ScaleFold: Reducing AlphaFold Initial Training Time to 10 Hours

Feiwen Zhu*
NVIDIA
Shanghai, China
mzhu@nvidia.com

Arkadiusz Nowaczynski*
NVIDIA
Warsaw, Poland
anowaczynski@nvidia.com

Rundong Li
NVIDIA
Shanghai, China
davidli@nvidia.com

Jie Xin
NVIDIA
Shanghai, China
jxin@nvidia.com

Yifei Song
NVIDIA
Shanghai, China
yifeis@nvidia.com

Michal Marcinkiewicz
NVIDIA
Warsaw, Poland
michalm@nvidia.com

Sukru Burc Eryilmaz
NVIDIA
Santa Clara, United States
seryilmaz@nvidia.com

Jun Yang
NVIDIA
Beijing, China
juney@nvidia.com

Michael Andersch
NVIDIA
Berlin, Germany
mandersch@nvidia.com

ABSTRACT

AlphaFold2 has been hailed as a breakthrough in protein folding. It can rapidly predict protein structures with lab-grade accuracy. However, its training procedure is prohibitively time-consuming, and gets diminishing benefits from scaling to more compute resources. In this work, we conducted a comprehensive analysis on the AlphaFold training procedure, identified that inefficient communications and overhead-dominated computations were the key factors that prevented the AlphaFold training from effective scaling. We introduced ScaleFold, a systematic training method that incorporated optimizations specifically for these factors. ScaleFold successfully scaled the AlphaFold training to 2080 NVIDIA H100 GPUs with high resource utilization. In the MLPerf HPC v3.0 benchmark, ScaleFold finished the OpenFold benchmark in 7.51 minutes, shown over 6× speedup than the baseline. For training the AlphaFold model from scratch, ScaleFold completed the pretraining in 10 hours, a significant improvement over the seven days required by the original AlphaFold pretraining baseline.

CCS CONCEPTS

• **Theory of computation** → **Distributed computing models**; • **Applied computing** → *Molecular sequence analysis*; • **Computing methodologies** → *Neural networks*.

KEYWORDS

AI for Science, Alphafold, Protein Folding, Distributed Training, GPU, High Performance Computing

*Equal contributions.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657326

ACM Reference Format:
Feiwen Zhu, Arkadiusz Nowaczynski, Rundong Li, Jie Xin, Yifei Song, Michal Marcinkiewicz, Sukru Burc Eryilmaz, Jun Yang, and Michael Andersch. 2024. ScaleFold: Reducing AlphaFold Initial Training Time to 10 Hours. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657326

1 INTRODUCTION

Predicting the three-dimensional (3D) protein structures from amino acid sequences has been an important long-standing question in bioinformatics. In recent years, deep-learning based computational methods have been emerging and show promising results [5, 7, 8]. Among these lines of work, AlphaFold [7] is the first method that achieved atomic accuracy. The AlphaFold model was built on a variant of the sequence attention mechanism [11] widely adopted by other contemporary deep-learning models. The AlphaFold model was trained on over 10 million samples with 128 TPUs, took over 11 days to converge. 7 days of this is initial training and the rest is finetuning. Such a low efficiency slows down the iterative speed of the research community.

As a result, improving the AlphaFold training performance has received increasing interest, and MLPerf HPC [6] also incorporated this challenge as a benchmark to promote broader participation. OpenFold [1] reproduced the AlphaFold training procedure and used BFloat16 numerical format and gradient checkpointing [2] to improve the training efficiency. DeepSpeed4Science [9] proposed a dedicated attention kernel design that reduced memory usage. FastFold [3] proposed Dynamic Axial Parallelism to parallelize the training with finer granularity. However, to our knowledge, none of the existing work has revealed the fundamental challenges in further accelerating the AlphaFold training.

We point out that the core challenge of improving the AlphaFold training performance is *scalability*. In this study, we conducted a comprehensive analysis on the AlphaFold training procedure, and show the major causes that prevented the training from scaling to more compute resources are: 1) *Communications* during the distributed training were intensive yet inefficient, largely due to

communication overheads and imbalances caused by slow workers; 2) *Computation* in the training hardly saturated each worker's compute resources, due to local CPU overheads, non-parallelizable workloads, and poor kernel scalability.

We proposed a collection of systematic optimizations to address these challenges. A novel non-blocking data pipeline was introduced to solve the slow-worker issue. By combining this with a series of fine-grained optimizations, which include tracing the training to CUDA Graphs to reduce overheads, the overall communication efficiency was largely improved. We identified critical computation patterns in the AlphaFold training and designed dedicated kernels for each of them, fused fragmented computations throughout the AlphaFold model, and carefully tuned kernel configurations for every workload sizes and target hardware architectures. We named the training method that incorporates these optimizations *ScaleFold*.

ScaleFold successfully addressed the scalability issue and scaled the AlphaFold training to 2080 NVIDIA H100 GPUs, whereas prior arts only scaled up to 512. In the MLPef HPC v3.0 benchmark, ScaledFold finished the OpenFold partial training task in 7.51 minutes, over 6× faster than the benchmark baseline. For training the AlphaFold model from scratch, ScaleFold finished the pretraining in 10 hours, set a new record compared to prior works.

In summary, contributions of this work are three-fold:

1) We identified the key factors that prevented the AlphaFold training from scaling to more compute resources;
2) We introduced ScaleFold, a scalable and systematic training method for the AlphaFold model;
3) We empirically demonstrated the scalability of ScaleFold, set new records for the AlphaFold pretraining and the MLPef HPC benchmark.

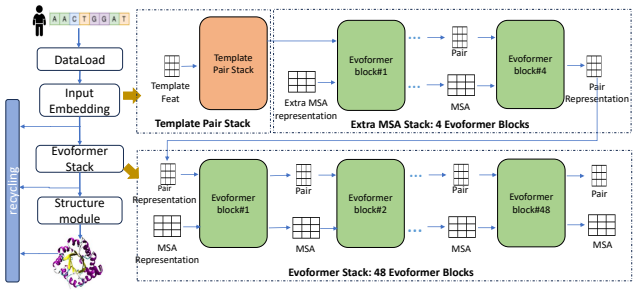
2 BACKGROUND

2.1 The AlphaFold Model

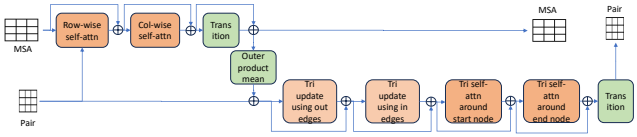
AlphaFold [7] is the first computational method that can regularly predict protein 3D structures from amino acid sequences with an atomic accuracy. AlphaFold introduced a novel mechanism to exchange information within the amino acid's multi-sequence alignments (MSA), and explicitly represented the 3D structure in the form of a rotation and translation for each residue of the protein.

The structure of AlphaFold model is illustrated in Figure 1a. *Data loading* module prepares the input, MSA and template sequences, crops these sequences to a predefined length. *Input Embeddings* module encodes MSA and template features of the input sequence into the initial MSA and pair representations. *Evoformer Stack* module iteratively refines internal MSA and pair representations. *Structure Module* outputs 3D structure for atoms in each residue. The AlphaFold training incorporates a *recycling* process, enabling the continuous enhancement and fine-tuning of predicted protein structures. The core building block of the AlphaFold model is *Evoformer*, its structure is illustrated in Figure 1b.

OpenFold [1] is a faithful reproduction of AlphaFold with a fully open-sourced training procedure and training dataset. In the rest of this work, *the AlphaFold training* refers to the procedure reproduced by OpenFold.



(a) Structure of the AlphaFold model.



(b) Structure of the Evoformer block.

Figure 1: Evoformer is the main building block of the AlphaFold model. In the AlphaFold model, Input Embeddings consist of Template Pair Stack, which contains 2 Evoformer blocks. Extra MSA Stack contains 4 Evoformer blocks. Evoformer stack contains 48 Evoformer blocks.

2.2 Challenges of the AlphaFold Training

Given the significance of the AlphaFold model, the AlphaFold training has been incorporated in the MLPerf HPC v3.0 benchmark [6]. This training procedure is highly challenging in many perspectives:

High Memory Consumption. The AlphaFold model has only 97M parameters but the volume of intermediate activations during training is enormous. This is attributed to the unique attention mechanism of Evoformer, which consumes $O(n^3)$ memories for each call (see § 1.6 of the supplementary to [7]), significantly higher than the $O(n^2)$ memory consumption than the normal Transformer [11] based models. OpenFold used gradient checkpointing [2] to mitigate this issue, yet at the cost of sacrificing the training speed.

Massive Memory-Bounded Kernels. Each step of the AlphaFold training launches over 150,000 operators. Profiling results of these kernels are listed in Table 1. In this table, matrix-matrix multiplications are categorized as math-bounded kernels. Memory copy and set are categorized as memory-operation. The rests are categorized as memory-bounded kernels. The number of calls to memory-bounded kernels far exceeds that of math-bounded kernels, and they take over 65% of the training time.

Table 1: Breakdown of kernels launched in the AlphaFold training. Most of these kernels are memory-bounded.

Kernel Type	Runtime (%)	#Calls
CPU Overhead	9.10	–
Math-bounded	24.06	18,147
Memory-bounded	65.03	97,749
Memory-operation	1.82	34,991

Suboptimal Key-Operation Performance. In the AlphaFold training, multi-head attention (MHA) and layer-normalization (LN) are the two major performance critical operations, each of them takes 34% and 14% of the total training time, respectively. However, after carefully profiling the OpenFold training implementation, we found that MHA only reached 26% of the theoretical performance, and LN only reached 10%. In addition, training routines such as optimizer update and parameter gradient clipping were far from optimal.

Limited Data-Parallel (DP) Degrees. DP is the most widely used strategy to scale training to multiple workers by splitting along the batch dimension of the training samples. However, it has been shown [1] that the training batch size of AlphaFold cannot exceed 256, otherwise it would fail to converge. This set a hard limit to the DP scaling degrees.

2.3 Dynamic Axial Parallelism

To scale the AlphaFold training beyond the hard limit imposed by DP, FastFold [3] proposed Dynamic Axial Parallelism (DAP). DAP splits intermediate activations and associated computations of a single training sample along a non-reductive axis, introducing another layer of parallelism under DP. However, DAP requires additional communication at both forward and backward, and its scaling efficiency is suboptimal.

3 SCALE THE ALPHAFOLD TRAINING

In this study, we first comprehensively analyzed the AlphaFold training performance in various parallelism strategies and identified crucial factors which prevented the AlphaFold training from scaling to more compute resources. This part of study is elaborated in § 3.1. We then divided above factors into two categories, one for those impact the communication scalability, and another for impact the computation scalability. Our systematic solutions to address issues in each of these categories are presented in § 3.2 and § 3.3, respectively. Combined with other optimizations we proposed, which are elaborated in § 3.4, we finally scaled the AlphaFold training to 2080 NVIDIA H100 GPUs and finished the pretraining in 10 hours. We named this training method as *ScaleFold*.

3.1 Barriers to AlphaFold's Training Scalability

We observed that existing approaches for scaling the AlphaFold training was suboptimal. Applying Dynamic Axial Parallelism [3] with DAP-2 and DAP-4 to a 128-way data-parallel training only provided 1.42× and 1.57× speedup, respectively. And we did not observe performance gains on DAP-8. Ideally, the scalability of DAP-n would provide a n× speedup.

To determine the root causes of these gaps, we ablated the contribution from each potential factor by subtracting the measured step time with the corresponding theoretically optimal time. The optimal time was calculated by assuming the mentioning factor was completely eliminated. The breakdown of analysis results on different DAP-n are illustrated in Figure 2.

In the small scale of DAP-2, the CPU overhead and execution of serial modules were the major limiting factors. *CPU overhead* means the cost of launching numerous small kernels sequentially

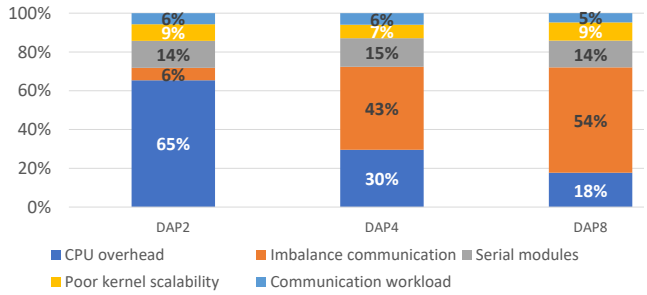


Figure 2: Breakdown of factors that prevent the AlphaFold training from achieving better scalability. Numbers indicate the relative difference between the actual time and the theoretically optimal time per training step.

during training. This cost is largely attributed to the nature of AlphaFold model we discussed in § 2.2. *Serial modules* means the part of computation which cannot be parallelized. In AlphaFold, this corresponds to the data pipeline and the Structure Module, which takes 11% of GPU time in total per training step.

In larger scales of DAP-4 and DAP-8, the impact of imbalanced communication became increasingly substantial. *Imbalanced communication* means that slow workers that fall behind the rest in reaching the synchronization point slow down the overall training progress. In AlphaFold training, this is mainly attributed to: 1) data pipeline, where ~ 10% of training data batches took significantly more time to process thus blocked the training pipeline, as shown in Figure 3; and 2) background processes in the cluster environment, which sporadically made CPU peaks and slowed down the corresponding workers. This issue is particularly troublesome to DAP, as DAP involves numerous additional communications in both forward and backward passes.

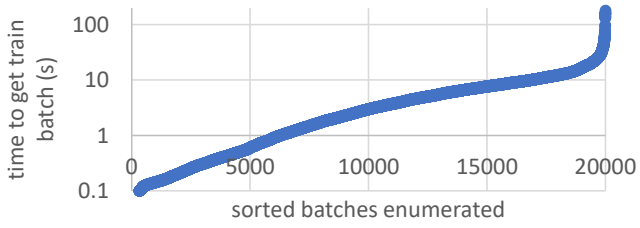


Figure 3: Sorted data batch preparation time of AlphaFold's training dataset. Depending on the data sample's initial sequence length and multi-sequence alignment size, the batch preparation time varies significantly, which could cause data pipeline blocking.

In addition, we observed *poor kernel scalability* as each kernel's problem sizes reducing. DAP-n reduces kernel workload by n×. Small workload is hard to saturate GPU bandwidth, which make kernel scalability become worse. We also observed *communication overhead* associated with DAP's all-gather and all-to-all communications. This is a minor factor and can be reduced by low precision.

3.2 Reduce Communication Imbalance

To reduce the communication imbalance caused by accidental data pipeline blocking and cluster machine CPU peaks, we proposed a new data pipeline design that continuously fed data batches to the main training process, along with a module wrapper that captured the AlphaFold training in CUDA Graphs to make the training process more robust to machine CPU usage fluctuations.

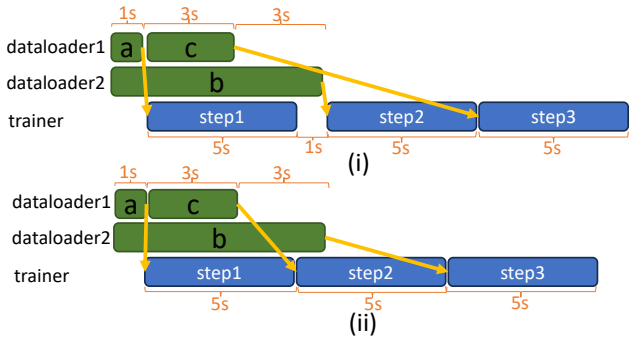


Figure 4: (i) The default PyTorch data loading pipeline vs (ii) our proposed pipeline. By default, the sampler order is enforced by its `DataLoader` even if it blocks the training: Slow batch *b* blocks training even though another batch *c* is available. In our proposed design: The batch *c* can be yielded before batch *b*, which prevents imbalance and idle ranks.

Non-Blocking Data Pipeline. In the AlphaFold training, the default data pipeline built on PyTorch `DataLoader` generates data batches in a deterministic order. However, if the time for preparing these batches varies significantly and certain batch's preparation time exceeds the training step time a lot, that batch could be blocking the training process and make other communication participants hang. E.g., in the situation demonstrated in Figure 4 (i), the slow batch *b* takes 7 seconds to process, while the training step has finished at the 6th second; As a result, the training process is blocked during the whole last second.

We proposed a data pipeline that yields a batch once any of the processing batches becomes ready. This design effectively resolved the aforementioned data pipeline blocking issue. As demonstrated in Figure 4 (ii), in the same situation discussed in (i), when the first training step finished at the 6th second, the data pipeline immediately yields the ready batch *c* while leaving the slow batch *b* being processed, so the training immediately proceed; When the second training step finished at the 11th second, the batch *b* has been ready and the data pipeline feeds it to the next training step.

Processed data batches were sent to the training process via a priority queue, with the batches' indices as the associated priorities. This ensured the data yielding order in a *best effort* extent. The overall data sample order could thus vary across different training instances. However, in our experiments, we did not observe any evidence showing this could hurt the training convergence.

CUDA Graph. CUDA Graph eliminates the need to interact with the CPU after graph capture, thus greatly improves training performance robustness against the CPU usage peaks. A typical way to use CUDA Graph is to define a scope, capture the computational graph within this scope, and execute the optimized graph. However, if the CUDA kernels within this scope are modified due to dynamic executions, such as the recycling mechanism in the AlphaFold training, CUDA Graph needs to be recaptured. To address this, we designed a CUDA Graph cache that can capture multiple graphs for different recycling scenarios.

In addition, we anecdotally found that disabling Python garbage collection at runtime could alleviate machine CPU usage peaks and accelerate the overall training progress.

3.3 Improve Computation Efficiency

To reduce the massive CPU overheads and improve the kernels' performance when their workload sizes are scaled down, we conducted both manual and automatic kernel fusions to reduce the kernel number and memory I/O. We also utilized the OpenAI Triton compiler [10] to search for optimal configurations for all generated kernels.

3.3.1 Manual Operator Fusion and Optimization. Kernel fusion is a common optimization methodology, which combines adjacent operations' logic into a single kernel, reducing the intermediate data movement and associated kernel launching overheads. We implemented several efficient fused Triton kernels for AlphaFold training.

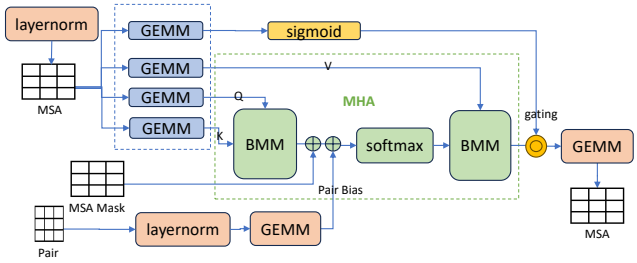


Figure 5: Structure of MSARowAttentionWithPairBias module. Its main structure is MHA.

Multi Head Attention (MHA). MHA takes 34% of step time in the AlphaFold training. AlphaFold uses a special variant of MHA, where a *pair bias* term is added to the logits matrix before the softmax operation, as shown in the dashed green box in Figure 5. This makes integrating existing optimized MHA implementations such as FlashAttention [4] inapplicable. We implemented a customized kernel to fuse all operations in MHA and harvested a considerable speedup.

GEMM Batching. In most AlphaFold model's building blocks, the matrix-matrix multiplications (GEMMs, the dashed blue box in Figure 5) prior to MHA do not fully leverage the potential parallelism. Four linear layers have no dependency on each other. We bundled these linear layers into batch operations to improve the degree of parallelism.

Layer-Normalization (LN). LN takes 14% of step time in the AlphaFold training. AlphaFold's typical LN dimensions are small (128 and 256), DAP further reduces problem sizes, preventing LN from fully utilizing GPU resources. We implemented a customized LN

kernel to increase GPU utilization: 1) In the forward pass, we allowed each CUDA thread block to process multiple input rows; 2) The normalization statistics were computed in a single pass, instead of using expensive iterative methods; 3) In the backward pass, weight and bias gradients were computed by a two-step reduction, where at the first step each CUDA thread block reduced a sub-region of upstream gradients to an intermediate buffer, then at the second step each column of this buffer was reduced to obtain the final weight and bias gradients. This design effectively avoided expensive atomic operations.

3.3.2 Automatic Fusion and Tuning. To further harvest the remaining optimization opportunities, we exploited the fusion ability provided by the `torch.compile` compilation stack in PyTorch-2.0. We used it to automatically capture and fuse the fragmented operations throughout the AlphaFold model, significantly accelerated serial modules such as the Structure Module. We also found that `torch.compile` did not always generate the most efficient kernels, so we controlled the compilation scope according to the target GPU architecture.

In our customized MHA and LN kernels, the OpenAI Triton compiler's auto tuning ability was exploited to search for the optimal hyper-parameters for all workload sizes that appear and target GPU architectures. The search space spanned a set of predefined tiling sizes and kernel launching dimensions. We found this to be particularly useful when workload sizes were scaled down by DAP.

3.4 Mitigate Other Limiting Factors

We found a surprisingly vast volume of optimization opportunities beyond the AlphaFold model itself. Due to the fragmented nature of the AlphaFold model, the ordinary training subroutines such as Adam optimization, stochastic weight average (SWA) and gradient clipping (grad-clip) together took 15% of the training time. We implemented a customized kernel that fused Adam, SWA, along with other adjacent element-wise training logic, and parallelized this kernel across all trainable parameters. We also reordered the gradient norm calculation in grad-clip, such that its latency was perfectly hidden by distributed training communications.

As we continuously optimize step time, the proportion of evaluation time to the total training time continues to increase from 22% to 43%, as illustrated in Figure 7. To eliminate the time for the evaluation, we implemented asynchronous evaluation, which can offload evaluation to separated nodes and free training nodes from executing evaluation work. Evaluation time must be smaller than training time, or evaluation time would become bottleneck. so we cached all evaluation data into the CPU DRAM instead of disk to improve evaluation performance.

4 EXPERIMENT RESULTS

This section presents a comprehensive evaluation of ScaleFold's performance on the NVIDIA A100 and H100 GPUs. Initially, we compared our performance to other AlphaFold2-like models. Then we assess our scale efficiency. Subsequently, we conducted a step-by-step evaluation of performance optimizations. Lastly, we evaluate the overall training time of ScaleFold on Eos cluster[1]. We used CUDA version 12.2 and PyTorch NVIDIA Release 23.09 in our experiments. Each MPI task is bound to an individual GPU and 28 hyper-threading CPU cores to fully exploit CPU-GPU affinity. 8 MPI tasks are bound to a node. And all the experiments are conducted on the OpenFold dataset.

[1]NVIDIA Eos supercomputer consists of 10,752 NVIDIA H100 Tensor Core GPUs and NVIDIA Quantum-2 InfiniBand networking.

4.1 Step Time Evaluation

We compared the step time of our implementation to public OpenFold and FastFold. The results, as presented in Table 2, demonstrating that the training performance of ScaleFold outperforms others. Public OpenFold doesn't support DAP and its step time on A100 is 6.19s, and the step time of FastFold DAP-2 is 2.49s on A100, while our DAP-2's step time on A100 is 1.88s. On H100, ScaleFold DAP-{2, 4, 8} show 1.6×, 2.4× and 2.77× speedup comparing to DAP-∅.

Table 2: Comparison of per-training-step time (in second) between ScaleFold and other baselines. ∅ indicates not applying dynamic axial parallelism (DAP).

Method	DAP	GPU	
		A100	H100
OpenFold	∅	6.19	–
FastFold	2	2.49	–
ScaleFold	∅	–	1.80
	2	1.88	1.12
	4	–	0.75
	8	1.21	0.65

We also evaluated speedups from each of the optimizations, as shown in Figure 6. Porting the baseline to H100 GPUs introduced 1.66× speedup. GEMM batching further accelerated training by 1.03× and brought the overall speedup to 1.71×. The non-blocking data pipeline then brought the overall speedup to 1.78×. Applying BF16 significantly reduced memory workload and the overall speedup went to 2.22×. Replacing default layer-norm, SWA and MHA kernels with our custom implementations accelerated the overall speedup to 3.29×. Increasing DAP degree to 8 further reduced memory footprint, made applying CUDA Graphs and disabling gradient checkpointing feasible. Together they brought the overall speedup to 5.90×. Reordering grad-clip and disabling Python GC reduced CPU overhead, brought the overall speedup to 8.91×. Finally, applying `torch.compile` fused remaining fragmented kernels and brought the final speedup to 10.39×.

4.2 Time To Train Evaluation

We evaluated ScaleFold using MLPerf HPC 3.0 benchmark setting on Eos. Our largest scale was extended to 2080 NVIDIA H100 GPUs, which 2048 GPUs are used for training, and rests are used for evaluation. At largest scale, the time to train of ScaleFold was reduced to 8 minutes, including ~2 minutes initialization and compilation overhead, as Figure 7. ScaleFold is 6X faster than the reference model, as Figure 8. Without asynchronous evaluation optimization and 2048 NVIDIA H100 GPUs for both training and evaluation, the train to time increased to about 11 minutes. We also trained ScaleFold from scratch. Firstly, we used global batch size 128 to train

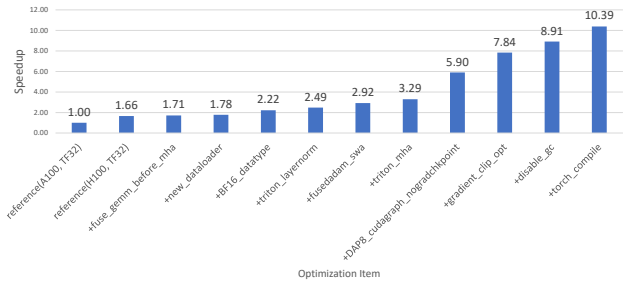


Figure 6: Relative speedups after incrementally applying each of the optimizations.

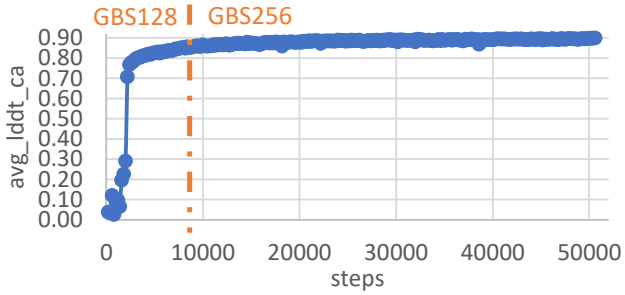


Figure 9: Convergence cruve of pretraining ScaleFold from scratch. Batch size 128 in early phase later switched to 256.

first 5000 steps on 1056 NVIDIA H100 GPUs, with 32 of them being used for evaluation. Training metric avg_lddt_ca must exceed 0.8 before first 5000 training steps. Then, we used global batch size 256 and disable Triton mha kernel to train the rest steps on 2080 NVIDIA H100 GPUs (with 32 of them being used for evaluation). The whole AlphaFold pretraining requires 50000 ~ 60000 steps to reach 0.9 avg_lddt_ca, which takes < 10 hours.

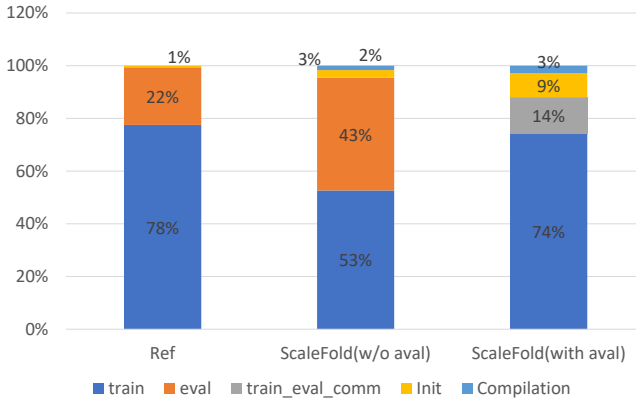


Figure 7: Breakdown Time to Train.

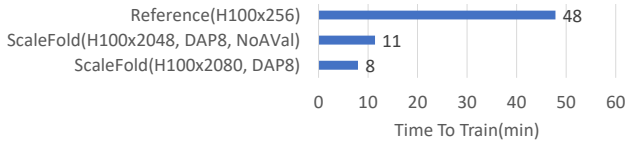


Figure 8: Time-to-train metrics in MLPerf benchmark. ScaleFold shown 6× speedup w/ all optimizations.

5 DISCUSSIONS

In this work, we identified that inefficient communications and overhead-dominated computations were the two major factors that prevented the AlphaFold training from scaling to more compute resources. We introduced ScaleFold, a systematic training method for the AlphaFold model, that incorporated solutions specifically for these challenges. By measuring ScaleFold's performance in the MLPerf HPC benchmark and in the AlphaFold pretraining, we demonstrated its scalability and efficiency: ScaleFold effectively scaled to 2080 NVIDIA H100 GPUs and set a new record of finishing the AlphaFold pretraining in 10 hours.

We hope this work can benefit the HPC and bioinformatics research community at large, by providing an effective regime to scale the deep-learning based computational methods to solve the protein folding problems. We also hope the workload profiling and optimization methodologies used in this work can shed lights on machine learning system designs and implementations.

REFERENCES

[1] Gustaf Ahdritz, Nazim Bouatta, Sachin Kadyan, Qinghui Xia, William Gerecke, Timothy J O'Donnell, Daniel Berenberg, Ian Fisk, Niccolò Zanichelli, Bo Zhang, et al. 2022. OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization. *bioRxiv* (2022), 2022–11.

[2] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[3] Shenggan Cheng, Ruidong Wu, Zhongming Yu, Binrui Li, Xiwen Zhang, Jian Peng, and Yang You. 2022. FastFold: Reducing AlphaFold Training Time from 11 Days to 67 Hours. (2022).

[4] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[5] Richard Evans, Michael O'Neill, Alexander Pritzel, Natasha Antropova, Andrew Senior, Tim Green, Augustin Žídek, Russ Bates, Sam Blackwell, Jason Yim, et al. 2021. Protein complex prediction with AlphaFold-Multimer. *biorxiv* (2021), 2021–10.

[6] Steven Farrell, Murali Emani, Jacob Balma, Lukas Drescher, Aleksandr Drozd, Andreas Fink, Geoffrey Fox, David Kanter, Thorsten Kurth, Peter Mattson, et al. 2021. MLPerf™ HPC: A holistic benchmark suite for scientific machine learning on HPC systems. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE, 33–45.

[7] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589.

[8] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. 2020. Improved protein structure prediction using potentials from deep learning. *Nature* 577, 7792 (2020), 706–710.

[9] Deepspeed team and OpenFold team. 2023. *DS4Sci_EvoformerAttention: eliminating memory explosion problems for scaling Evoformer-centric structural biology models.* https://deepspeed4science.ai/2023/09/18/model-showcase-openfold/

[10] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).