TUM Data Innovation Lab

Munich Data Science Institute

Technical University of Munich

&

TUM Chair of Aerodynamics and Fluid Mechanics

Final report of project:

Acceleration of Neural Network Training with Microsoft DeepSpeed

Authors	Yamei Chen, Daniel Herbst, Daniel Stoll
Mentor(s)	M.Sc. Ludger Pähler TUM Chair of Aerodynamics and Fluid Mechanics
Project Lead	Dr. Ricardo Acevedo Cabra (MDSI)
Supervisor	Prof. Dr. Massimo Fornasier (MDSI)

Aug 2022

Abstract

Large-scale model training on massive data is a way to achieve top-tier accuracy, but training of such models is difficult due to memory constraints and communication overhead when scaling out to multiple devices. Much of ongoing research in scalable machine learning is currently limited to giant transformer models such as Bert or GPT-3. However, other types of models benefit as well from scaling up the architectures and model parameters, in particular Graph Neural Networks (GNN) for atomic simulations. In our work, we use optimizations from Microsoft DeepSpeed, a toolkit for efficient training of large-scale models, to scale and efficiently train GNNs for predicting energies and forces of catalysts. Our integration of DeepSpeed into GNN architectures includes mixed precision training, offloading of optimizer states and parameters to CPU or non-volatile memory, and most notably the Zero Redundancy Optimizer (ZeRO) to reduce the amount of replicated states during data parallel training. We evaluated our optimizations by increasing the size of the two most successful GNN architectures for atomic simulations, DimeNet and GemNet, to several hundred million parameters and measuring memory usage as well as epoch runtime. As dataset we used catalysts labeled with energy and forces from the OC20 dataset of the Open Catalyst Project. The performed optimization of our approach shows up to 50% less reserved memory and 87% less allocated memory on GemNet. However, we also observe weaknesses in ZeRO such as increased communication costs and slower runtimes when using offloading strategies.

Contents

1 Introduction

1.1 Motivation

Training large-scale models is the Formula 1 of the machine learning research community: Large labs and corporations with almost unlimited resources compete to outperform each other in model size and accuracy improvements to the decimal points. Consequently, in recent years there has been a surge of more efficient and scalable methods to train deep learning models in parallel on multiple GPUs or even multiple machines. Most of the methods are designed for large language models, however also offer general solutions to problems such as linear scalability, memory efficiency and communication reduction. With graph machine learning gaining ever more prominence in physics-focused prediction tasks and simulations, we adopted some of these scaling methods and applied them to machine learning models that operate on graphs.

Graphs offer a powerful way to structure connected data by elevating the relationships in-between data points. Fundamentally a graph consists of a set of nodes, which are connected to each other by a set of edges. Both nodes and edges may contain features to further describe the entity represented by a node or the relationship between entities. Graphs are used in a wide range of application areas such as social analysis by representing people and their relationship to each other as graphs, or in the medical domain by modeling population graphs for disease prediction, or using molecular graphs for drug discovery. Graphs also have a long history of applications in quantum chemistry [Gil+17], representing molecular structures and the energy as well as forces of the structure as graphs.

In recent years, geometric deep learning, a machine learning approach to bring deep learning models to the graph domain, is getting more and more popular since they achieve comparable or even better results than traditional analytical graph algorithms often at much lower computational costs. Graph Neural Networks (GNNs) are currently the dominant architecture for designing geometric deep learning models, most of which are based on the message passing paradigm, where nodes gather features from neighbors, transform them via differentiable functions and scatter to outgoing neighboring nodes. Especially for molecular predictions, GNN-based architectures are orders of magnitudes faster than traditional methods from quantum chemistry. Meta AI introduced the Open Catalyst Project (OCP) [Cha+21] in 2020 to provide unified baseline models and datasets for force and energy predictions of molecular structures, more specificially Catalysts. There has been a number of high-quality GNN based model submissions [KGG20; GBG21; Sri+22] to the project which we will describe further in Section 2.2.

1.2 Problem definition

Although GNNs allow faster prediction of the outcome of molecular simulations, they are still computationally intensive to train. Meta AI used a cluster of 256 GPUs to scale out the training of the most recent top-performing GNN model which contained a few hundred millions of parameters. The training of one configuration of this model still took over 5 years of summed-up GPU runtime. So, although GNN models have been very successful for molecular predictions, the problem remains that training, even on high-end hardware

and multiple GPUs, takes a long time and has extremely high memory requirements. In addition, using hyper-optimization requires many iterations with different parameters to find the optimal model. Any memory and runtime savings will result in days to weeks of less training time and significantly reduced costs.

Since 2020, Microsoft is developing DeepSpeed, which is designed to reduce computing costs and memory requirements of large distributed models through better parallelization and minimal state replication. DeepSpeed runs on top of PyTorch and thus integrates into existing model training pipelines. However, DeepSpeed was developed for large transformer models and, to the best of our knowledge, never applied to large-scale training of GNNs.

In our work, we forked the OCP project to integrate DeepSpeed into the training pipeline of the most widely-used and successful models in energy and force predicition of catalysts. We benchmark our approach by performing an abloation analysis the epoch runtime and memory usage of the GNN architectures GemNet and DimeNet++ using DeepSpeed on the OC20 dataset.

1.3 Contributions

Our main contributions are:

1. We scaled force and energy predictions in atomic simulations of catalysts with GNNs by integrating Microsoft DeepSpeed into the Open Catalyst Project. [1]

2. We evaluated the effectiveness of DeepSpeed features, in particular the Zero Redundancy Optimizer, on real-world graph models in contrast to only applying it to large-scale transformer models.

3. We developed a profiling module for the OCP codebase to allow deeper investigations of CPU/GPU memory usage and model runtimes.

1.4 Structure

Our final report is structured as follows: Section 2 follows with an overview of important preliminary background topics of our work. Section 3 describes important scaling techniques from DeepSpeed and our work to integrate them into the OCP repository. In Section 4, we provide benchmark data for our changes and evaluate them in detail. Section 5 presents similiar literature to our work, while Section 6 gives a brief review of our project organization. Finally, Section 7 wraps up the report with a conclusion.

2 Preliminaries

2.1 Graph Neural Networks

In most of the contemporary supervised machine learning tasks, we are given datasets $\mathcal{D} \coloneqq \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n}$ with $\boldsymbol{x}_i \in \mathbb{R}^d$, $\boldsymbol{y}_i \in \mathbb{R}^k$, where the inputs $\boldsymbol{x}_i$ are assumed to be drawn

[1] https://github.com/TUM-DI-Lab-Graph-Scaling/ocp

independently from an identical distribution and we want to generally predict the labels $\boldsymbol{y}$ from $\boldsymbol{x}$. However, for many complex structured non-Euclidean types of data, these assumptions might not be the best fit. For example, just consider the two following settings:

- Suppose we are given molecule structures as input graphs $\boldsymbol{x}_i$ and, as an outcome, we want to predict the energy $\boldsymbol{y}_i$ of the respective structures. Clearly, there is no straightforward way to embed a molecule into some $\mathbb{R}^d$.
- Suppose our inputs $\boldsymbol{x}_i$ are features of users in a social network and we want to predict interaction $\boldsymbol{y}_i$ with some new content. In this case, the iid assumption of the individual data points $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is unreasonable because users themselves might share common interests or make each other aware of the content. Therefore it would be sensible to consider the existing connections between the data points.

Surely, one could find some representation of the above input data and try to use some conventional method (i.e. try to let a neural network learn these structures on its own), however this would be very inefficient or even unfeasible for many large datasets.

For simpler structured data like sequences or images, neural network architectures that exploit their intrinsic structure have been successful for quite some time, the most prominent examples including Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) respectively.

Graph Neural Networks (GNNs) can now be regarded as a generalization of CNNs and RNNs to general graph structures. More precisely, we can say that GNNs exploit the structure of the input graph(s) by incorporating reasonable assumptions about its predicting functions like equivariance (in the case of node prediction) or invariance (for global predictions) with respect to permutations of the input nodes.

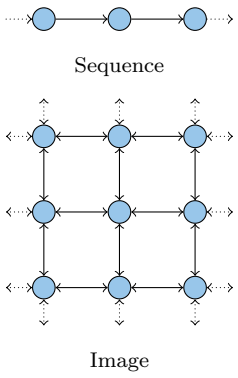


Figure 1: Structured data as graphs.

We express GNNs via the Graph Network (GN) framework introduced in [Bat+18], which defines a common language for a quite general class of graph-input functions made out of individual building blocks. EGNs (Extended Graph Networks) [Sri+22], which we will introduce afterwards, are even more expressive than normal GNs as they also model higher-order interactions between nodes (i.e. interactions between 3 or more nodes, or—alternatively—between two edges). Particularly recent GNNs for atomic simulations that make use of such higher-order interactions, as the ones we will describe in Subsection 2.2, [KGG20; GBG21], can be expressed with EGNs.

2.1.1 Graph Networks

Most of this section is an outline of [Bat+18, Section 3.2]. Within the GN framework, a graph is modeled as a 3-tuple

$$G = (\boldsymbol{u}, V, E),$$

where $\boldsymbol{u} \in \mathbb{R}^{d_u}$ is a global attribute of the graph. V and E are the set of nodes and set of edges respectively;

$$\begin{aligned} V &= \{\boldsymbol{v}_i\}_{i=1}^{n_v}, & \boldsymbol{v}_i &\in \mathbb{R}^{d_v}; \\ E &= \{(\boldsymbol{e}_k, r_k, s_k)\}_{k=1}^{n_e}, & \boldsymbol{e}_k &\in \mathbb{R}^{d_e},\ r_k, s_k \in \{1, \dots, n_v\}; \end{aligned}$$

where $\boldsymbol{v}_i$ represents the node attributes and $\boldsymbol{e}_k$ is the attribute of an edge going from s_k to r_k. In many cases, these attributes are also called *embeddings*.

A GN block now consists of the update functions ϕ_e, ϕ_v and ϕ_u for edges, nodes, and attributes respectively, as well as aggregation functions $\rho_{e\to v}$, $\rho_{e\to u}$, and $\rho_{v\to u}$, with

$$\begin{aligned} \boldsymbol{e}'_k &\coloneqq \phi_e(\boldsymbol{e}_k, \boldsymbol{v}_{r_k}, \boldsymbol{v}_{s_k}, \boldsymbol{u}), & \overline{\boldsymbol{e}}'_i &\coloneqq \rho_{e\to v}(E'_i), \\ \boldsymbol{v}'_i &\coloneqq \phi_v(\overline{\boldsymbol{e}}'_i, \boldsymbol{v}_i, \boldsymbol{u}), & \overline{\boldsymbol{e}}' &\coloneqq \rho_{e\to u}(E'), \\ \boldsymbol{u}' &\coloneqq \phi_u(\overline{\boldsymbol{e}}', \overline{\boldsymbol{v}}', \boldsymbol{u}), & \overline{\boldsymbol{v}}' &\coloneqq \rho_{v\to u}(V'); \end{aligned}$$

where $E'_i \coloneqq \{\, (\boldsymbol{e}'_k, r_k, s_k) \mid k = 1, \dots, n_e,\ r_k = i \,\}$ is the set of incoming edges at node i, $E' \coloneqq \{(\boldsymbol{e}'_k, r_k, s_k)\}_{k=1}^{n_e}$ contains all updated edges, and $V' \coloneqq \{\boldsymbol{v}'_i\}_{i=1}^{n_v}$ consists of the updated nodes of the graph. All in all, this results in the updated graph $G' = (\boldsymbol{u}', V', E')$ being the output of the GN block. The computations performed by a GN block can also be described in terms of the following phases:

Edge Update: The edges are updated based on their respective edge attribute, their source and receiver node attributes, and the graph's global attribute using the update function ϕ_e.

Edge Aggregation: The new edge attributes are aggregated on each node as well as the whole graph using the aggregation functions $\phi_{e\to v}$ and $\phi_{e\to u}$ respectively.

Node Update: The nodes are updated based on their respective node attributes, the newly aggregated messages from the incoming edges, and the graph's global attribute using the update function ϕ_v.

Node Aggregation: The new node attributes are aggregated on the entire graph via the aggregation function $\phi_{v\to u}$.

Global Update: The graph's global attribute is updated based on its predecessor and the newly aggregated edge and node attributes.

Furthermore, the aggregation functions have to be invariant w.r.t. permutations of the input nodes or edges. In many cases, these aggregation functions are kept simple and are just a sum or mean operation. A GN is just a sequence of such GN blocks which successively updates the input graph's attributes. A GNN is simply a GN whose blocks use neural networks to implement the update functions ϕ_e, ϕ_v, ϕ_u. Note that global graph prediction, node prediction as well as edge prediction can be performed with a GNN as just the corresponding attribute of the GNN output graph (or some final transformation of it) can be seen as the network output.

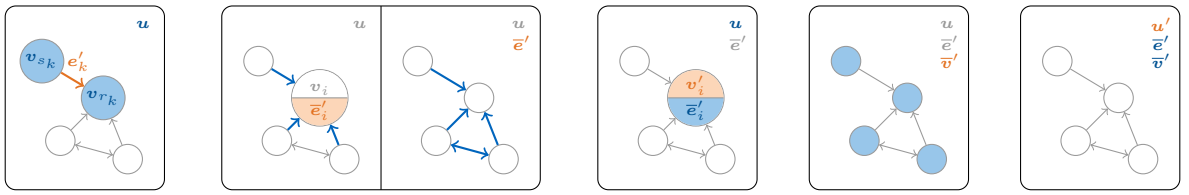


Figure 2: Schematic depiction of the GN block computation phases. Orange attributes are updated or created while blue ones contribute to the update.

2.1.2 Extended Graph Networks

The Extended Graph Network (EGN) framework [Sri+22, Section 2.1] is a generalization of GNs to graphs which incorporate higher-order interactions: In the EGN framework, a graph is defined as

$$G \coloneqq (\boldsymbol{u}, V, E, T)$$

with $\boldsymbol{u}$, V, and E as before, as well as

$$T \coloneqq \left\{(\boldsymbol{t}_m, e_1^{(m)}, \dots, e_{l_m}^{(m)})\right\}_{m=1}^{n_t}, \qquad \boldsymbol{t}_m \in \mathbb{R}^{d_t},\ e_1^{(m)}, \dots, e_{l_m}^{(m)} \in \{1, \dots, n_e\}, \tag{1}$$

where $\boldsymbol{t}_m$ represents the attribute of an interaction between the l_m edges indexed by $e_1^{(m)}, \dots, e_{l_m}^{(m)}$.

In an EGN block, these higher-order interactions are now taken into account by successively updating the interactions starting from the highest order: The interactions of a certain order l are first updated individually and then aggregated at each interaction of order $l-1$, whereafter the same is carried out for the interactions of order $l-1$, and so on. In the special case of only 3-order interactions, i.e. triplets of nodes or pairs of edges, this would mean that the usual GN computation phases are preceded by a **Triplet Update** and a **Triplet Aggregation** phase. In Subsection 2.2, we will use this EGN framework to introduce some of the most recent and well-performing GNN architectures for atomic simulations—so, for now, think of edge attributes as the distance between two atoms in a molecule and of the triplets as the bond angle between two chemical bonds.

2.2 GNNs for Atomic Simulations

2.2.1 General

Suppose one has information about the chemical and spatial structure of a molecule and would like to predict properties of this very molecule, e.g. its energy (at the current point in time or after some evolution) or the forces acting on the individual atoms. Some of these properties might be directly computable using quantum mechanical methods—like Density Functional Theory (DFT) [BT14]. However, quantum mechanics based methods often come with significant drawbacks as they are extremely expensive from a computational standpoint, imposing the necessity to either optimize these conventional methods as far as possible or use completely different approaches.

One alternative approach is to compute or predict such molecule properties using supervised machine learning in order to learn a function that maps molecule structures

to their respective properties. For this task, Graph Neural Networks (GNNs) are very well suited not only because a molecule can naturally be expressed as a graph (e.g. using its structural formula and interatomic distances), but also because in this case, the assumption of invariance w.r.t. permutations of the atoms or the assumption that the update rules do not differ between nodes or edges and that one is hence learning somewhat universal functions, seem very reasonable.

From now on, we define a molecule consisting of n atoms by its atomic numbers $\boldsymbol{z}$ and its positions $\boldsymbol{X}$, where

$$\boldsymbol{X} \coloneqq \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}, \ \boldsymbol{x}_i \in \mathbb{R}^3 \qquad \text{and} \qquad \boldsymbol{z} \coloneqq \{z_1, \dots, z_n\}, \ z_i \in \mathbb{N}. \tag{2}$$

Suppose $E(\boldsymbol{X}, \boldsymbol{z})$ is the energy of the whole molecule structure, then one can immediately obtain the forces acting on the individual atoms indexed by $i \in \{1, \dots, n\}$ as

$$\boldsymbol{F}_i \coloneqq \frac{\partial E(\boldsymbol{X}, \boldsymbol{z})}{\partial \boldsymbol{x}_i} \in \mathbb{R}^3.$$

If we now want to simultaneously predict the energy E and forces $\boldsymbol{F}_i$ of a molecule structure, there are two options: In *energy-centric models*, a global graph feature which models molecule energy is regarded as the GNN output and the forces are obtained by differentiating w.r.t. the atom locations. In contrast, *force-centric models* predict both energy and forces directly as a global attribute and node embeddings respectively.

In many cases there is still some arbitrariness in the choice of the exact positions $\boldsymbol{X}$: Energies of molecule structures might be invariant w.r.t. translations and rotations, or forces might be invariant w.r.t. translations and equivariant w.r.t. rotations. Rather than letting the model figure out these fundamental invariants of its predicting functions on its own (e.g. by data augmentation), they are often directly incorporated into the model to reduce complexity. One approach to achieve this is to compute pairwise distances d_{ij} between the atoms and use them instead of the individual atomic locations $\boldsymbol{X}$. However, this might not always be sufficient to completely characterize the spatial structure of a molecule: Since the interatomic distances are only kept for a part of the pairs of nodes, there exist pairs of different molecules which cannot be distinguished with this approach [KGG20, Appendix A]. In the following subchapters, we will introduce two GNN architectures that were initially proposed by Gasteiger et al. and overcome this problem.

2.2.2 DimeNet and DimeNet++

DimeNet ("Directional Message Passing Neural Network") [KGG20], which was proposed by Gasteiger et al. in 2020, is an energy-centric GNN architecture for predicting energies and forces of molecule structures. DimeNet does not only consider pairwise distances between atoms, but also takes advantage of directional information. In the EGN language (see 2.1.2 and [Sri+22]), 3-way interactions between nodes are modeled.

Just as in (2), the network receives the positions $\boldsymbol{X}$ and atomic numbers $\boldsymbol{z}$ of a molecule as inputs. At first, define the pairwise distances and relative direction vectors

$$d_{ij} \coloneqq \|\boldsymbol{x}_j - \boldsymbol{x}_i\|_2, \qquad \vec{\boldsymbol{x}}_{ij} \coloneqq \boldsymbol{x}_j - \boldsymbol{x}_i.$$

In the beginning, a molecular graph is given or it is defined by connecting the atoms with distance below some threshold c. Denote by $\mathcal{N}_i \subseteq \{1, \dots, n\}$ the neighborhood of the atom i. Directional information is now leveraged by also taking the angles

$$\alpha_{kji} \coloneqq \measuredangle(\vec{\boldsymbol{x}}_{jk}, \vec{\boldsymbol{x}}_{ji})$$

between neighboring edges $(\_, k, j)$ and $(\_, j, i)$ into account. The distances and angles are then transformed to a representation which is related to DFT calculations [KGG20, Section 5] by using radial and spherical Bessel functions:

$$\boldsymbol{e}_{\mathrm{RBF}}^{(ji)} \coloneqq \boldsymbol{e}_{\mathrm{RBF}}(d_{ji}), \qquad \boldsymbol{a}_{\mathrm{SBF}}^{(kji)} \coloneqq \boldsymbol{a}_{\mathrm{SBF}}(d_{kj}, \alpha_{kji}).$$

In an embedding phase, edge embeddings $\boldsymbol{m}_{ji}^{(1)}$ and outputs $t_i^{(1)}$ or a global attribute $t^{(1)} = \sum_{i=1}^{n} t_i^{(1)}$ are initialized based on the atomic numbers $\boldsymbol{z}$ and $\boldsymbol{e}_{\mathrm{RBF}}^{(ji)}$. Both node embedings $\boldsymbol{h}_i \coloneqq \sum_{j \in \mathcal{N}_i} \boldsymbol{m}_{ji}$ and triplet embeddings are handled implicitly (i.e. the **Triplet Update** and **Node Update** outputs from 2.1.1, 2.1.2 do not depend on their respective predecessors). From now on, we denote implicit or irrelevant embeddings by "_".

In the EGN language, the initial graph is defined as follows:

$$G = (t^{(1)}, V^{(1)}, E^{(1)}, T^{(1)})$$

with nodes, edges and triplets

$$\begin{aligned}
V^{(1)} &\coloneqq \left\{ \boldsymbol{h}_i^{(1)} \;\middle|\; i \in \{1, \dots, n\} \right\}, \\
E^{(1)} &\coloneqq \left\{ (\boldsymbol{m}_{ji}^{(1)}, i, j) \;\middle|\; i, j \in \{1, \dots, n\}, d_{ji} \leq c \right\}, \\
T^{(1)} &\coloneqq \left\{ (_, (\boldsymbol{m}_{kj}^{(1)}, j, k), (\boldsymbol{m}_{ji}^{(1)}, i, j)) \;\middle|\; i, j, k \in \{1, \dots, n\},\ d_{ji}, d_{kj} \leq c \right\};
\end{aligned}$$

i.e. the triplets are all possible paths of length two on the edges of G.

After this initial graph embedding has been computed, a series of EGN blocks, here called *interaction blocks*, are applied to the graph.

Within such an interaction block, the edge embeddings are updated as follows (i.e. corresponds to **Edge Update**):

$$\boldsymbol{m}_{ji}^{(l+1)} = f_{\mathrm{update}}\left(\boldsymbol{m}_{ji}^{(l)}, \underbrace{\sum_{k \in \mathcal{N}_j \setminus \{i\}}}_{\textbf{Triplet Aggr.}} \underbrace{f_{\mathrm{int}}\left(\boldsymbol{m}_{kj}^{(l)}, \boldsymbol{e}_{\mathrm{RBF}}^{(ji)}, \boldsymbol{a}_{\mathrm{SBF}}^{(kj,ji)} \right)}_{\textbf{Triplet Update}} \right),$$

where f_{int} and f_{update} are implemented by neural networks. These edge embeddings are then directly aggregated to the new node embeddings

$$\boldsymbol{h}_i^{(l+1)} = \sum_{j \in \mathcal{N}_i} \boldsymbol{m}_{ji}^{(l+1)}$$

and the outputs $t_i^{(l+1)}$ as well as the global attribute $t^{(l+1)}$ are updated (based on another **Edge Aggregation** step). Finally, the global output t that represents the molecule's

energy is aggregated over all interaction block outputs. The whole DimeNet architecture as depicted in its original paper [KGG20] can be seen in Figure 3.

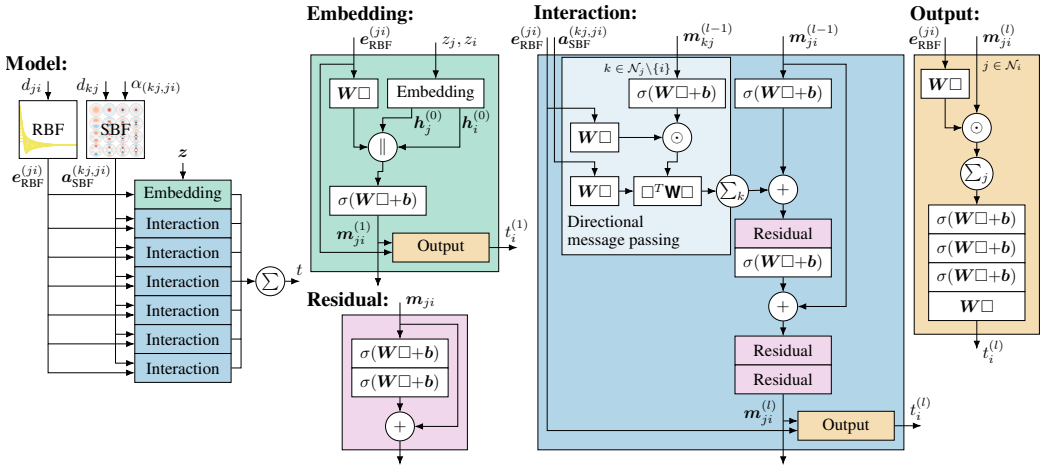


Figure 3: The original DimeNet architecture as depicted in [KGG20].

DimeNet++ [Gas+20] is an upgraded version of DimeNet that increases the efficiency of training without impacting the quality of results. These improvements include replacing the bilinear layer in the directional message passing step by a simpler Hadamard product, downprojecting the embeddings into lower dimensions in computation-intensive parts of the model and up-projecting them back to the original dimensions after these layers, and using less interaction layers.

2.2.3 GemNet

Gasteiger et al. also authored GemNet ("Geometric Message Passing Neural Network") [GBG21] as an improvement over DimeNet and DimeNet++ (see [KGG20; Gas+20] and 2.2.2) that offers the flexibility for both energy-centric and force-centric predictions as well as 3-way or 4-way message passing.

Just like DimeNet and DimeNet++, GemNet takes the positions $\boldsymbol{X}$ and atomic numbers $\boldsymbol{z}$ of a molecule (defined in (2)) as inputs. For $a, b \in \{1, \dots, n\}$ write

$$d_{ab} := \|\boldsymbol{x}_b - \boldsymbol{x}_a\|_2 \qquad \text{and} \qquad \vec{\boldsymbol{x}}_{ab} := \boldsymbol{x}_b - \boldsymbol{x}_a$$

for the distance and relative direction from $\boldsymbol{x}_a$ to $\boldsymbol{x}_b$ respectively. In GemNet, two graphs are considered: an interaction graph and an embedding graph. The molecule's interaction graph does not change during the forward pass of the network and two atoms *interact* if their distance is below some cutoff c_{int}. In addition to this interaction graph, all atom pairs whose distance is below some threshold c_{emb} are *embedded*—and, thus, edges of the embedding graph.

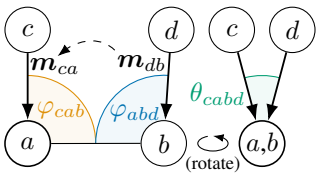


Figure 4: [GBG21]

In contrast to DimeNet, the 3- or 4-way interactions for message passing are used in a slightly different way: Consider pairwise different $a, b, c, d \in \{1, \dots, n\}$, where $(\_, a, b)$ are

interacting and $(\_, c, a)$, $(\_, d, b)$ are embedded. Set

$$\varphi_{cab} \coloneqq \measuredangle(\vec{\boldsymbol{x}}_{ca}, \vec{\boldsymbol{x}}_{ab}) \qquad \text{and} \qquad \varphi_{abd} \coloneqq \measuredangle(\vec{\boldsymbol{x}}_{ab}, \vec{\boldsymbol{x}}_{db}),$$

as well as

$$\phi_{cabd} \coloneqq \measuredangle(\boldsymbol{\Pi}\vec{\boldsymbol{x}}_{ca}, \boldsymbol{\Pi}\vec{\boldsymbol{x}}_{db}), \qquad \boldsymbol{\Pi} \coloneqq \boldsymbol{I}_3 - \vec{\boldsymbol{x}}_{ab}\vec{\boldsymbol{x}}_{ab}^{\top},$$

where $\boldsymbol{\Pi}$ is the orthogonal projection on the orthogonal complement of $\vec{\boldsymbol{x}}_{ab}$ (see Figure 4).

Similarly to DimeNet, this relative directional information is then encoded by the transformations

$$\boldsymbol{e}_{\mathrm{RBF}}^{(ca)} = \boldsymbol{e}_{\mathrm{RBF}}(d_{ca}), \quad \boldsymbol{e}_{\mathrm{CBF}}^{(cab)} = \boldsymbol{e}_{\mathrm{CBF}}(d_{ca}, \varphi_{cab}), \quad \boldsymbol{e}_{\mathrm{SBF}}^{(cabd)} = \boldsymbol{e}_{\mathrm{SBF}}(d_{ca}, \varphi_{cab}, \phi_{cabd})$$

which are related to DFT calculations. These three quantity types $\boldsymbol{e}_{\mathrm{RBF}}^{(ca)}$, $\boldsymbol{e}_{\mathrm{CBF}}^{(cab)}$, $\boldsymbol{e}_{\mathrm{SBF}}^{(cabd)}$ can now be regarded as the (constant) features of the molecule that are used by GemNet for updating the input graph. In GemNet, the higher-order interactions are

$$T \coloneqq \{(\_, c, a, b) \mid c, a, b \in \{1, \dots, n\},\ d_{ca} \leq c_{\mathrm{emb}},\ d_{ab} \leq c_{\mathrm{int}}\} \tag{3}$$
$$\cup \{(\_, c, a, b, d) \mid c, a, b, d \in \{1, \dots, n\},\ d_{ca} \leq c_{\mathrm{emb}},\ d_{ab} \leq c_{\mathrm{int}},\ d_{db} \leq c_{\mathrm{emb}}\}; \tag{4}$$

where the higher-order attributes are indexed by the involved nodes (contrary to the definition in (1)), and all appearing a, b, c, d are assumed to be pairwise different.

GemNet admits the following variants:

- *GemNet-T* is an energy-centric variant of GemNet that only uses the 3-way interactions from (3), which are updated using $\boldsymbol{e}_{\mathrm{RBF}}^{(ca)}$ and $\boldsymbol{e}_{\mathrm{CBF}}^{(cab)}$. In Figure 5, this is indicated by the **T-MP** block.
- In contrast, *GemNet-Q* is an energy-centric version of GemNet that also successively updates the 4-way interactions (4), taking the entirety of $\boldsymbol{e}_{\mathrm{RBF}}^{(ca)}$, $\boldsymbol{e}_{\mathrm{CBF}}^{(cab)}$ and $\boldsymbol{e}_{\mathrm{SBF}}^{(cabd)}$ into account. In Figure 5, this is represented by the **Q-MP** block.

Force-centric versions of GemNet-T and GemNet-Q are called *GemNet-dT* and *GemNet-dQ* respectively.

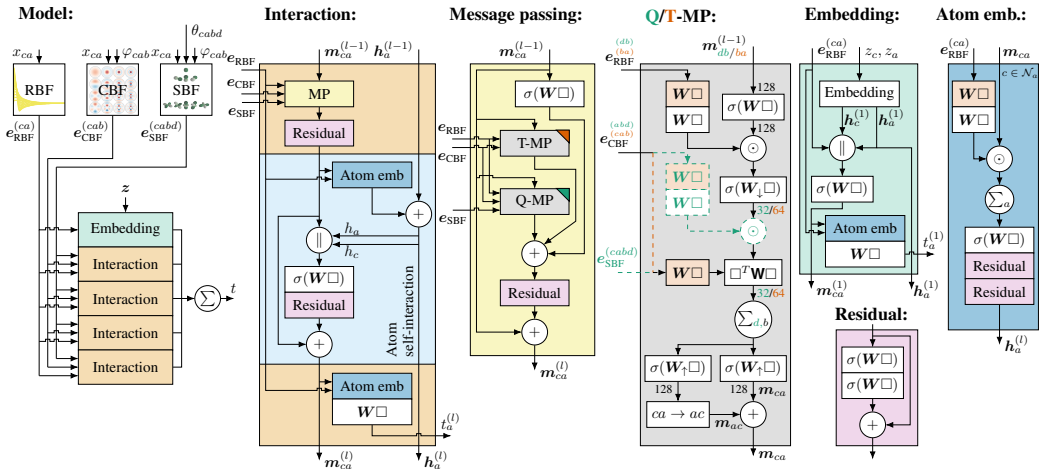


Figure 5: Complete architecture of GemNet as depicted in [GBG21, Appendix F].

2.3 Open Catalyst Project

Catalysis plays a crucial role in the chemical industry, including energy storage and conversion in fuel cells or the production of ammonia for fertilizers by enabling new reactions and improved process efficiencies [Cha+21]. Problematic is that the number of materials that can be used or modified in catalysis is very large and modeling of materials during reactions is very complex and compute-intensive.

In recent years, machine learning methods and in particular GNNs have shown great success in catalyst discovery: Being able to efficiently and accurately predict the forces and energy of inorganic and organic interfaces for use in catalysis avoids the computational bottlenecks of traditional simulation tools (see 2.2.1).

The Open Catalyst Project[2] [Cha+21] is a joint effort by Facebook AI and Carnegie Mellon University's Department of Chemical Engineering which was launched in 2020 and aims to provide a unified dataset and baseline models for predicting forces and energy in molecular simulations of catalysts. The project provides the Open Catalyst 2020 (OC20) dataset, which contains about 1.2 million relaxations of molecular adsorptions onto surfaces simulated with DFT. Fundamentally this precomputed dataset opens the door for precise predictions through machine learning models and allows for large-scale explorations of new catalysts.

Furthermore, the Open Catalyst Project offers unified implementations of DimeNet 2.2.2 and GemNet 2.2.3, as well as its subvariants. Developers can clone the repository and select an model to be trained or evaluated on force-centric or energy-centric tasks with precomputed simulations of catalysts. New models can be submitted to the evaluation server and get listed with their metrics on the OCP leaderboard[3].

Recently, the OCP team released a new version version of the dataset, Open Catalyst 2022 (OC22) [Tra+22].

2.4 Scalable Machine Learning

Training data is getting increasingly larger and model architectures are getting more complex with increasing numbers of parameters. Scaling and parallelization of large-scale machine learning models play a key role in reaching peak performance and accuracy. In this chapter, we will discuss scaling strategies for deep learning models.

2.4.1 Scaling up vs. Scaling out

Before scaling the training of a machine learning model, the question in which form the scaling should happen has to be answered. There are two fundamental types of scaling: *scaling up* and *scaling out*. When only the resources of the existing machine, on which the model is trained, are expanded in order to achieve better performance, we speak of scaling up.

On the other hand, if an increase in resources of a single machine is no longer technically possible or not possible for other reasons, the underlying resources must be increased in a different way. Distributing the training pipeline over several processes presents a way

[2] https://github.com/Open-Catalyst-Project/ocp

[3] https://opencatalystproject.org/leaderboard.html

to further scale, whereby the processes can also be distributed over multiple machines. This is known as scaling out.

A single machine multi-GPU setup is an interesting middle ground between scaling up and scaling out. Although the entire pipeline is executed on only one machine, we still employ a multi-process system since each of the GPUs is used through independent processes.

2.4.2 Data, Model and Pipeline Parallelism

In our work, we used a scaling out approach to scale the existing pipelines. Three common scaling-out strategies have emerged in recent years for machine learning models. With a large amount of training data, long training on only a single machine becomes unfeasible. To scale the training to multiple devices, *data parallelism* is used to partition and distribute the training data while replicating the model on each machine. Gradients are exchanged across machines after the backward pass, ensuring that all models will be in the same state at optimizer updates.

When the model size exceeds the memory capacities of a single device, *model parallelism* [Sha+18; Sho+19] can be used to split model parameters among multiple processes, devices or even machines. A neural network architecture can be horizontally and vertically partitioned, whereby in horizontal partitioning each device holds different layers of the model and in vertical partitioning layers are cut internally and each device holds a part of each layer.

A disadvantage of the naïve implementation of model parallelism is the sequential execution of forward and backward passes through the layers. All other machines are forced to wait when a single machine is performing computations, often referred to as bubble overhead in literature. To avoid bubble overhead, a more advanced form of model parallelism known as *pipeline parallelism* was developed, in which the model is split horizontally and the communication between layers gets overlapped through micro-batches. Popular implementations of pipeline parallelism include G-Pipe [Hua+18] and PipeDream [Har+18]. G-Pipe pipelines the execution of micro-batches accross machines, effectively overlapping executions of batches during forward and backward passes, while PipeDream intersperses the execution of forward and backward passes through more complex scheduling algorithms.

2.5 Microsoft DeepSpeed

Microsoft DeepSpeed[4] is a deep learning training optimization library built on top of PyTorch enabling large-scale distributed neural network training and making training on multiple devices (or machines) more memory- and time-efficient. Particularly for large-scale training of transformer models like Megatron-LM [Sho+19], DeepSpeed has already been used extensively and proven very useful[5]. As we will dive deeper into DeepSpeed's optimizations and their integration into the OCP codebase in the following chapter 3, this section is merely intended as a high-level overview of the optimizations that DeepSpeed offers.

[4] https://github.com/microsoft/DeepSpeed
[5] https://github.com/microsoft/Megatron-DeepSpeed

At the heart of DeepSpeed lies the *Zero Redundancy Optimizer* [Raj+19; Ren+21; Raj+21], or short *ZeRO*, which is a sophisticated and broad parallelization strategy combining data, model and pipeline parallelism paradigms. As it was already implied in the previous section 2.4.2, both model and data parallelism separately have their own flaws: While data parallelism is memory inefficient as there is a lot of redundancy in storing a copy of the whole model in each data-parallel process, model parallelism can impose significant communication overheads.

At this point, let us just briefly remind ourselves of the types of memory consumption that occur when training neural networks: The main components that consume memory are the model itself (including parameters, gradients and optimizer states), the current data batch, as well as residual state memory, namely activations that are computed during the forward pass or temporary buffers.

ZeRO, as introduced initially in [Raj+19], optimizes both model memory consumption and residual memory consumption in the following ways:

- *ZeRO-DP* ("data parallelism") is based on standard data parallelism and is designed for large models with significant memory footprints. ZeRO-DP can be seen as a combination of data and model parallelism in which (parts of) the model states are partitioned among the data-parallel processes, while a dynamic communication schedule ensures relatively low communication overhead. Various stages determining the scope of the model state partitioning can be progressively activated, like stage 1 (partitioning of optimizer states among the processes), stage 2 (partitioning of optimizer states and gradients) as well as the final stage 3 (partitioning of optimizer states, gradients, and parameters) which corresponds to full model parallelism.

- *ZeRO-R* ("residual memory") tackles residual memory consumption: Generally, activation memory consumption can be decreased by *activation checkpointing*, which means that certain activations are released from memory after computation in the forward pass and recomputed when needed in the backward pass—at the cost of this additional computation time. Inter alia, ZeRO-R now also partitions such activation checkpoints across the data-parallel processes and gathers them again when needed; apart from that, activations might also be moved to CPU memory.

ZeRO-Offload [Ren+21] and *ZeRO-Infinity* [Raj+21] are feature extensions of ZeRO that were added at a later point in time to enable offloading of model states to CPU memory for very large models in order to further increase GPU memory savings. While ZeRO-Offload enables offloading of optimizer states in stage 2 of ZeRO-DP, ZeRO-Infinity enables both offloading of model parameters and optimizer states in stage 3.

The specific demands of parallelization strategy in modern-day large-scale model training have furthermore given rise to custom extensions to DeepSpeed such as EleutherAI's DeeperSpeed [6] and the addition of custom highly-optimized kernels.

[6] https://github.com/EleutherAI/DeeperSpeed

3 Implementation

3.1 Usage of DeepSpeed

Our goal for this project was to use the DeepSpeed engine to speed up and scale GNNs. The Open Catalyst Project presents the ideal benchmark for our acceleration approaches since it contains several model definitions for GNNs in the context of molecular simulations. Previous research has shown that scaling OCP models to multiple hundreds of millions of parameters leads to better accuracies but requires ever larger infrastructure and ever more time for training [Sri+22]. Therefore we want to utilize DeepSpeed to scale beyond standard data parallelism and enable training of billion parameter models.

We took multiple steps to integrate DeepSpeed into the OCP project, which we will describe in the following section. All of our changes are available on GitHub in our organization fork of OCP[7].

To start off, we initialized the DeepSpeed engine. The initialization in our code looks roughly like this:

```
self.model, self.optimizer, _, _ = deepspeed.initialize(
    config=self.config["deepspeed_config"],
    model=self.model,
    model_parameters=self.model.parameters(),
    optimizer=self.optimizer,
)
```

DeepSpeed provides a model and optimizer wrapper to offer the same PyTorch API as before, but performs optimizations under the hood when forward, backward and step functions are invoked. Furthermore, we had to replace the original distributed environment setup with DeepSpeed's distributed setup. An example of why DeepSpeed needs a custom distributed environment is to spawn CPU worker processes for optimizer or parameter offloading. We will dive deeper into this optimization in Section 3.4.

To enable or disable specific features of the optimizer engine, DeepSpeed is configured using an extra JSON file. We extended the previous OCP configuration code to handle the DeepSpeed config as additional argument and pass it through to the DeepSpeed engine at the beginning of training. This also allows us to make feature specific changes to the codebase, since we also parse the config and make it available at runtime.

An example DeepSpeed configuration file looks like this:

```
{
    "train_batch_size": 16,
    "train_micro_batch_size_per_gpu": 2,
    "gradient_accumulation_steps": 1,
    "bf16": {
        "enabled": true
    },
    "zero_optimization": {
        "stage": 3,
        "contiguous_gradients": true
    }
}
```

[7] https://github.com/TUM-DI-Lab-Graph-Scaling/ocp

In this configuration, `train_micro_batch_size_per_gpu` refers to the batch size processed by one GPU in one step. `gradient_accumulation_steps` refers to the number of steps to accumulate gradients before averaging and applying them. `train_batch_size` is the effective batch size per accumulation step, so

$$\texttt{train\_batch\_size} = \#\,\text{GPUs} \cdot \texttt{train\_micro\_batch\_size\_per\_gpu} \cdot \texttt{gradient\_accumulation\_steps}.$$

We introduce the other configurations in later sections. `fp16` and `bf16` are mixed-precision configurations and are explained in Section 3.2. All the configurations in `zero_optimization` are introduced in Section 3.3 and amended in Section 3.4 by offload features.

3.2 Mixed Precision Training

Mixed precision training uses lower bit floating point types for optimizer states, gradients and model parameters to reduce the memory footprint during model training.

PyTorch offers out-of-the-box support for two types of mixed precision training: Floating point with 16 bits (FP16) and brain floating point with 16 bits (BFloat16) which we will introduce in the next subsections and explain our integration into the OCP project.

3.2.1 FP16

FP16 is a floating-point number format which uses 5 bits for the exponent, 10 bits for the fraction and 1 bit for the sign. This halves the used memory compared to the default single-precision floating-point format which uses 8 bits for the exponent and 24 bits for the fraction. The specification of FP16 can be found in the official IEEE 754-2008 standard[8]. FP16 lowers the precision and reduces the range of numbers that can be represented, making calculations less accurate and increasing the risk of overflows/underflows.

PyTorch and DeepSpeed offer automatic mixed precision for gradients. Developers can define regions with context managers to allow PyTorch to automatically choose the precision for GPU operations. Even though autocast worked for learnable parameters of the model as well as gradients, we had to convert input data and all constants defined in the models to half precision manually. With our adaptions, FP16 can be used by just setting the `fp16` flag in the DeepSpeed configuration.

3.2.2 BFloat16

As mentioned before, the use of FP16 leads to a higher risk of overflows and underflows. We experienced many overflows of gradients when using FP16 with DeepSpeed. To avoid overflows, DeepSpeed offers support for the Bfloat16[9] number format which allocates more bits to the exponent to extend the range of representable numbers while lowering precision. As with FP16, we had to include manual type casts in the OCP codebase to integrate Bfloat16 support. Bfloat16 can also be enabled from the DeepSpeed configuration using the `bf16` flag.

[8] https://standards.ieee.org/ieee/754/4211/

[9] https://cloud.google.com/tpu/docs/bfloat16

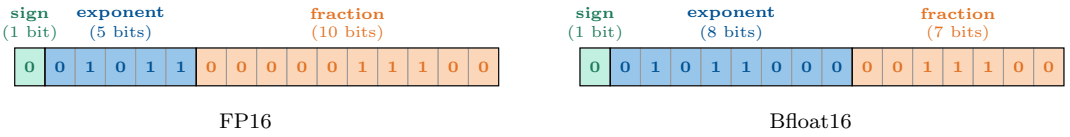


Figure 6: Comparison of the FP16 and the Bfloat 16 formats.

3.3 Zero Redundancy Optimizer (ZeRO)

The Zero Redundancy Optimizer (ZeRO) is a scaling engine on top of data-parallel training for machine learning models. In short, ZeRO removes memory redundancies by partitioning states which are used accross all processes such as the optimizer states, gradients and model parameters. Since all processes only have to maintain a shard of states, ZeRO should use less memory while retaining computational granularity and communication overhead compared to traditional data parallelism.

DeepSpeed contains three levels of ZeRO that progressively partition and distribute more and more states of the model. We will go through all stages, explain them and shortly explain our integration into OCP.

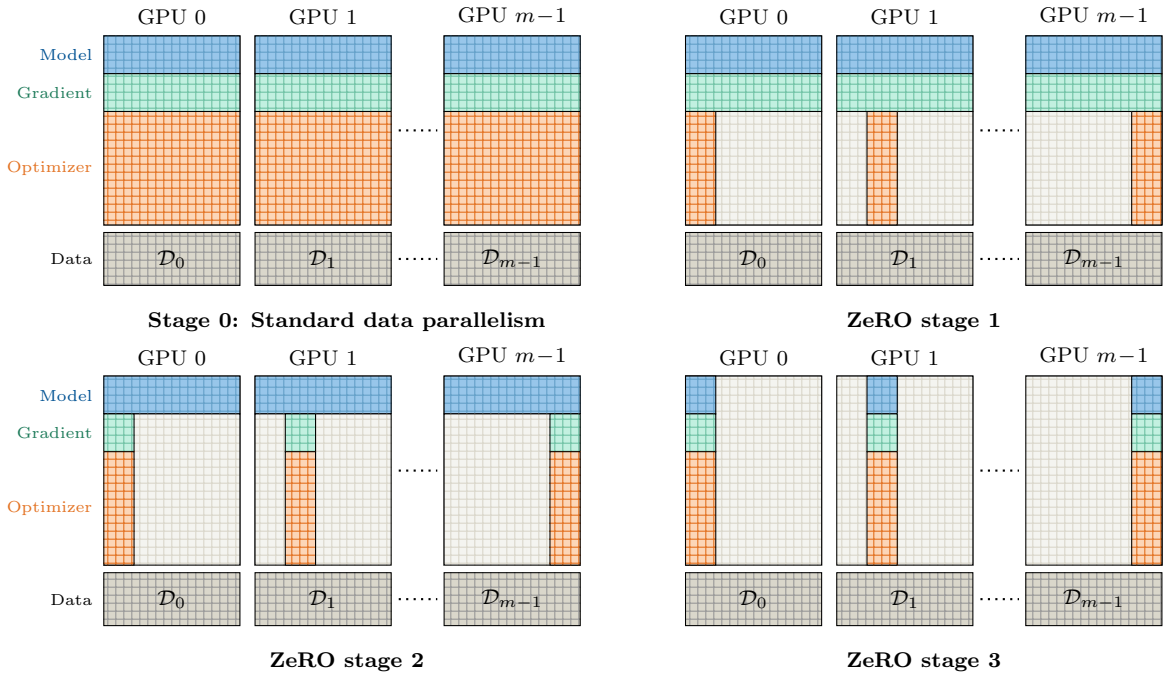


Figure 7: GPU memory utilization with different stages of ZeRO activated. The data $\mathcal{D}$ is distributed evenly among the m GPUs into $\mathcal{D}_0, \ldots, \mathcal{D}_{m-1}$.

3.3.1 Stage 1 (OS)

Stage 1 of ZeRO enables optimizer state partitioning. Each GPU only holds a shard of optimizer states. All GPUs do a normal training step over their data subset, however during the optimizer step, each GPU only updates and stores optimizer states of its own shard. After the update, all GPUs synchronize by an all-gather communication collective to update parameters accross all devices.

Stage 1 required FP16 or Bfloat16 integration for the OCP codebase. As mentioned in the previous section, we encountered overflow problems with FP16, so we mainly used Bfloat16 to enable ZeRO-1.

3.3.2 Stage 2 (OS+G)

Stage 2 of ZeRO partitions both optimizer states and gradients. ZeRO-2 generally follows the following execution path for each step: Each GPU holds a replica of the entire model (all parameters), however only a mutually exlusive portion of the parameters are updated on each of the devices. The GPUs only store the shard of gradients and optimizer states which they need for their particular portion of parameters during the forward and backward pass. After doing the optimizer step, the GPUs update each other through an *all-gather* NCCL communication collective.

Stage 2 is very tighly connected to Stage 1 regarding its implementation in DeepSpeed. Fortunately for us, this meant that after integrating Stage 1, Stage 2 worked out-of-the-box without any problems.

3.3.3 Stage 3 (OS+G+P)

Stage 3 of ZeRO partitions optimizer states, gradients and model parameters, thus merging the traditional data parallelism approach with model parallelism. Optimizer states and gradients are partitioned and synchronized as mentioned in Stage 1 and 2. The model parameter partitioning works with the following principles: Each GPU is assigned a shard of parameters. When model parameters outside the shard are required for the forward or backward pass, the GPU that owns the shard containing the parameters shares them via broadcast. This leads to far more communication overhead, however further reduces GPU memory requirements of the overall model.

The integration of ZeRO-3 was fraught with many problems. GemNet, for example, accesses the parameters of the layers directly in the model definition, which leads to problems with DeepSpeed since it replaces the tensors with its own wrapper classes that contain parameters only latently (parameters that are not in the shard of the GPU must first be loaded via NCCL). For this, we had to extend GemNet to first perform an all-gather operation on the parameters so that these are locally available with all GPUs before using them inside the model directly.

Another problem occurred with DimeNet where ZeRO did not automatically partition the model parameters. Our solution was to explicitly pass all model parameters as external parameters to the DeepSpeed engine. This does not provide optimal partitioning of the parameters, but at least lets the Zero-3 engine run.

After our integration, the ZeRO optimization can be enabled for all OCP models in the Deepspeed configuration file. The specification is as follows:

```
"bf16": {
    "enabled": "true"
}
"zero_optimization": {
    "stage": [1|2|3]
}
```

Note that the current DeepSpeed implementation requires Bfloat16 floating point precision to be enabled.

3.4 Offloading Optimizations

To further reduce GPU memory usage, DeepSpeed contains optimizations to offload data and compute from the GPU to CPU or even Non-Volatile Memory express (NVMe) memory.

Offloading optimizations in DeepSpeed were introduced in ZeRO-Offload [Ren+21]. ZeRO-Offload builds on top of the ZeRO Stage 2 optimizer which partitions gradients and optimizer states. With offloading enabled, each GPU offloads part of its partition to the CPU. The offload engine automatically determines gradients and optimizer states which can be computed on the CPU while minimizing communications between CPU and GPU and maximize memory savings on the GPU.

To determine the offload strategy, the offload engine represents the model as data-flow graph with nodes are model states (parameters, gradients and optimiter states) and edges are communication volume between that flows through the model during training. ZeRO-Offload uses a two-way partitioning scheme to split the graph into CPU and GPU partitionings while trying to optimize CPU-CPU computation overhead, communication overhead and memory savings.

Generally speaking, offloading is a scale-up approach to utilize as much resources on each machine as possible, including CPU cores and memory. Offloading should reduce GPU memory consuption at the cost of CPU memory and core utilization, while the overall runtime increases.

ZeRO-Offload for Zero Stage 2 is enabled by specifing the offload optimizer configuration inside of the ZeRO configuration:

```
"zero_optimization": {
    "stage": 2,
    "offload_optimizer": {
        "device": "[cpu|nvme]"
    }
}
```

ZeRO-Infinity [Raj+21] adds further features to ZeRO-Offload by building on top of ZeRO-3 (Stage 3) and also introducing NVMe offloading.

ZeRO-Infinity builds on top of the offload engine of ZeRO-Offload by also mapping the model to a data-flow graph as described above. However, the novel infinity offload engine also allows assigning data-flow partitions to NVMe memory and extends the graph by including activation memory hence also allowing offloading activation checkpoints to CPU memory or NVMe. ZeRO-Infinity also introduces the concept of memory-centric tiling for working memory to split larger layers in the model into smaller tiles, allowing the offload engine to further improve the partitioning scheme and prevent GPU memory overflows with single large layers which often occured before in large language models. To sum it up, ZeRO-Infinity allows models that are significantly larger than the GPU memory to be trained as long as enough CPU or NVMe is available.

Similiar to ZeRO-Offload, after our integration of ZeRO-3 into the OCP code base, infinity offloading worked out-of-the-box. ZeRO-Infinity's offload engine can be enabled

in Stage 3 using the two ZeRO configurations `offload_optimizer` and `offload_param` to offload either optimizer states or entire parts of the model:

```
"zero_optimization": {
    "stage": 3,
    "offload_optimizer": {
        "device": "[cpu|nvme]"
    },
    "offload_param": {
        "device": "[cpu|nvme]"
    }
}
```

4 Evaluation

4.1 Datasets

The OC20 dataset [Cha+21] contains training and evaluation data of roughly 1.2M DFT relaxations using a range of different materials, surfaces and adsorbates. Jointly with the dataset, three different challenges for the community were published: IS2RE (Initial Structure to Relaxed Energy), S2EF (Structure to Energy and Forces) and IS2RS (Initial Structure to Relaxed Structure). In our work, we focused on the first two.

4.1.1 Initial Structure to Relaxed Energy (IS2RE)

The Initial Structure to Relaxed Energy (IS2RE) task takes an initial atomic structure as input and predicts the energy after relaxation of the structure [Cha+21; Tra+22].

OC20 provides three dataset scaling factors for the IS2RE task: 10k, 100k and all structures + relaxations. Each dataset is a subset of the next higher-scaled dataset.

4.1.2 Structure to Energy and Forces (S2EF)

The Structure to Energy and Forces (S2EF) task takes an atomic structure and predicts energy of the structure and per-atom forces [Cha+21; Tra+22]. Four metrics are tracked during the S2EF task: Mean Absolute Error for the energy and forces (MAE), Force Cosine similarity, and energy as well as forces within a threshold (EFwT).

OC20 provides three dataset scaling factors for the S2EF task: 200k, 2M, 20M, and all. The scaling factor determines the number of atomic structures and pre-computed energy and per-atom forces with DFT. The input structures are provided as compressed trajectory files, so we had to locally uncrompress the files and embedded the structures into Lightning Memory-Mapped Databases (LMBDs). Similiar to the datasets in IS2RE, each dataset is a subset of the next higher-scaled dataset.

4.2 Evaluation Setup

We evaluated all models on a single machine with 2 AMD Epyc 7542 (32 CPU-Cores each), 528 GB memory, and 8 Nvidia RTX A6000 with 48 GB GDDR6-memory. The GPUs were not connected by NVLink, so we used P2P communication through PCI for NCCL

communication collectives. Practically speaking, this means that we set `NCCL_P2P_LEVEL` to `PIX`.

GemNet The first model we evaluated with our DeepSpeed enhancements was the force-centric GemNet-dT scaled up to roughly 292 million parameters corresponding to Gemnet-XL [Sri+22]. The model consists of 6 interactions blocks, an atom embedding size of 128, an edge embedding size of 1536, and triplet embedding size of 384. All other hyperparameters are equal to Gemnet-dT as it was configured for the OCP benchmark evaluation.

DimeNet++ Furthermore we evaluated the energy-centric DimeNet++ with the configuration defined in [Sri+22]. The model has 4 interaction blocks, a hidden dimension of 2048, an output block of 1536, and triplet dimension of 256, amounting to approximately 216 million parameters. Again, all other hyperparameters are equal to DimeNet++ as it was configured for the OCP benchmark evaluation.

For detailed analysis we gathered metrices from all our experiment runs regarding epoch runtimes, CPU memory usage, GPU memory usage, as well as GPU memory reserved or allocated by PyTorch directly. We collected the data with a custom profling framework for PyTorch which we present in more detail in Appendix A. In addition, we used the integrated PyTorch profiler[10] to collect step-by-step performance metrics, including information about communcation collectives and volume or breakdown of calls to CUDA operations. PyTorch profiler also includes plugins to load and visualize the profiling data in Tensorboard, which allowed us to do ad-hoc profiling even when the experiment was still running.

4.3 Results

4.3.1 GemNet

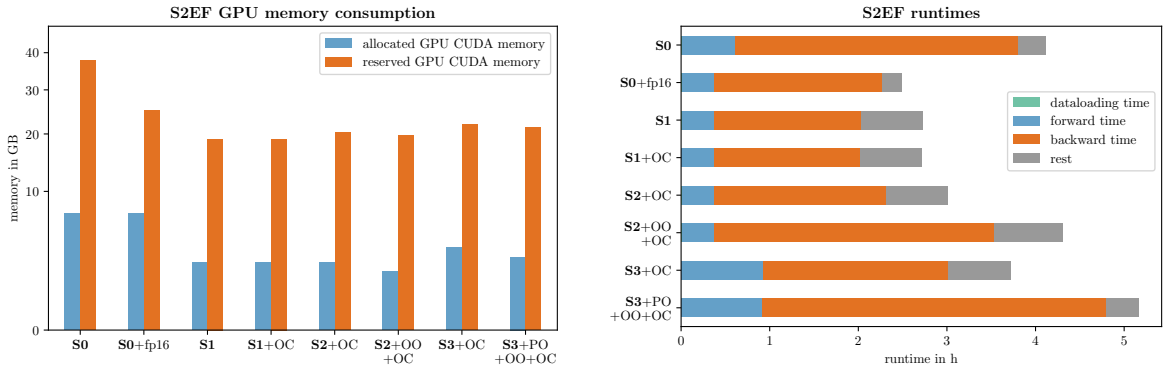
We evaluated the GemNet model on the S2EF task with the 200k dataset and the IS2RE task with the 100k dataset. For both tasks, we trained for one epoch using a batch size of 16 (since data parallelism was activated in all of the runs, this implies that each of the 8 GPUs received 2 molecules per forward pass).

We evaluated the GPU memory consumed by CUDA tensors and runtimes for GemNet training with 8 different configurations of DeepSpeed: The so-called DeepSpeed ZeRO stage 0 served as baseline for evaluations, in which all DeepSpeed optimizations are deactivated—i.e. stage 0 corresponds to basic data-parallel training. As a second configuration, we trained GemNet in half precision without any ZeRO optimizations in order to make sure that potential memory savings by ZeRO are actually caused by the model state partitioning and not merely by reducing the model parameters' precision.

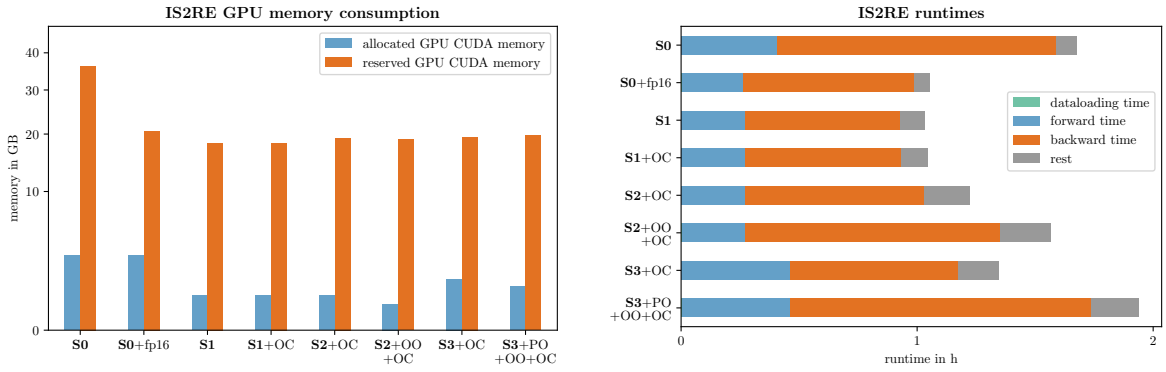
Apart from this, we evaluated two configurations of each stage of ZeRO-DP to benchmark them against the two baselines. More precisely, for stage 1, we did one run with and without overlapping communication each, while overlapping communication was activated throughout stage 2 and stage 3. In order to investigate potential memory savings by CPU offloading, we conducted standard stage 2 and stage 3 runs as well as runs in which the

[10] `https://pytorch.org/docs/stable/profiler.html`

respective maximal CPU offloading ability was exploited. The exact configurations and results can be seen in Figure 8, where "OC" stands for overlapping communication, "OO" for optimizer offloading and "PO" for parameter offloading.



S2EF		S0	S0+fp16	S1	S1+OC	S2+OC	S2+OO +OC	S3+OC	S3+PO +OO+OC
memory	allocated	7.11	7.1	2.36	2.36	2.36	**1.78**	3.51	2.78
	reserved	37.71	24.98	**18.99**	**18.99**	20.3	19.77	21.99	21.32
runtimes	epoch	04:07:00	**02:29:03**	02:43:45	02:42:51	03:00:41	04:18:11	03:43:01	05:09:53
	dataloading	00:00:51	00:00:45	00:00:29	00:00:26	00:00:30	00:00:29	00:00:27	**00:00:24**
	forward	00:35:43	**00:21:36**	00:21:59	00:21:57	00:21:58	00:21:57	00:54:57	00:54:32
	backward	03:11:31	01:53:35	01:39:38	**01:38:45**	01:56:29	03:09:20	02:05:26	03:52:43
	rest	00:18:53	**00:13:05**	00:41:39	00:41:41	00:41:43	00:46:23	00:42:11	00:22:13



IS2RE		S0	S0+fp16	S1	S1+OC	S2+OC	S2+OO +OC	S3+OC	S3+PO +OO+OC
memory	allocated	2.88	2.88	0.62	0.62	0.62	**0.36**	1.36	1.0
	reserved	36.27	20.6	**18.19**	**18.19**	19.24	**18.19**	19.44	19.68
runtimes	epoch	01:40:32	01:03:09	**01:01:58**	01:02:39	01:13:23	01:33:50	01:20:55	01:56:20
	dataloading	00:00:09	00:00:13	00:00:09	00:00:09	00:00:08	00:00:11	**00:00:07**	00:00:15
	forward	00:24:17	**00:15:43**	00:16:07	00:16:06	00:16:04	00:16:07	00:27:31	00:27:30
	backward	01:10:58	00:43:16	**00:39:29**	00:39:35	00:45:34	01:04:43	00:42:51	01:16:32
	rest	00:05:07	**00:03:55**	00:06:12	00:06:47	00:11:35	00:12:47	00:10:25	00:12:01

Figure 8: Memory consumption and runtimes for GemNet. Memory is in GB, runtimes in the *h:m:s* format.

While enabling half precision training on stage 0 does significantly reduce reserved memory, allocated memory is only marginally affected. Nevertheless, a decrease of runtime

by about 40% (S2EF) and 41% (IS2RE) can be observed.

When comparing stage 0 with half precision to standard stage 1, a significant decrease both in allocated GPU memory and reserved GPU memory can be observed (S2EF: -67% allocated, -23% reserved; IS2RE: -78% allocated, -12% reserved). This drop in memory consumption has to be caused by optimizer state partitioning among the GPUs since this is the only difference between the configurations. Both for the S2EF task and the IS2RE task, forward and backward pass are marginally faster with stage 1 activated. However, especially for the S2EF task, it is also clearly visible that communication overhead between the GPUs rises for stage 1: The part of the total epoch runtime which neither belongs to one of the tracked stages (calles "rest" in Figure 8) and thus mainly encompasses communication, increases by 219% (S2EF) and 58% (IS2RE).

Additionally, CPU offloading saves memory—most notable is a 58% decrease of allocated GPU memory with IS2RE when optimizer offloading is activated in stage 2. Nevertheless, in all of the observed cases this came at the cost of increased runtime.

Among all DeepSpeed configurations that we tested out, stage 2 with CPU offloading reduced allocated memory most both on S2EF and IS2RE, while also reserving the least (or almost the least) memory.

Stage 3 seems to be ineffective for GemNet as neither memory footprints nor runtime are improved. Likewise, communication overlapping (which we had hoped would reduce runtime) did not have the desired effect as can be seen from the comparison between the two stage 1 runtimes.

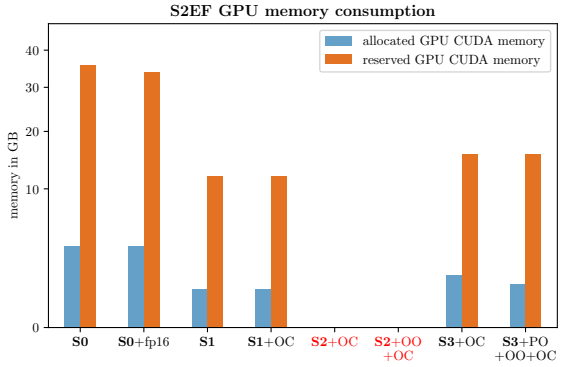
4.3.2 DimeNet++

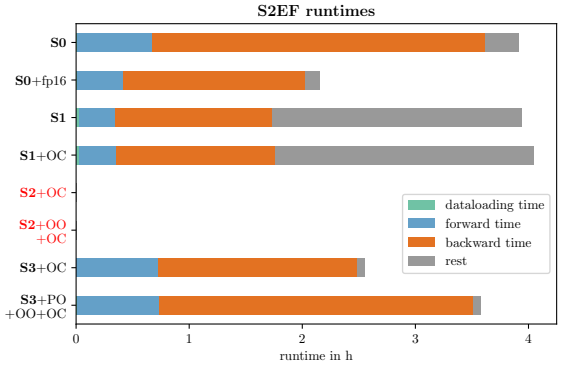
Similiarly to GemNet, we also evaluated DimeNet++ on S2EF with the 200k dataset and IS2RE with the 100k dataset. Both tasks were trained for one epoch on 8 GPUs with an effective batch size of 16 (each GPU receives 2 molecules per forward pass).

Baseline and DeepSpeed configurations were identical to our GemNet evaluation. With DimeNet++, as with GemNet, there is an improved runtime of 45% (S2EF) and 40% (IS2RE) by enabling mixed-precision, but the memory savings in the reserved memory are only marginal this time. Unusual behavior occurred after activating stage 1 and stage 2, as the runtime deteriorated compared to the baseline. The additional runtime was mainly due to very long reduce NCCL operations. Unfortunately, we could not debug why these operations took significantly longer on DimeNet++ and thus had a large impact on the runtime. However, it must be dependent on the model architecture, since this behavior was not observed with GemNet. Stage 2 on the S2EF tasks even encountered NCCL timeouts during reduce operations resulting in even longer runtimes. Both stage 2 runs on S2EF were aborted with an out-of-time error after a threshold of 6 hours.

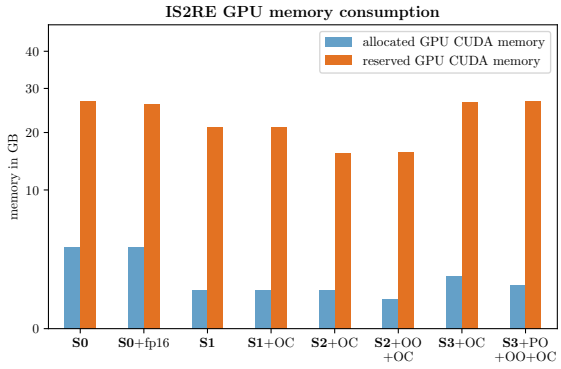
Other results of DimeNet++ were very similiar to our observations from GemNet. Stage 3 seems ineffective for the low number of GPUs. Optimizer and parameter offloading leads to increased runtime as seen in the comparison of both stage 3 runs. For the S2EF task, Stage 1 had the lowest memory consumption with 78% less allocated CUDA memory and almost 67% less reserved memory than the baseline. For the IS2RE task, stage 2 with optimizer offloading had the lowest memory consumption with 87% less allocated CUDA memory and 39% less reserved memory than the baseline. In both tasks, stage 0 with mixed-precision achieved the best runtime due to the tremendous communication

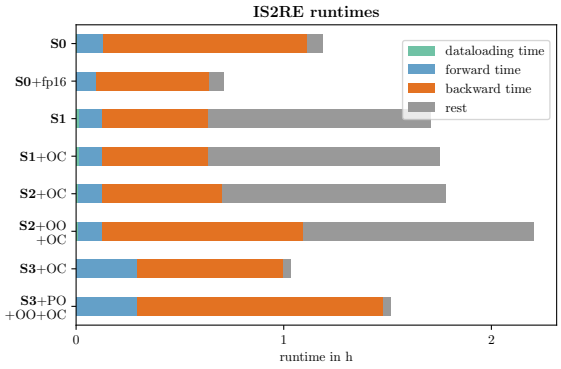
overhead of all stage 1 and 2 runs. Detailed information about the individual runs can be found in Figure 9.





S2EF		S0	S0+fp16	S1	S1+OC	S2+OC	S2+OO +OC	S3+OC	S3+PO +OO+OC
memory	allocated	3.46	3.47	**0.76**	**0.76**	–	–	1.44	1.0
	reserved	35.82	33.83	**11.87**	**11.87**	–	–	15.59	15.67
runtimes	epoch	03:54:52	**02:09:22**	03:56:28	04:02:47	OOT	OOT	02:33:25	03:34:34
	dataloading	00:00:26	00:00:28	00:01:39	00:01:44	OOT	OOT	**00:00:17**	00:00:19
	forward	00:40:09	00:24:52	**00:19:14**	00:19:41	OOT	OOT	00:43:02	00:43:50
	backward	02:56:26	01:36:06	**01:23:00**	01:24:09	OOT	OOT	01:45:56	02:46:15
	rest	00:17:49	00:07:54	02:12:33	02:17:11	OOT	OOT	**00:04:08**	00:04:10





IS2RE		S0	S0+fp16	S1	S1+OC	S2+OC	S2+OO +OC	S3+OC	S3+PO +OO+OC
memory	allocated	3.46	3.47	0.76	0.76	0.76	**0.46**	1.44	1.0
	reserved	26.92	26.25	21.08	21.08	**16.07**	16.2	26.68	26.82
runtimes	epoch	01:11:14	**00:42:37**	01:42:24	01:45:05	01:46:43	02:12:14	01:01:56	01:31:00
	dataloading	00:00:10	00:00:12	00:00:50	00:00:52	00:00:47	00:00:43	**00:00:06**	00:00:08
	forward	00:07:47	**00:05:32**	00:06:42	00:06:47	00:06:43	00:06:46	00:17:29	00:17:31
	backward	00:58:40	00:32:46	**00:30:28**	00:30:38	00:34:37	00:58:13	00:42:07	01:11:02
	rest	00:04:34	00:04:05	01:04:22	01:06:46	01:04:35	01:06:31	**00:02:12**	00:02:18

Figure 9: Memory consumption and runtimes for DimeNet++. Memory is in GB, runtimes in the *h:m:s* format. Red runs went out of time.

5 Related Work

Scaling of machine learning models has a rich scientific research history, mainly since the field represents an interesting and active collaboration between high performance computing, distributed systems, and deep learning communities. In this section we will highlight literature in the area of accelerating deep learning that are similar to our project or complement our work.

A. Sriram et al. [Sri+22] from Meta FAIR propose a very different approach to scaling Message Parsing Graph Neural Networks for atomic simulations, wherein the automata structures are internally partitioned across multiple GPUs. In their approach, known as *graph parallelism*, the highest-order interactions—which are triplets in their work—are partitioned so that each GPU receives a subset of the triplets and updates and aggregates them locally. The triplet messages are then gathered on all GPUs, allowing subsequent edge and node-level layers to be computed as before. Using graph parallelism, the authors scaled up DimeNet++ and GemNet, referring to their large models as DimeNet++-XL and GemNet-XL. GemNet-XL, containing close to 300 million model parameters, is the largest known model fully trained on OC20 for the OCP benchmark to date, and was also at the top of the leaderboard for both IS2RE and S2EF until the release of Gemnet-OC.

For more general purpose Graph Neural Network architectures like Graph Attention Networks [Vel+17] or GraphSAGE [HYL17], sampling is mostly used to avoid neighborhood explosions when building the message flow graphs. To also allow training on large-scale graphs, both popular geometric deep learning frameworks PyTorch Geometric (PyG) [FL19] and Deep Graph Library (DGL) [Wan+19] offer support for scale-out strategies. For PyG, there is Torch Quiver[11], which optimizes and scales single-node multi-GPU training. In DGL, multi-GPU training and optimizations are directly integrated into the codebase. In first iterations of our project, we evaluated both Quiver on Pytorch Geometric and DGL. In short, Torch Quiver is unfortunately very poorly maintained and currently not useable from our point of view. DGL offers great support for multi-GPU and our tests were very positive, however most atomic simulations models are implemented on top of PyTorch Geometric. Consequently, we decided against DGL because the effort of reimplementing DimeNet, GemNet, and GemNet-OC in DGL would have been too costly.

Moving away from atomic simulation and Graph Neural Network specific optimizations, several toolkits for more memory-efficient training of large models have been developed in recent years, either complementary or as alternatives to DeepSpeed. *Fairseq* [Ott+19], developed by Meta AI, was originally developed to allow smaller labs and developers to fine-tune pre-trained language models. Nowadays, the toolkit also includes support for mixed precision training, parameter offloading to CPU, and multi-GPU optimizations such as parameter and optimizer state sharding. A direct competitor to DeepSpeed is *FairScale* [Bai+21]. FairScale is also developed by Meta AI and contains optimizations similar to Zero-1, Zero-2 and Zero-3, combined with features for CPU offloading, FP16, and activation checkpointing. Another alternative to DeepSpeed is the *Fully Sharded Data Parallel* (FSDP)[12] module newly introduced in PyTorch 1.11. FSDP is a native integration of a subset of DeepSpeed and FairScale features directly into Py-

[11] https://github.com/quiver-team/torch-quiver

[12] https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Torch without the need to install external modules. The implementation of FSDP heavily borrows from FairScale while bringing a more intuitive and streamlined API for developers.

6 Project Organization

All work in this project was realized within the TUM Data Innovation Lab in collaboration with the Chair of Aerodynamics and Fluid Mechanics. Special thanks go to our Scientific Lead PhD candidate Ludger Pähler, as well as our Project Lead and Co-Mentor Dr. Ricardo Acevedo Cabra.

Our project was structured in 4 phases:

1. *Literature review (4 weeks)*: We read through important papers in the topics of our project such as Graph Learning, Graph Neural Networks, Graph Sampling, Message Passing for Quantum Chemistry, and Scalable Machine Learning; and discussed them weekly.

2. *Baseline (3 weeks)*: After getting into the topic theoretically, we started looking at the OCP project and elaborating the current state of the project to see what baseline we are comparing against.

3. *Implementation (4 weeks)*: After defining the baseline, we integrated and tested all DeepSpeed stages in the OCP project during our implementation phase.

4. *Evaluation and Final report (3 weeks)*: In the final weeks of the project, we conducted baselines and DeepSpeed experiments and recorded our observations and analysis. In addition, we focused on the final report in this phase.

Our team met twice each week on Monday and Friday to discuss progress and define next steps. Most of the time, individual team members met two to three additional times during the week to discuss specific tasks in their area of activity. For organizational purposes, team and personal conversations, we used a private Slack channel.

For version management of our code we created a GitHub organization[13] where we placed multiple projects like our own fork of the OCP project, evaluation scripts and also the source files for the final report. We tracked problems and errors we encountered as issues in the corresponding projects and worked with a branch structure to avoid disrupting the work of our teammates.

7 Conclusion

In this work, we scaled force and energy predictions of organic molecules on potential catalyst surfaces for a wide variety of recently proposed GNN architectures in the domain of atomic simulations. This was achieved by integrating Microsoft DeepSpeed into the OCP project, facilitating the exploration and use of DeepSpeed optimizations for the GNN models that are part of the OCP repository. This does not only include GemNet and

[13] https://github.com/TUM-DI-Lab-Graph-Scaling

DimeNet++, but also—as of recently—the current state-of-the-art GemNet-OC [Gas+22]. Additionally, we developed a module for OCP that enables comprehensive tracking of memory- and runtime-related metrics.

We evaluated memory and runtime savings on scaled-up versions of GemNet-dT (∼292M parameters) and DimeNet++ (∼216M parameters) which correspond to GemNet-XL and DimeNet++-XL [Sri+22], the state-of-the-art on the OCP leaderboard when our project phase began.

For our GemNet setup, we observed that using DeepSpeed's ZeRO optimization can significantly reduce memory footprints during training, resulting in almost 50%/50% less reserved memory or 75%/87% less allocated memory on the S2EF and IS2RE tasks respectively (depending on the configuration of DeepSpeed) compared to standard data parallel training. In terms of runtime, however, in almost all of our evaluations we were not able to speed up training as communication overhead increased when nontrivial DeepSpeed features were activated. Runtime-wise, just training in half precision has mostly yielded the best results.

In our GemNet evaluation, stage 3 of the ZeRO optimizer which corresponds to full model parallelism on top of data parallelism did not outperform the simpler stages 1 and 2 neither with respect to memory nor runtime. We suspect that for stage 3, being originally designed for trillion-parameter transformer models that cannot even be stored on single GPUs, our models were simply too small to see any memory savings.

In contrast, our results for DimeNet++ were rather unexpected as communication between the processes got very slow when we activated stage 1 and 2 of ZeRO. We ruled out various potential problems related to our DeepSpeed configuration and the model itself and suspect that this might be a problem related to the NCCL communication on our evaluation machine.

Since each configuration was only trained for one epoch and not all of the OC20 data was used for training due to time constraints, it would be interesting to see in the future if our results also prove true for longer training. It also remains yet to be evaluated whether our parallelization optimizations resulted in decreased model performance, as none of the two models was fully trained to completion and tuned for top-performance. In addition, it could also be very interesting to explore if the results we obtained for our GemNet setup can be transferred to the current state-of-the-art architecture, GemNet-OC, which does not only consider 3- but also 4-way interactions. Apart from all of this, we would be curious to see some of the DeepSpeed features being combined with other parallelization strategies in the domain of atomic simulations, like graph parallelism [Sri+22] which was also added to OCP recently during our project phase.

References

[Bai+21] Mandeep Baines et al. *FairScale: A general purpose modular PyTorch library for high performance and large scale training*. https://github.com/facebookresearch/fairscale. 2021.

[Bat+18] Peter W. Battaglia et al. *Relational inductive biases, deep learning, and graph networks*. 2018. DOI: 10.48550/ARXIV.1806.01261. URL: https://arxiv.org/abs/1806.01261.

[BT14] Kyle A. Baseden and Jesse W. Tye. "Introduction to Density Functional Theory: Calculations by Hand on the Helium Atom". In: *Journal of Chemical Education* 91.12 (2014), pp. 2116–2123. DOI: 10.1021/ed5004788. eprint: https://doi.org/10.1021/ed5004788. URL: https://doi.org/10.1021/ed5004788.

[Cha+21] Lowik Chanussot et al. "Open Catalyst 2020 (OC20) Dataset and Community Challenges". In: *ACS Catalysis* 11.10 (May 2021), pp. 6059–6072. DOI: 10.1021/acscatal.0c04525. URL: https://doi.org/10.1021%2Facscatal.0c04525.

[FL19] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. DOI: 10.48550/ARXIV.1903.02428. URL: https://arxiv.org/abs/1903.02428.

[Gas+20] Johannes Gasteiger et al. *Fast and Uncertainty-Aware Directional Message Passing for Non-Equilibrium Molecules*. 2020. DOI: 10.48550/ARXIV.2011.14115. URL: https://arxiv.org/abs/2011.14115.

[Gas+22] Johannes Gasteiger et al. *How Do Graph Networks Generalize to Large and Diverse Molecular Systems?* 2022. DOI: 10.48550/ARXIV.2204.02782. URL: https://arxiv.org/abs/2204.02782.

[GBG21] Johannes Gasteiger, Florian Becker, and Stephan Günnemann. *GemNet: Universal Directional Graph Neural Networks for Molecules*. 2021. DOI: 10.48550/ARXIV.2106.08903. URL: https://arxiv.org/abs/2106.08903.

[Gil+17] Justin Gilmer et al. *Neural Message Passing for Quantum Chemistry*. 2017. DOI: 10.48550/ARXIV.1704.01212. URL: https://arxiv.org/abs/1704.01212.

[Har+18] Aaron Harlap et al. *PipeDream: Fast and Efficient Pipeline Parallel DNN Training*. 2018. DOI: 10.48550/ARXIV.1806.03377. URL: https://arxiv.org/abs/1806.03377.

[Hua+18] Yanping Huang et al. *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*. 2018. DOI: 10.48550/ARXIV.1811.06965. URL: https://arxiv.org/abs/1811.06965.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2017. DOI: 10.48550/ARXIV.1706.02216. URL: https://arxiv.org/abs/1706.02216.

[KGG20] Johannes Klicpera, Janek Groß, and Stephan Günnemann. "Directional Message Passing for Molecular Graphs". In: *CoRR* abs/2003.03123 (2020). arXiv: 2003.03123. URL: https://arxiv.org/abs/2003.03123.

[Ott+19] Myle Ott et al. "fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: *Proceedings of NAACL-HLT 2019: Demonstrations*. 2019.

[Raj+19] Samyam Rajbhandari et al. *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models*. 2019. DOI: 10.48550/ARXIV.1910.02054. URL: https://arxiv.org/abs/1910.02054.

[Raj+21] Samyam Rajbhandari et al. "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning". In: *CoRR* abs/2104.07857 (2021). arXiv: 2104.07857. URL: https://arxiv.org/abs/2104.07857.

[Ren+21] Jie Ren et al. "ZeRO-Offload: Democratizing Billion-Scale Model Training". In: *CoRR* abs/2101.06840 (2021). arXiv: 2101.06840. URL: https://arxiv.org/abs/2101.06840.

[Sha+18] Noam Shazeer et al. *Mesh-TensorFlow: Deep Learning for Supercomputers*. 2018. DOI: 10.48550/ARXIV.1811.02084. URL: https://arxiv.org/abs/1811.02084.

[Sho+19] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2019. DOI: 10.48550/ARXIV.1909.08053. URL: https://arxiv.org/abs/1909.08053.

[Sri+22] Anuroop Sriram et al. *Towards Training Billion Parameter Graph Neural Networks for Atomic Simulations*. 2022. DOI: 10.48550/ARXIV.2203.09697. URL: https://arxiv.org/abs/2203.09697.

[Tra+22] Richard Tran et al. *The Open Catalyst 2022 (OC22) Dataset and Challenges for Oxide Electrocatalysis*. 2022. DOI: 10.48550/ARXIV.2206.08917. URL: https://arxiv.org/abs/2206.08917.

[Vel+17] Petar Velickovic et al. *Graph Attention Networks*. 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[Wan+19] Minjie Wang et al. *Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks*. 2019. DOI: 10.48550/ARXIV.1909.01315. URL: https://arxiv.org/abs/1909.01315.

Appendices

A Profiling Framework

We implemented a custom profiling framework for PyTorch to collect data for detailed runtime and memory usage analysis. Our profiler contains three components:

1. *Resource monitor thread*: Runs in parallel to the data parallel training processes and tracks CPU memory usage, as well as memory usage on each GPU. The monitor takes snapshot at a configurable time interval (2 seconds for all our runs).

2. *Phase clocking component*: Tracks runtime of all phases during each training step, as well as total epoch time. The phases include time spent on fetching data from the dataloader, forward pass, and backward pass. All phase times are tracking through clocks which are activated when the training script enters a specific phase and deactivated when it leaves the phase.

3. *PyTorch internal memory tracker*: Tracks allocated and reserved memory from PyTorch directly. This allows the profiler to detect how much GPU memory is used only for tensors and not any other components of the framework.

The profiler is implemented as part of the OCP codebase. The implementation of each component is located in the tracking package[14].

Our profiler is not an alternative to the official PyTorch profiler, but rather a complementary module to gather data on long running experiments. For shorter experiments, which just run over the span of a few training steps, the PyTorch profilier offers far more detailed data collection and better visualization support through a Tensorboard plugin.

[14] https://github.com/TUM-DI-Lab-Graph-Scaling/ocp/tree/main/ocpmodels/tracking

B Per-Experiment Memory Usage

The figures below show memory utilization for CUDA tensors averaged over all GPUs over time. Allocated memory is blue, while reserved memory is orange.

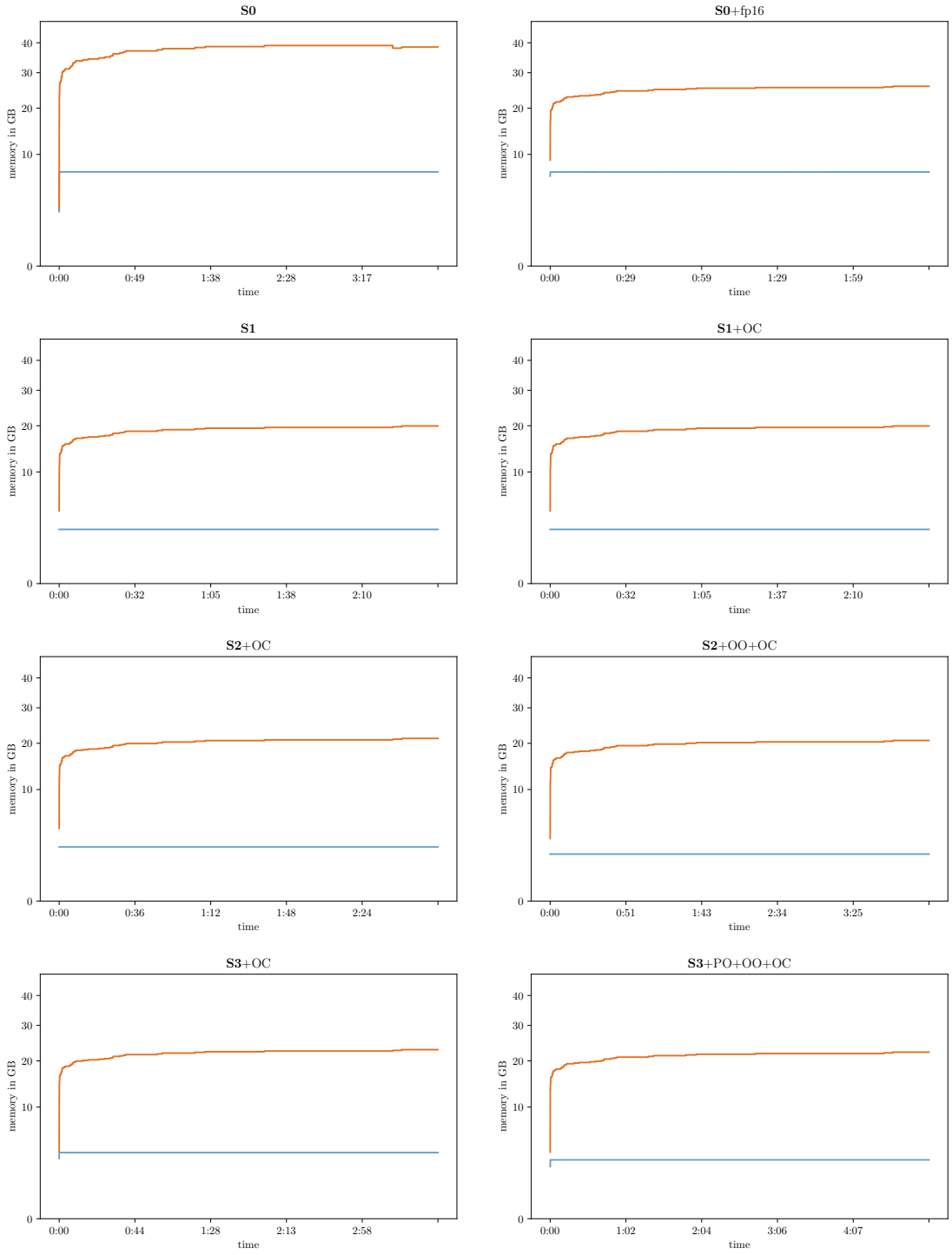


Figure 10: GPU memory consumption for GemNet on the S2EF task.

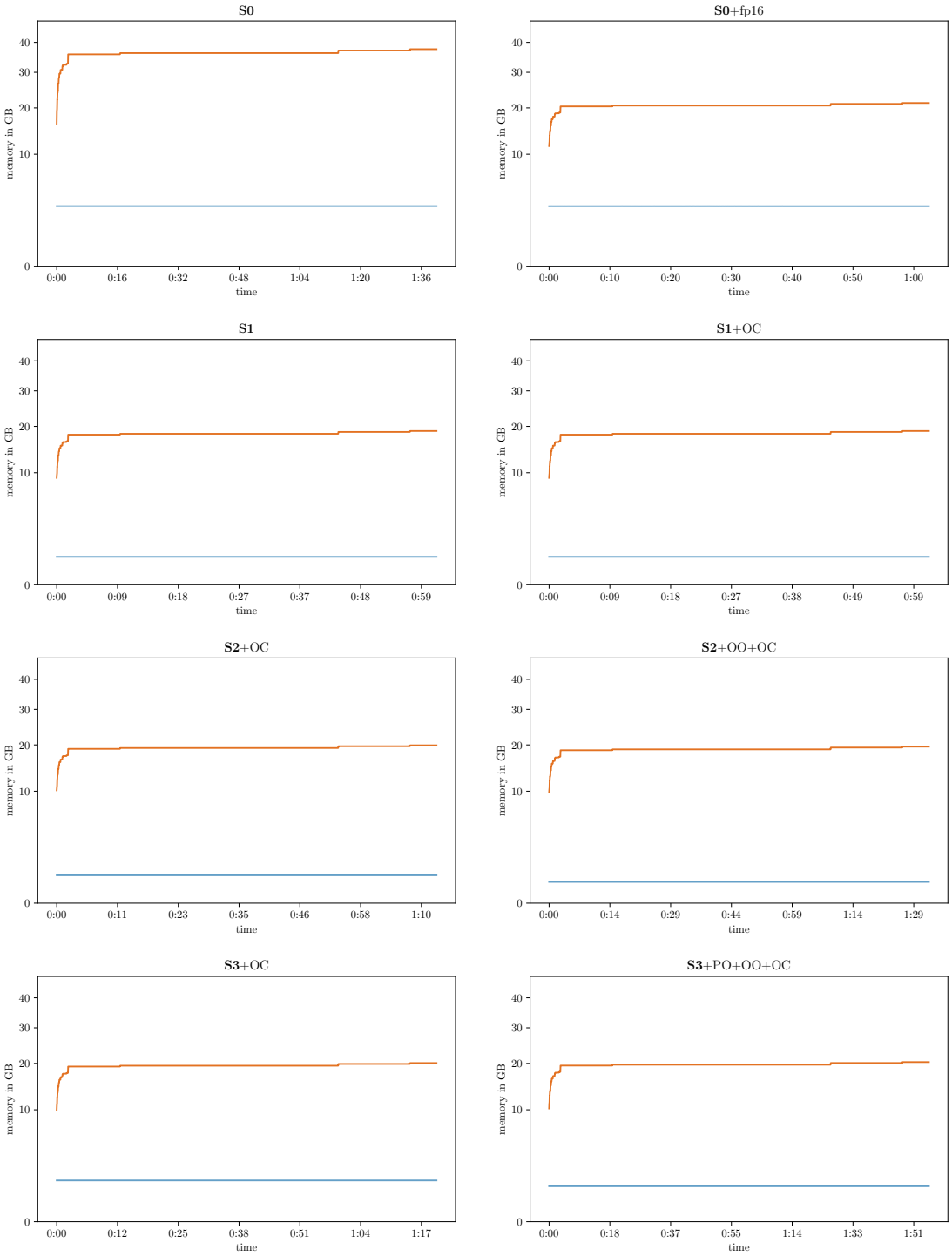


Figure 11: GPU memory consumption for GemNet on the IS2RE task.

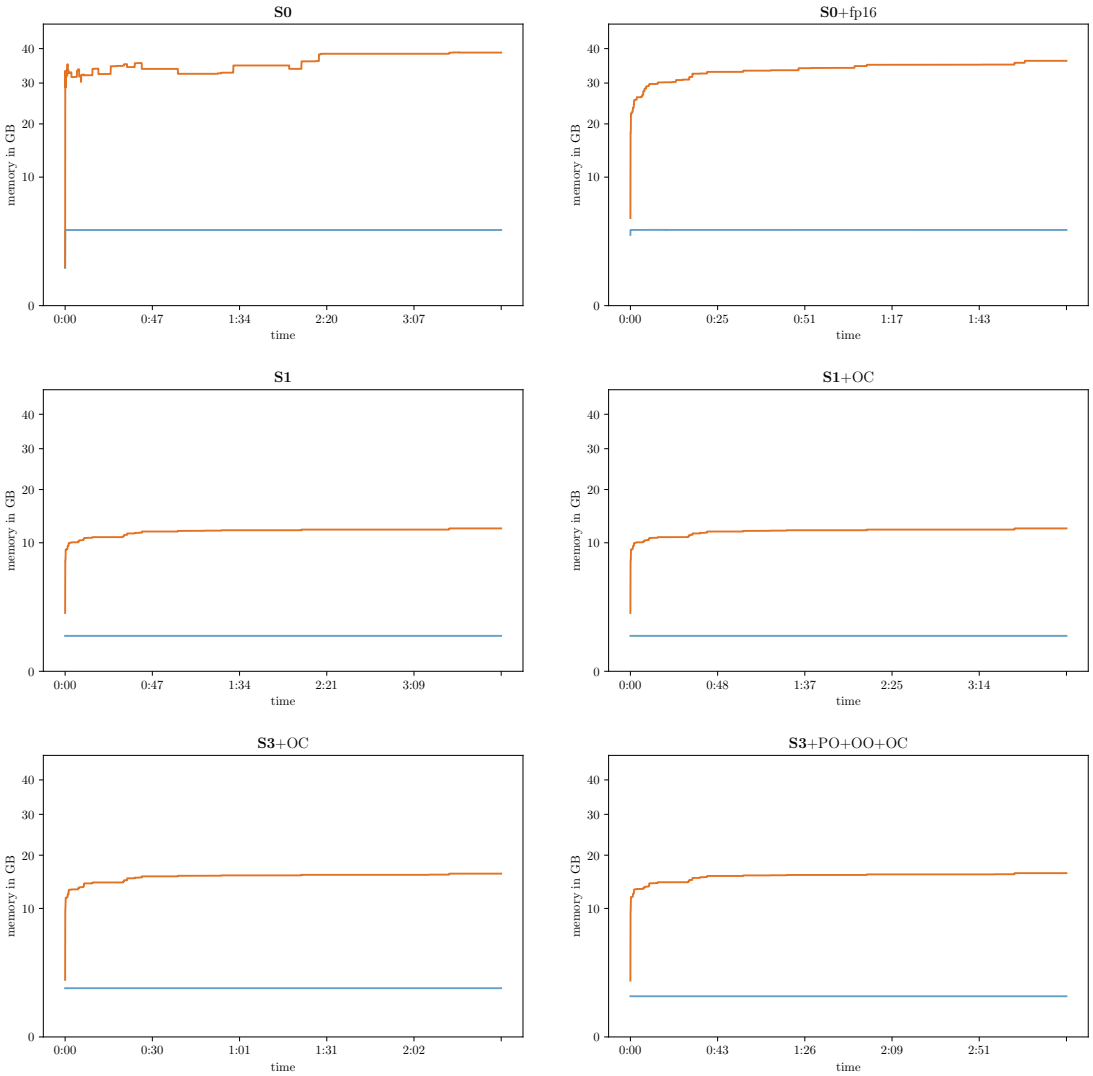


Figure 12: GPU memory consumption for DimeNet++ on the S2EF task.

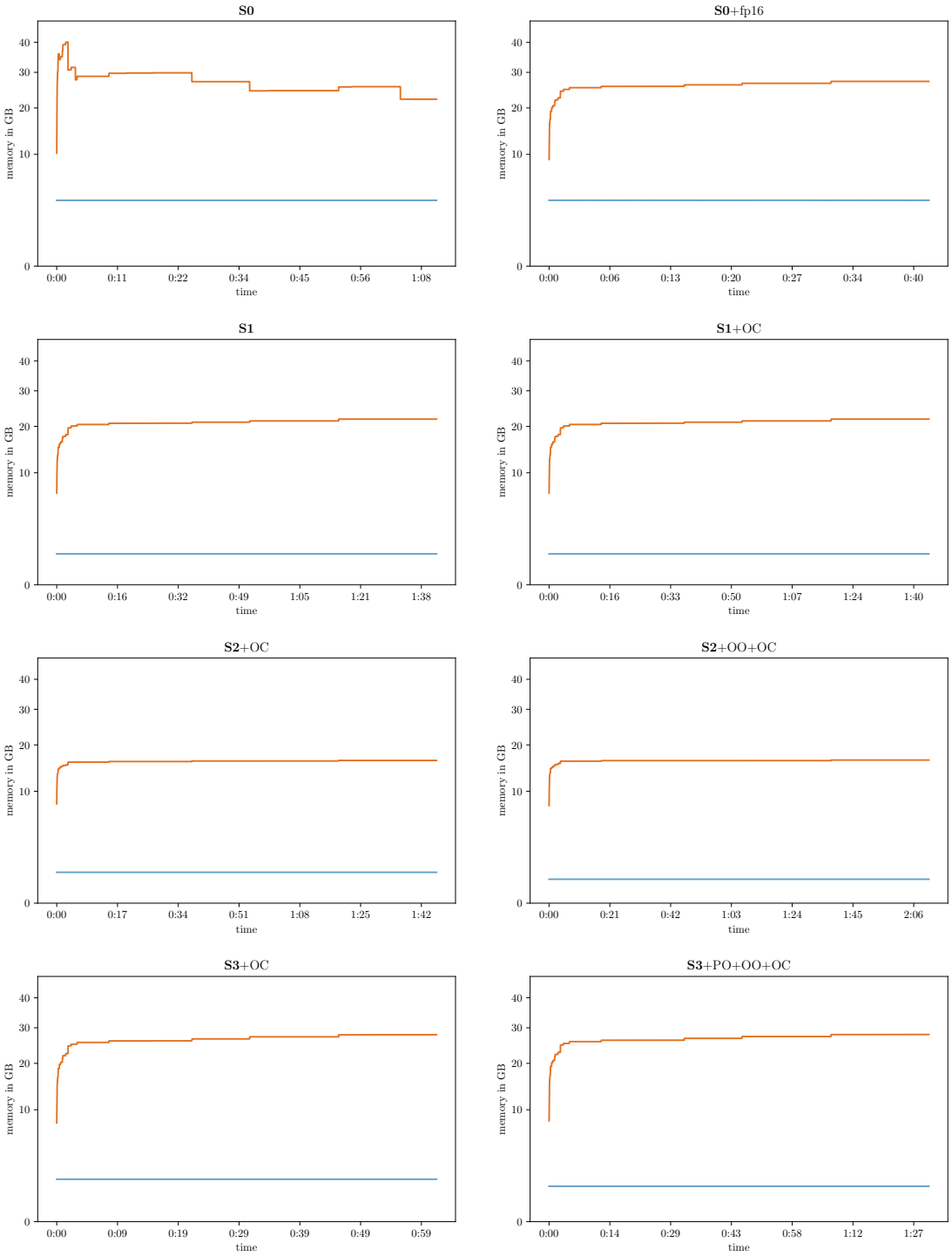


Figure 13: GPU memory consumption for DimeNet++ on the IS2RE task.