when more GPUs are employed. Finally, for 16 GPUs, TF32 is only 1.17 times slower than BF16, with 1.09 times better accuracy.

Table 2 also shows performance and accuracy achieved without mixed precision when only the fully 32-bit FP32 format is used. The general conclusion is that the accuracy behavior of the mixed precision solution with TF32 is practically the same as that of the full precision option with FP32. At the same time, TF32 allows us to decrease computation time by about 1.25 times compared to FP32 across the considered range of GPU numbers.

Table 2: Performance and accuracy achieved for TF32 and FP32 ($B_S = 2$).

	1×GH200	2×GH200	4×GH200	8×GH200	16×GH200
TF32					
T_p [s]	1805	938	485	250	129
Final loss	0.118	0.114	0.12	0.138	0.139
S_p	1	1.92	3.72	7.22	13.99
FP32					
T_p [s]	2256	1157	603	309	161
Final loss	0.118	0.128	0.129	0.135	0.137
S_p	1	1.95	3.74	7.3	14.01

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

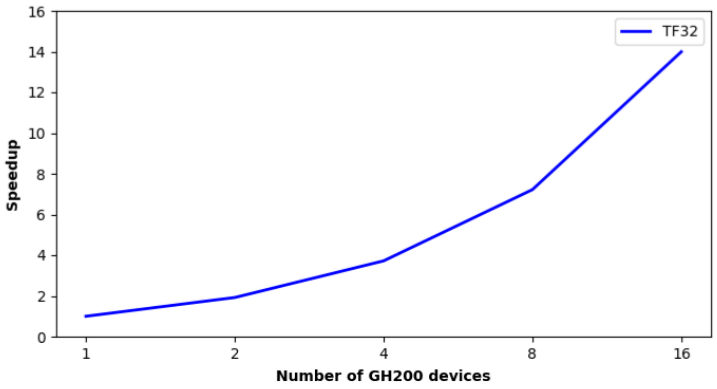

Fig. 2: The scalability achieved for training with TF32 format on the multi-GPU system using NVIDIA GH200 chips. The scalability graph for the FP32 format is practically identical.

8 Conclusion and Future Works

In this paper, we tackle the challenge of efficiently incorporating mixed precision computing into the data parallel implementation of DNN training. Our contributions can be summarized as follows:

1. *Unleashing the potential of mixed precision in distributed data parallel computations on multi-GPU systems to accelerate the training of DNN models.* Investigating the impact of diverse floating-point data formats - BF16, TF32, and FP32 - on the performance and accuracy of training CFD AI models on multi-GPU platforms is a focus of this work. We systematically explore the performance gains and scalability associated with each datatype. Furthermore, we highlight the key aspect of validating and tuning the accuracy of results obtained using these datatypes to ensure their applicability.

2. *Acceleration of training the DNN model based on the variational autoencoder architecture we propose for CFD motorBike simulation.* Incorporating AI-driven methods into CFD simulations, specifically by integrating DNN models with the OpenFOAM open-source software, is a vital step towards more efficient and adaptive fluid flow predictions. Engaging parallelization across multiple GPUs allows us to train effectively a large AI model with a more robust deep learning architecture tailored to the increased complexity introduced by the expanded parameter space of the considered CFD scenario.

3. *Verification on the modern HPC platform with multiple GPUs and nodes.* Extending beyond theoretical considerations, we delve into the practical implementation of our findings with the PyTorch AI framework and the HPC system, including four nodes each with four NVIDIA Grace Hopper GH200

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

superchips - one of the most powerful GPU-based accelerators for AI. In particular, we show that leveraging mixed precision based on the BF16 format on 16 GPUs allows us to accelerate training the model by about 11.6 times, preserving the same loss value as for a single GPU. Besides good scalability (speedup of about 14 times for 16 GPUs), the TF32 format provides better accuracy than BF16 for all numbers of GPUs. However, TF32 requires more computational overheads, which results in 1.42 and 1.22 times longer execution time on a single GPU and four GPUs, respectively. However, these overheads are counterbalanced by better training accuracy, so the performance gap between TF32 and BF16 decreases when more GPUs are employed. For 16 GPUs, TF32 is only 1.17 times slower than BF16, with 1.09 times better accuracy. The important conclusion is that the accuracy behavior of the mixed precision solution with TF32 is practically the same as that of the full precision option with FP32. At the same time, TF32 allows us to speed up computations by about 1.25 times compared to FP32 across the considered range of GPU numbers.

Below, we outline possible directions for future work. The first one involves studying the feasible methods of increasing the effective batch size more systematically without decreasing training accuracy [3]. The second direction concerns exploiting AI accelerators with alternative architecture, such as Intel Habana Gaudi 2 and Gaudi 3 platforms [7]. The last direction has quite a different nature. It relates to incorporating the power of large language models (LLMs) into CFD simulations [28]. The efficient utilization of these models requires harnessing advanced HPC systems with accelerators.

Acknowledgements We gratefully acknowledge Polish high-performance computing infrastructure PLGrid (HPC Center: ACK Cyfronet AGH) for providing computer facilities and support within grant no. PLG/2024/017422.

References

1. Abadi, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proc. 12th USENIX Conf. on Operating Systems Design and Implementation, 265–283 (2016)
2. Barbera, A.G., et al.: AI-Driven Acceleration of Computational Fluid Dynamic Simulations. In: PPAM 2025. LNCS, vol. 15580 (in print) (2025)
3. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Comput. Surv. **52**(4) (2019)
4. Calzolari, G., Liu, W.: Deep learning to develop zero-equation based turbulence model for cfd simulations of the built environment. Indoor/Outdoor Airflow and Air Quality **17**, 399–414 (2024)
5. Helios supercomputer. https://www.cyfronet.pl/en/19951,artykul,helios_supercomputer.html (2024)
6. Dörrich, M., Fan, M., Kist, A.M.: Impact of mixed precision techniques on training and inference efficiency of deep neural networks. IEEE Access **11** (2023)

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11

7. Intel Gaudi 3 AI Accelerator White Paper (2024), https://www.intel.com/content/www/us/en/content-details/817486/intel-gaudi-3-ai-accelerator-white-paper.html
8. Halbiniak, K., Rojek, K., Iserte, S., Wyrzykowski, R.: Unleashing the Potential of Mixed Precision in AI-Accelerated CFD Simulation on Intel CPU/GPU Architectures. In: ICCS 2024. LNCS, vol. 14837, 213-217 (2024)
9. He, X., Sun, J., Chen, H., Li, D.: Campo: Cost-Aware Performance Optimization for Mixed-Precision Neural Network Training. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), 505–518 (2022)
10. Introduction to PyTorch. https://pytorch.org/tutorials/beginner/basics/intro.html (2024)
11. Iserte, S., et al.: Accelerating Urban Scale Simulations Leveraging Local Spatial 3D Structure. Journal of Computational Science **62**, 101741 (2022)
12. Kalamkar, D., et al.: A study of bfloat16 for deep learning training. arXive preprint (2019), https://arxiv.org/abs/1905.12322
13. NVIDIA GH200 Grace Hopper Superchip Architecture (2024), https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper
14. Obiols-Sales, O., et al.: CFDNet: a deep learning-based accelerator for fluid simulations. In: Proc. 34th ACM Int. Conf. on Supercomputing. pp. 1–12 (2020)
15. OpenFOAM Foundation: https://openfoam.org (2025), accessed: 2025-01-20
16. Panchigar, D., et al.: Machine learning-based CFD simulations: a review, models, open threats, and future tactics. Neural Computing and Applications **34**, 21677–21700 (2022)
17. Rojek, K., Gepner, R.W.P.: Chemical Mixing Simulations with Integrated AI Accelerator. In: ICCS 2023. LNCS, vol. 14074, 494–508 (2023)
18. Rojek, K., Wyrzykowski, R.: Performance and scalability analysis of AI-accelerated CFD simulations across various computing platforms. In: Singer, J., Elkhatib, Y., Heras, D., Diehl, P., Brown, N., Ilic, A. (eds.) Euro-Par 2022: Parallel Processing Workshops. pp. 223–234. Springer International Publishing, Cham (2023)
19. Sadrehaghighi, I.: Artificial Intelligence (AI) and Deep Learning for CFD (2022), https://www.researchgate.net/publication/339795951
20. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint (2018), https://arxiv.org/abs/1802.05799
21. Shallue, C.J., et al.: Measuring the Effects of Data Parallelism on Neural Network Training. arXiv preprint (2019), https://arxiv.org/pdf/1811.03600
22. Shen, L., et al.: Pytorch distributed: experiences on accelerating data parallel training. arXive preprint (2020), https://arxiv.org/abs/2006.15704
23. Srivastava, S., Damodaran, M., Khoo, B.C.: Machine Learning Surrogates for Predicting Response of an Aero-structural-sloshing System (2019), https://arxiv.org/pdf/1911.10043
24. Tschannen, M., Bachem, O., Lucic, M.: Recent advances in autoencoder-based representation learning. arXive preprint (2018), https://arxiv.org/pdf/1812.05069
25. Versteeg, H.K., Malalasekera, W.: An Introduction to Computational Fluid Dynamics: The Finite Volume Method. Pearson Education (2007)
26. Vinuesa, R., Brunton, S.L.: The Potential of Machine Learning to Enhance Computational Fluid Dynamics. arXive preprint (2021), https://arxiv.org/pdf/2110.02085
27. Wu, X., et al.: Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks. In: ICPP 2019: Proc. 48th Int. Conf. on Parallel Processing. ACM (2019)
28. Zhu, M., Bazaga, A., Lio, P.: Fluid-llm: Learning computational fluid dynamics with spatiotemporal-aware large language models. arXive preprint (2024), https://arxiv.org/abs/2406.04501

ICCS Camera Ready Version 2025
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-97554-7_11