MLPerf HPC: Benchmarking Machine Learning Workloads on HPC Systems

Anonymous Author(s)

ABSTRACT

Scientific communities are increasingly adopting machine learning and deep learning models in their applications to accelerate scientific insights. High performance computing systems are pushing the frontiers of performance with a rich diversity of hardware resources and massive scale-out capabilities. There is a critical need to understand fair and effective benchmarking of machine learning applications that are representative of real-world scientific use cases emerging today. MLPerf is a community-driven standard to benchmark machine learning workloads, focusing on end-to-end performance metrics. In this paper, we introduce MLPerf HPC, a benchmark suite of large-scale scientific machine learning training applications. We present the results from the first submission round with systematic analyses of our findings across a diverse set of some of the world's largest HPC systems, characterizing each benchmark with compute, memory and I/O behaviours.

KEYWORDS

Deep Learning, Benchmarks, Supercomputers, Scientific Applications

1 INTRODUCTION

Scientific applications are leveraging the potential of AI to accelerate scientific discovery. This trend is prevalent in multiple domains, such as cosmology, particle physics, biology and clean energy. These applications are innately distinct from traditional industrial use cases regarding complexity of the models, volume, and type of data. There have been significant success stories where AI-driven applications led major insights on the scientific discoveries that would have taken many more years, otherwise. A recent breakthrough in addressing one such grand challenge in biology for 50 years was the development of an AI model that solved the protein folding problem with the AlphaFold tool by Google's Deepmind [47]. Scientific experiments will see a growth of data at an unprecedented scale with the advancement of large-scale experiments. The authors of [34] highlight the challenges faced with understanding the exponential experimental data and present opportunities of using machine learning techniques to advance science. The AI for Science report [50] put forth by stakeholders from leadership computing facilities, DOE labs, and academia details a vision for leveraging AI in science applications critical to US DoE missions. The vision outlined in this report emphasizes the need for a systematic understanding of how these applications perform on diverse supercomputers.

An ideal benchmark suite will help assess HPC system performance while driving innovation in systems and methods. Developing one is difficult because of the inherent trade-off between building generalizable and representative proxies of real AI workloads and building high-fidelity proxies that probe specific features of an HPC system. Benchmarks of the former type are general enough to capture what the average user on a large multi-user system is doing with AI while benchmarks of the latter type focus on how a specific kernel performs on the system. Scientific machine learning applications implement models such as 3D image segmentation with sparse datasets, UNet, and custom-designed models that are generally distinct from widely used traditional ones. Implementation of the complex models on supercomputers at full scale poses challenges that are not typically exposed on small-scale, say a few number of nodes. The massive datasets used by these applications exhibit stress the I/O subsystem. Hence, adopting existing benchmarking approaches for scientific machine learning problems would not be able to capture realistic behaviour of the scientific applications.

There have been significant efforts in benchmarking supercomputers with traditional HPC workloads with major ones listed in Table 1. TOP500[23] ranks supercomputers across the world and publishes their performance numbers (in Flops) with High Performance Linpack (HPL). It captures many of the general features that large-scale scientific applications share, such as domain decomposition and heavy use of linear algebra solvers. A single benchmark can be run across the entire system in a weak-scaling fashion. One of the limitations is that the HPL benchmark is not particularly relevant to address scientific machine learning applications. Green500 [48] ranks supercomputers based on their energy efficiency. The list reports the performance per rated watt using the LINPACK benchmark at double precision format. Similar to Top500, Green500 is currently limited by the characteristics of a benchmark that is not representative of the problems at scale.

Several benchmarking efforts have previously aimed to characterize performance of machine learning workloads, including Deep500 [26], HPCAI500 [35], and HPL-AI [10]. The largest dataset used across these attempts is obtained from the Extreme Weather Dataset [45], coming in at roughly 1.65 TB. These efforts are summarized in Table1. DAWNBench [32] is a benchmark suite for end-to-end deep learning training and inference with time-to-accuracy as the performance metric. The analyses [31] highlight that specialized hardware units such as tensor cores are heavily under-utilized. DeepBench [5], TBD [55], Fathom [24] provide a systematic approach to analyze deep learning applications across the stack of implementation on hardware. ParaDNN [52], HPE DLBS [9] provide a set of implementations of few neural networks where the target workloads are limited when running on large-scale supercomputers. XSP [40] proposed methods to characterize machine learning workloads on GPUs. Mahon et al. [41] evaluate different deep learning framework implementations on HPC systems. The challenges and limitations of existing benchmarking drive the need to develop a benchmark suite with science applications that run at scale.

In this paper, we present MLPerf HPC benchmark suite aimed to address the limitations of prior efforts. This is driven by MLCommons [11], an open engineering consortium that aims to accelerate

Table 1: HPC Machine Learning Benchmarks

Benchmark	Performance metrics	Application Domain	Data volume	Comments
HPL, HPL-AI	Flops, Flops/Watt	Random dense system of linear equations	Variable	Used in Top500 and Green500 to rank supercomputers. Problem size scaled to optimize the performance for machine size. HPL measures performance at double precision, HPL-AI measures performance in mixed precision
HPCAI500	Valid Flops, Valid Flops/Watt	Image classification, Weather analytics	150 GB and 1.65 TB	Convolution and GEMM layers measure the performance in valid Flops which impose penalty based on failure to meet target accuracy. Limited to Microbenchmarks, Object Detection and Image Classification tasks with microscopic view of common deep learning models (Faster-RCNN, ResNet)
Deep500	Throughput, Time to solution	any machine learning application	150 GB	Helps evaluate different framework implementations and multiple levels of operators. Challenging to integrate into scientific applications. Evaluated with ImageNet dataset.
MLPerf HPC	Time to train	Cosmology and Weather analytics	5.1 TB and 8.8 TB	Targets representative scientific machine learning applications with massive datasets. Provision of two types submissions, closed and open enable novel optimizations. Time to solution metric and focused timing captures holistic model performance

machine learning innovation through MLPerf benchmarks, public datasets, and best practices. MLPerf Training benchmarks[43] aim to measure the performance of training models while MLPerf Inference benchmarks [46] aim to measure how fast systems can produce results using a trained model. MLCommons takes a neutral stand about any form of comparison of results across submissions. The primary contributions of this work are:

(1) discuss the two scientific machine learning applications of MLPerf HPC benchmark suite viz CosmoFlow, and DeepCAM and explain the benchmarking methodology and process.
(2) throughly investigate the performance trends in submitted results by leading supercomputing platforms in the world
(3) characterize compute, memory, network, and IO behaviours of benchmarking applications.
(4) demonstrate 2.61 × and 1.12 × improvements in performance of Cosmoflow and DeepCAM applications respectively through hyperparameters tuning and gradient skipping techniques over closed division implementations on GPU clusters.
(5) demonstrate 3.38 × improvement in performance of Cosmoflow application leveraging hyperparameters tunings over closed division implementation on CPU clusters.

2 MLPERF HPC BENCHMARK SUITE

The MLPerf HPC benchmarks aim to fill the gaps left by existing benchmark frameworks regarding the ability to characterize performance on large scale systems and remain relevant. These are holistic in the sense that they capture fundamental features of emerging scientific AI workloads, massive data volume, large complex models, and training schemes which incorporate data-parallelism, model-parallelism, and hyper-parameter optimization - making them relevant to industry and academia alike.

The MLPerf HPC benchmarks drive innovation in system and software design alike - impacting not only machine learning workloads, but the entire set of applications which depend heavily on accelerator devices for fast compute, interconnects for high-bandwidth, low-latency communication, or I/O subsystems that govern the rate at which data can be accessed and moved on multi-user systems. Additional requirements of the benchmark suite, such as training to convergence, profile generation, and coarse-grained time reporting enable each individual benchmark's performance to be characterized relative to its utilization of a system's I/O, communication, memory, and compute capabilities. This makes the MLPerf HPC benchmark suite uniquely capable of characterizing the ability of existing HPC systems to run an exciting class of new workloads, while simultaneously providing engineers a sorely needed standard set of benchmarks for informing the design of tomorrow's large-scale high performance computers.

Table 2: MLPerf HPC Benchmarks Overview

Benchmark	Quality Target	#Runs	Tunable Hyperparameters
CosmoFlow	MAE < 0.124	10	batch size, learning rates
DeepCAM	IOU > 0.82	5	optimizer (LAMB or AdamW) batch size, learning rates

The first version of the MLPerf HPC benchmark suite includes two benchmark applications, CosmoFlow and DeepCAM. The reference implementations of these applications are available at [12].

2.1 CosmoFlow

The CosmoFlow benchmark is based on the work by Mathuriya et. al. [42], continued by the ECP ExaLearn project [6]. The task is to predict cosmological parameters from the distribution and structure of dark matter in the universe. The dataset comes from N-body cosmology simulations produced by the ExaLearn team [4] binned into 3D histograms of size 512^3 with four channels representing different red-shift values. These large volumes present considerable challenges for training models due to the large required memory footprint, and so, similar to what is done in [42], the samples are split into smaller cubes of size 128^3 with four red-shift channels. The target quantities are four cosmology parameters, Ω_M, σ_8, n_s, and H_0, which are important to describe the evolution of the universe. The final dataset used for this benchmark has 262,144 samples for training and 65,536 samples for testing and is stored in TFRecord [21] files.

The CosmoFlow reference model was adapted from [25] which introduced some modifications with respect to the original published work. The model is a 3D convolutional neural network with five convolutional layers and three fully-connected layers. Each convolutional layer has kernel size 2 with $32 \times i$ filters in the ith layer. The first two fully connected layers have sizes 128 and 64, respectively. The final layer has output size 4, corresponding to

the predicted target quantities. All hidden layers have leaky ReLU activations, with the exception of the output layer which has a tanh activation scaled by a factor 1.2. After each convolutional layer, there is a 3D Max-Pool operation reducing the sample size by half along each dimension. Finally, the model has dropout layers after the first two fully connected layers with dropout probability 0.5. The model is trained with a mean-squared-error (MSE) loss function and the standard SGD optimizer with an initial learning rate of 0.001 which is dropped to 2.5×10^{-4} at 32 epochs and 1.25×10^{-4} at 64 epochs. The global batch size is set to 64.

The target quality is chosen to be mean-absolute-error (MAE) < 0.124, which allows for good convergence when scaling the batch size and learning rate above the reference configuration. CosmoFlow training exhibited high variability in the number of epochs to converge, which motivated a requirement of 10 training runs to get a reliable measurement of the time to solution.

2.2 DeepCAM

DeepCAM [39] implements a convolutional encoder-decoder segmentation architecture trained on CAM5 climate simulation data with TECA generated heuristic segmentation masks [44] to identify extreme weather phenomena such as atmospheric rivers and tropical cyclones. DeepCAM was the first Deep Learning application which scaled to the full OLCF Summit system and was awarded the ACM Gordon Bell Prize in 2018 [38]. Since then, the model developed into its current form: the ResNet-50 [33] encoder backend was replaced with an Xception [30] network and the input skip-connection was dropped because the network could achieve pixel-level accuracy without it. Furthermore, batch normalization was re-introduced and the original ADAM/LARS optimizer [37, 53] was dropped in favor of the more modern LAMB [54]. The most notable features of this network are 20 residual blocks comprised of depthwise-separable convolutions, which are themselves comprised of grouped convolutions with maximal group count, followed by a point-wise convolution. The bottleneck layer employs atrous spatial pyramid pooling [29] with various filter sizes, as well as global average pooling to extract features at different scales. The results of those operations are concatenated and fed to the deconvolutional decoder. Outside the residual blocks, the network has a single skip connection which propagates low level features directly to the decoder without routing them through the bottleneck layer.

The network takes $16 \times 1152 \times 768$ sized input tensors and predicts 1152×768 sized segmentation masks for three classes (background, tropical cyclone/hurricane, atmospheric river). There are 121,266 training and 15,158 testing samples and no data augmentation is used. DeepCAM is trained with weighted cross-entropy loss, to account for the high class imbalance (about 95% of the pixels are background). The target score is the intersection-over-union (IOU) between the predictions and the targets. The scientifically motivated target score is 0.82, which corresponds to a similarity of 82%.

Unique Characteristics: It is critical to understand what makes these benchmarks different from traditional industrial applications. CosmoFlow is trained on volumetric (3D) data, rather than the 2D data commonly employed in training image classifiers. DeepCAM is trained on images with 768×1152 pixels and 16 channels, which is substantially larger than standard vision datasets like ImageNet, where the average image is 469×387 pixels with at most 3 or 4 channels. Moreover, the massive datasets (5.1 TB for CosmoFlow and 8.8 TB for DeepCAM, compared to 150GB for ImageNet) introduce significant I/O challenges that expose storage and interconnect performance limitations.

3 BENCHMARKING PROCESS

The MLPerf HPC benchmark methodology is closely modeled after the MLPerf Training benchmark, including the general design, metrics, division rules, and submission and review procedures. For instance, the MLPerf HPC benchmarks use the same holistic view of performance and report *time to train* as the primary metric. This choice captures end-to-end performance including both system speed and accuracy. A few changes were made in the rules to improve the relevance of the benchmarks for scientific workloads in the HPC setting. For example, one motivation is the need to capture the impact of data-movement for massive scientific datasets on large HPC parallel file systems and node-local accelerated storage, for which we introduce a rule to include data staging in the measured benchmark time.

3.1 Measurement

Here we describe the finer details and rules relating to measuring *time to train* performance in the benchmarks.

3.1.1 Divisions: MLPerf HPC has two types of submissions, *closed* division and *open* division. In the closed division, the submissions need to be equivalent to the reference implementation. This means that they must have mathematically equivalent model definitions and training algorithms. Such a process enables a direct comparison of the systems. In the open division, submitters are allowed to change the model architectures and training procedure freely but are restricted to evaluate the quality metric in the same way as the reference. This division aims to encourage innovations to further optimize the benchmarks.

3.1.2 Timing rules: Each submission reports the performance metric in *time to solution*. At the start of a run, the benchmark dataset must sit on the parallel file system of the HPC center. On-node caches must be reset, though we do not require system-level caches to be reset due to the difficulties in doing so consistently across systems without severe system disruption. The benchmark timer begins as soon as the dataset is touched, which includes staging into node-local storage. The timer stops when the convergence criteria, as described in the rules, is met.

3.1.3 Run results: ML model training is inherently stochastic due to random initializations, dataset shuffling, etc. Therefore, to get an accurate measurement of the expected time to train, submitters must run the benchmark applications a specified number of times to convergence. In the final scoring, we drop the fastest and slowest results and report the arithmetic mean of the remaining.

3.1.4 Logging: The benchmarks use the `mlperf-logging` library [13], which provides logging utilities and helper functions for all submissions. These help in collecting metadata and evaluating if the submissions meet compliance checks with the set run rules.

3.2 Submission

The submission process is designed to be fair, robust, and reproducible. This is achieved through the enforcement of a required structure for submissions and a peer review process. A submission schedule specifies when benchmarks and rules are finalized, when the submission window opens, the deadline for all submissions, as well as the schedule for the reviews and final deadline for results.

3.2.1 Structure: Submitters must upload their full code used to produce results, as well as system descriptions and the result log files containing the timing information. The submissions must conform to a specified file and directory structure and naming scheme for parsing, summarizing, and peer-review. The required submission structure is described in [7].

3.2.2 Review: After the submission deadline, the peer-review process begins. A set of scripts from the `mlperf-logging` library are first used to check submissions and log files for compliance with the rules. Then, submitters review each other's implementations and results to further verify that they are compliant, sensible, and comprehensible. During the review stage, submitters are also allowed to do "hyperparameter borrowing", in which they may perform additional sets of training runs using the hyperparameter settings of other submissions (but still using their original implementations). This is to allow all submitters to benefit from the hyperparameter tuning performed by everyone, to prevent giving an unfair advantage to submitting teams that can dedicate significantly more resources to hyperparameter tuning.

4 RESULTS

The inaugural MLPerf HPC submission round (v0.7)[†] took place during the summer of 2020. The results from submissions on 7 supercomputers, released in November, showcased the capabilities of state-of-art systems for training large-scale scientific problems. The results are summarized in Table 4 for both closed and open divisions. Benchmark performance is reported with the mean time to solution. Details of the participating HPC systems are listed in Table 3. This section will present a detailed analysis of the submissions, with highlights and observations from each submitter.

4.1 Analysis

The time to solution, broken into data staging, training and evaluation components, is illustrated in Fig. 1 for each submission.

4.1.1 Discussion of data staging time: The data staging time ($T_{staging}$) is shown in Table 5 for the systems where it was measured, along with the ratio to the average epoch time, T_{epoch}. We observe that the staging times are very different for the two benchmarks. By interpolation, ABCI handles staging for CosmoFlow more than 5× faster than for DeepCAM in absolute terms. This difference comes not only from the fact that DeepCAM's data set is 73% larger than CosmoFlow's, but also from the data reduction ratio by compression, which is 88% for CosmoFlow, whereas it is only 23% for DeepCAM.

[†]naming chosen to be consistent with the submission round in MLPerf Training

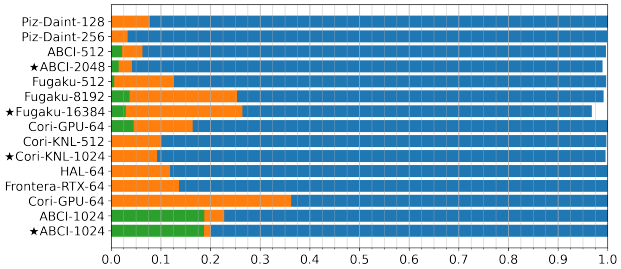


Figure 1: Relative breakdown of time to solution into staging (green), evaluation (orange) and training (blue). Upper 12 submissions: CosmoFlow, lower 3: DeepCAM.
[Lukas: clearly separate DeepCAM from CosmoFlow in figure]

To understand the relative importance of staging ($T_{staging}$) in time to solution ($T_{solution}$), we assume the runtime model

$$T_{solution} = T_{staging} + T_{compute} + T_{extra}. \quad (1)$$

With $T_{compute} = T_{epoch} \cdot \#epochs$ and $T_{extra} \approx 0$ (Fig. 1), we get

$$\frac{T_{staging}}{T_{solution}} \approx \frac{T_{staging}/T_{epoch}}{T_{staging}/T_{epoch} + \#epochs}. \quad (2)$$

The ratio $T_{staging}/T_{epoch}$ (the last column of Table 5) quantifies the *relative* overhead in units of compute epochs that staging adds to time to solution irrespective of convergence[*]. Furthermore, this number also corresponds to the ratio between compute and staging throughput.

Since CosmoFlow has a large number of epochs to converge as we will see later (4× more than for DeepCAM on ABCI-submissions, Table 6), by equation 2 the staging overhead will be marginal compared to the overall runtime (< 5% in all cases in Figure 1). The opposite is true for DeepCAM, where data staging takes 19% of the runtime on the ABCI submissions (only 2.2% & 1.5% for CosmoFlow). This can be explained by both the fewer epochs to converge and the compute-to-staging throughput ratio, which is about 2.5 − 3.5× higher for DeepCAM, which DeepCAM also has higher T_{epoch}.

Fugaku's CosmoFlow staging times are highest at 8,192 processors because of 16-way replication of the data set for 8,192 processors (4-way for 16,384). This is an extra overhead of model-parallelism that could be avoided by optimizing data reading and broadcasting across MPI ranks.

From these observations we conclude that staging adds an relative overhead that strongly depends on the application, in the sense of data compressibility and model convergence and that smaller systems with fewer number of epochs are generally more sensitive to it than larger ones.

4.1.2 Analysis of compute time: This subsection presents an analysis of the compute time ($T_{compute}$ in eq. 1), that is the time spent in training and evaluation, after data staging is completed. It represents ≥ 90% of time to solution in CosmoFlow and ≥ 80% in DeepCAM as shown in Figure 1. We start with a detailed analysis of CosmoFlow and DeepCAM, before comparing the two.

[*]Note that this number depends on the batch size, though, because T_{epoch} does

Table 3: HPC System details

System	#Nodes	Processors (per node)	Accelerators (per node)	Memory (per node)	Node-local Storage (per node)	Interconnect (topology) *to-do: clean this column*
Piz Daint [20]	5,704	1x Intel Xeon E5-2690 v3	1x NVIDIA P100 (16 GB)	64 GB	N/A	Cray Aries [9.7 GB/s internode bi-directional] (Dragonfly)
ABCI [1]	1,088	2x Intel Xeon Gold 6148	4x NVIDIA V100 (16 GB)	384 GB	1600 GB (SSD + NVMe)	InfiniBand EDR (100Gbps) ×2, full-bisection bandwidth in the same rack (34 compute nodes)
Cori-KNL [3]	9,688	1x Intel Xeon Phi 7250	N/A	96 GB DDR4 + 16 GB MCDRAM	N/A	Cray Aries (Dragonfly)
Cori-GPU [2]	18	2x Intel Xeon Gold 6148	8x NVIDIA V100 (16 GB)	384 GB DDR4	930 GB (NVMe)	4 dual-port Mellanox MT27800 (ConnectX-5) EDR InfiniBand network cards
HAL [36]	16	2x IBM POWER 9 model 2.2	4x NVIDIA V100 (16 GB)	256 GB DDR4	N/A	Infiniband EDR
Frontera-RTX [49]	90	2x Intel Xeon E5-2620 v4	4x NVIDIA Quadro RTX 5000 (16 GB)	128 GB DDR4	240 GB (SSD)	FDR (Fat Tree)
Fugaku [19]	158,976	1x Fujitsu A64FX	N/A	32 GB		TofuD [68GB/s x2 (in/out)] (6D Mesh/Torus Network)
ThetaGPU [22] *	24	2x AMD EPYC 7742	8x NVIDIA A100 (40 GB)	1 TB DDR4	15TB SSD, 3.84TB NVMe	20 Mellanox QM9700 HDR200 40-port switches in a fat-tree topology, 25 GB/s bandwidth
Summit [51] *	4,600	2x IBM 3.07 GHz POWER9	6x NVIDIA V100 (16 GB)	512 GB DDR4	1.6TB (NVMe SSD)	dual-rail EDR InfiniBand network

* Measured performance metrics but did not submit for v0.7 submissions

Table 4: Performance metrics (time to solution in minutes) from submissions in closed and open divisions. The average per-run std deviation is 1.7% for DeepCAM, 2.8% for ⋆ABCI-2048/⋆Fugaku-16384 and 11.1% for all other CosmoFlow submissions.

Division	System	Submission	Software	#Processors	#Accelerators	CosmoFlow	DeepCAM
Closed	Piz Daint	Piz-Daint-128	TensorFlow 2.2.0	128	128	461.01	-
	Piz Daint	Piz-Daint-256	TensorFlow 2.2.0	256	256	327.01	-
	ABCI	ABCI-1024	PyTorch 1.6.0	512	1,024	-	11.71
	ABCI	ABCI-512	TensorFlow 2.2.0	256	512	34.42	-
	Fugaku	Fugaku-512	TensorFlow 2.2.0 + Mesh TensorFlow	512	0	268.77	-
	Fugaku	Fugaku-8192	TensorFlow 2.2.0 + Mesh TensorFlow	8,192	0	101.49	-
	Cori-GPU	Cori-GPU-64	PyTorch 1.6.0	16	64	-	139.29
	Cori-GPU	Cori-GPU-64	TensorFlow 1.15.0	16	64	364.73	-
	Cori-KNL	Cori-KNL-512	TensorFlow 1.15.2	512	0	536.06	-
	HAL	HAL-64	TensorFlow 1.15.0	32	64	265.59	-
	Frontera-RTX	Frontera-RTX-64	TensorFlow 1.15.2	32	64	602.23	-
Open	ABCI	⋆ABCI-1024	PyTorch 1.6.0	512	1,024	-	10.49
	ABCI	⋆ABCI-2048	TensorFlow 2.2.0	1,024	2,048	13.21	-
	Fugaku	⋆Fugaku-16384	TensorFlow 2.2.0 + Mesh TensorFlow	16,384	0	30.07	-
	Cori-KNL	⋆Cori-KNL-1024	TensorFlow 1.15.2	1,024	0	419.69	-

Table 5: Data staging time

Benchmark	Submission	Staging time (minutes)	Compute-to-staging throughput ratio
CosmoFlow	Cori-GPU-64	16.49 ± 0.61	2.55
	ABCI-512	0.76 ± 0.004	2.27
	⋆ABCI-2048	0.20 ± 0.004	1.56
	Fugaku-512	1.55 ± 0.11	0.64
	Fugaku-8192	3.77 ± 0.51	3.59
	⋆Fugaku-16384	0.88 ± 0.08	4.93
DeepCAM	ABCI-1024	2.20 ± 0.01	5.55
	⋆ABCI-1024	1.96 ± 0.08	5.45

Compute analysis for CosmoFlow: Using $T_{compute} = T_{epoch} \cdot$ #epochs, we observe that the number of epochs to convergence is primarily an algorithmic property of SGD, and as such, only dependent on the optimizer and choice of hyperparameters (the data and model are held fixed), but not the particular system or implementation (up to floating point precision). On the other hand, T_{epoch} is a system-specific property that depends on both the hardware and the specific parallel implementation of the model as well as the choice of optimizer, but not necessarily on all hyperparameters. In fact, from the rule set in Table 2, T_{epoch} only depends on the value of the *batch size* hyperparameter.

Therefore, we provide analysis of the epoch throughput T_{epoch}^{-1} and # epochs to solution separately for each of the submissions by means of Figure 2. We focus on the number and type of compute unit (accelerators for GPU-based systems, processors for CPU-based systems) to characterize the system and the batch size as most important parameters. The choice of these parameters for each of the submissions is shown in Figure 2 a). The ratio of batch size to # compute units is proportionate to the average amount of computation per compute unit that occurs in every step of distributed SGD. Since the synchronization of weights shared by different compute units is required before the next training step can be processed, we use the inverse of this ratio to quantify the communication intensity of a submission (constant along diagonal level lines). The communication intensity does not directly determine the parallel implementation for a given system, but all submissions in round v0.7 with a batch size to # compute units ratio ≥ 1 chose a purely data-parallel implementation, whereas those with < 1 used a hybrid form of model- and data-parallelism that will be discussed in subsection .

Epoch scaling in CosmoFlow: In Figure 2 (b), we show the scaling of epochs required to converge as a function of the batch size

(*dependent variable on the x-axis!*). As discussed above, this is a *system-independent* property up to floating point precision. The level lines along the diagonal identify points of an identical number of training steps to the solution, which puts a limit on data-parallel scalability. That is, once an increase in the batch size leads to a larger number of training steps to solution, a system can no longer train a model faster by growing the compute resources proportionally (ignoring caching effects). The submissions ⋆ABCI-2048 and ⋆Fugaku-16384 are close to this limit and the specialized techniques to converge efficiently at these very large batch sizes will be discussed in greater detail in subsection 4.2.1. The remaining submissions all closely follow the reference implementation. These turn out to scale efficiently to a batch size of 256 with only 1.3× increase in # epochs to converge compared to batch size 64, but past this point becomes significantly harder to train and less stable (1.6× mean & 7× standard deviation in # epochs for doubling the batch size). Hence these submissions accumulated at batch size 512 as the largest for closed division. The submission ⋆Cori-KNL-1024 discussed in subsection 4.2.3, managed to overcome the convergence issues at batch size 1,024 by slightly modifying the learning rate schedule of the closed division.

Throughput scaling in CosmoFlow: In Figure 2 (c), throughput is shown as a function of the number of compute units *and* the batch size, which jointly determine the communication intensity of training. For each submission, we plot sample throughput (left axis) for training (▲) and evaluation (×) as well as the combined average sample throughput (distribution, the curved lines rooted at the mean only illustrate the averaging by holding the point cloud together) according to the split of the data set (80% training and 20% evaluation) at the abscissa corresponding to the # compute units (bottom). This is illustrated on the submission ABCI-512. On the diagonal we find lines of constant per-accelerator throughput, commonly used to analyze scaling efficiency. Dividing the combined training and evaluation throughput by the overall number of samples in the data set, we obtain the epoch throughput, T_{epoch}^{-1}, which can be read off for the distribution from the right scale. To specify the algorithmically relevant throughput, we plot the training batch size at which a particular epoch throughput has been achieved (⋄, batch size scale on top in fixed 1:1 ratio to # compute units). Where batch size and # compute units disagree, we draw a dashed line parallel to the x-axis between the ⋄ and the distribution on the ordinate of the epoch throughput (the direction and length of this line showing the communication intensity). The scheme is exemplified on ⋆Fugaku-16384, which used model-parallelism. The resulting plot allows us to understand the scaling efficiency of training, evaluation, and combined average (total) throughput for submissions that relate through weak or strong scaling (or extrapolation thereof) and corresponding insights will be discussed in subsection 4.2.

Interestingly, for GPU-based systems, there is a transition occurring from smaller systems to those with 256 GPUs and more, where the gap between training and evaluation throughput becomes very high. Evaluation is inherently bound by data access speed in CosmoFlow and it turns out that the memory configurations of these systems is exactly such that the larger ones - Piz-Daint-256, ABCI-512 and ⋆ABCI-2048 - benefit from caching the data set in RAM. As a result, these systems spend very little (< 5%) of their time in evaluation (Figure 1).

Compute time from throughput- and epoch-scaling in CosmoFlow: Figure 2 (d) shows the outcome of the competition between epoch- and throughput-scaling when growing the system. To obtain it, we *join* the epoch- (Fig. 2 b) and throughput-scaling (Fig. 2 c) of each submission *on the batch size* along the shared axes. The compute time $T_{compute} = T_{epoch} \cdot$ #epochs is constant along diagonal level lines indicated. We find e.g. that while Fugaku-512 has a higher sample throughput than HAL-64, the smaller batch size of HAL-64 causes it to still converge slightly faster in time overall. A similar observation can be made when comparing e.g. ⋆Cori-KNL-1024 to Piz-Daint-256. As a further insight, we are able to trace back the run-to-run variation in time to solution (Table 4) to the number of epochs rather than system throughput. Finally, the technique of graphically *joining throughput- and epoch-scaling on the batch size* allows the inexpensive prediction of compute time to convergence at system configurations other than the ones used in the submissions. This can be done either by (a) additional throughput measurements in Figure 2 c) at batch size with known epochs to convergence or (b) approximately by data-parallel extrapolation of throughput along diagonal lines of constant throughput per accelerator (weak scaling since batch size to # compute unit ratio remains constant) and joining that value with the known epoch scaling from Figure 2 b) on the batch size.This concludes our discussion of CosmoFlow's compute time for now with further details being presented in Section 4.2.

Compute analysis for DeepCAM:. We observe from Figure 1 that more than a third (36.3%) of time to solution is spent in the evaluation phase on Cori-GPU whereas it is comparably negligible on ABCI (4% for closed, 1.3% for open). The reason for this is not primarily due to the different evaluation throughput as seen in Table 6), but the fact that evaluation is triggered after a fixed number of training steps instead of once per epoch. As a consequence, Cori-GPU calculates the IOU score 8× as often as ABCI's closed division submission and 36× as often as ABCI's open submission, where the evaluation frequency has been further decreased. As DeepCAM uses training steps to define its evaluation rhythm, a compute analysis similar to CosmoFlow's, but based on training steps instead of epochs to solution would be possible. However, we omit this here due to the lack of space and instead discuss the results for DeepCAM in comparison with CosmoFlow in the following.

4.1.3 Benchmark differences: In Table 6, we present the compute characteristics of both CosmoFlow and DeepCAM on systems with submissions to both benchmarks. We see that DeepCAM requires fewer epochs to converge than CosmoFlow (20-25%), which as a function of the batch size grows at a rate that is roughly the same as in CosmoFlow, but with much more stable convergence. For sample throughput, we find that (1) CosmoFlow has higher throughput in both training ($3-5\times$) and evaluation ($7-20\times$), which can be attributed to the lower number of layers, whereas (2) DeepCAM has a much smaller gap between training and evaluation throughput (1.4× vs. 5.7×) Comparing the resource footprint, we find that to reach convergence, the compute budget for CosmoFlow is $1.5-2.6\times$ larger than for DeepCAM, with correspondingly higher

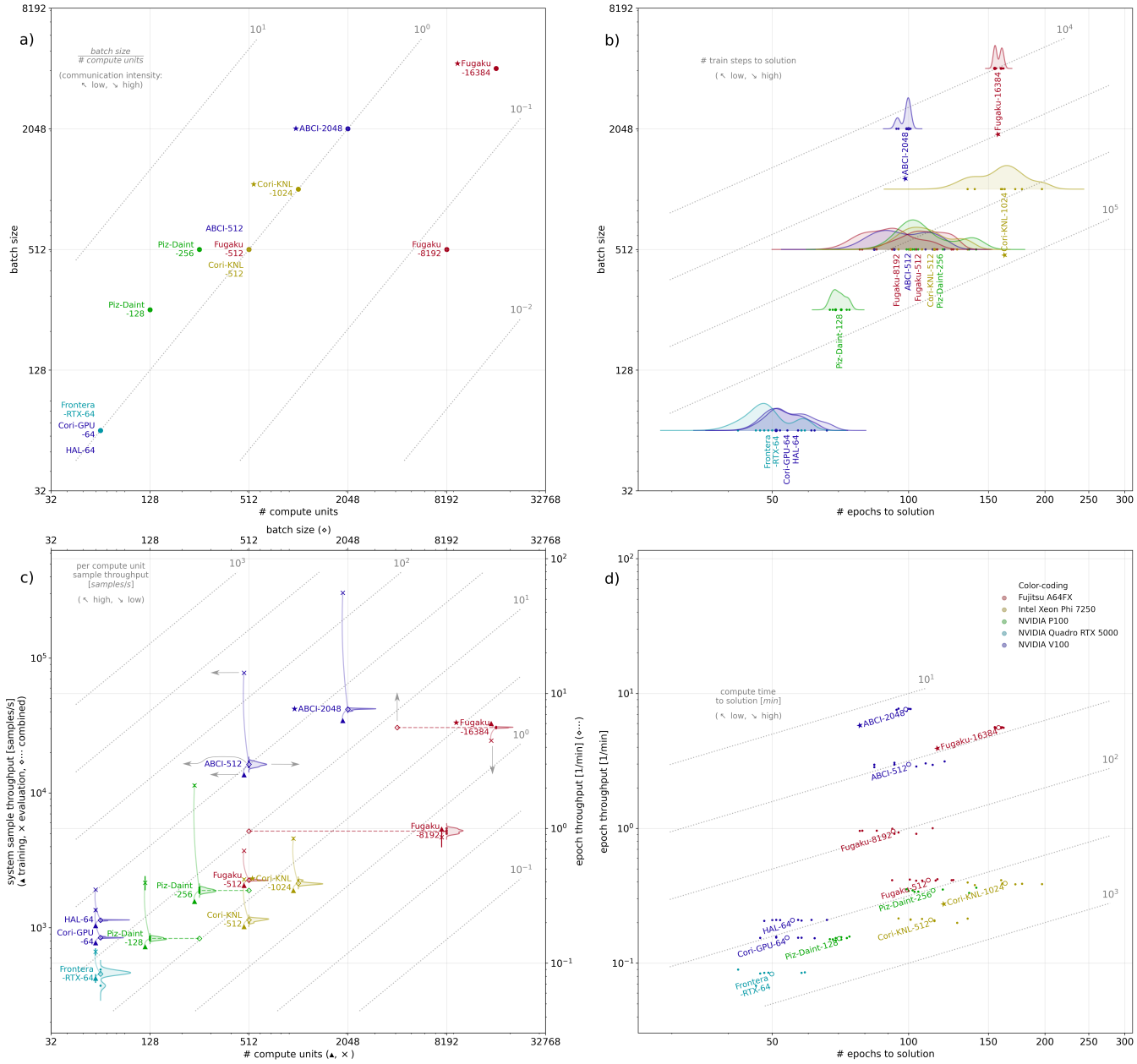


Figure 2: Compute analysis of CosmoFlow with figures on a) parameter choice, b) epoch scaling (*dependent variable on x-axis*), c) throughput scaling (arrows in ⋆Fugaku-16384 explain how to read x-axis, in ABCI-512 how to read y-axis), and d) compute time. Note that axes are *shared* along rows (y-axis in a & b: batch size, in c & d: epoch throughput) and columns (x-axis in a & c: # compute units, in b & d: # epochs to solution).

time to solution that is 2.6 − 2.9× that of DeepCAM for closed division. Notably, though, we see for both benchmarks that despite a relatively large increase in number of # accelerators by 8 − 16×, the compute budget with the right optimizations can be constrained and a decrease in ∼ 10× time to solution is reliably possible.

4.2 Highlights

The MLPerf HPC submission round has an entry with the largest scale of runs so far in any MLPerf benchmarking submissions. The entry on the Fugaku system has implemented CosmoFlow in a model parallel fashion on a massive scale with 16,384 processors. We

Table 6: Scaling of required epochs, throughput, time to solution and compute budget in CosmoFlow vs. DeepCAM.

Benchmark	Submission	Batch size	# Epochs	# Accelerators	Training throughput/# acc. (samples/second)	Evaluation throughput/# acc. (samples/second)	Time to solution (minutes)	Compute budget (h · acc)
CosmoFlow	Cori-GPU-64	64	53.88 ± 4.85	64	12.07 ± 0.09	21.17 ± 0.56	364.73 ± 32.77	389.04
	ABCI-512	512	100.00 ± 13.50	512	26.59 ± 0.90	151.88 ± 0.68	34.42 ± 4.03	293.03
	★ABCI-2048	2,048	98.50 ± 2.56	2,048	16.79 ± 0.23	149.21 ± 0.28	13.21 ± 0.35	450.96
DeepCAM	Cori-GPU-64	128	10.00 ± 0.00	64	3.56 ± 0.13	3.55 ± 0.18	139.29 ± 3.63	148.58
	ABCI-1024	2,048	24.00 ± 0.00	1,024	5.24 ± 0.02	7.37 ± 0.01	11.71 ± 0.02	199.78
	★ABCI-1024	2,048	23.67 ± 1.16	1,024	5.57 ± 0.13	4.67 ± 0.82	10.49 ± 0.23	178.95

now present additional details of the implementations and present highlights from several systems.

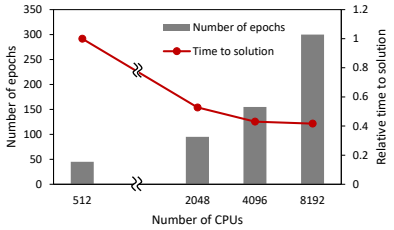


Figure 3: Plot showing the time to solution and the number of epochs using data parallelism with increasing CPU count on Fugaku, using a local batch size one (for open division).

4.2.1 Fugaku: Submissions on the Fugaku supercomputer utilized hybrid data and model parallel execution. These parallelism schema are implemented by a technique described in the section below. The local batch size per CPU is set to one, which means that the number of CPUs and the number of nodes are the same. Data reformatting by compressing and archiving multiple files was effective to reduce data staging time. Training data is staged in RAM disks in advance. Only in the case of the 512-node submission, RAM disks do not have enough capacity, so data staging is performed on local SSDs on I/O nodes that are assigned to each unit of 16 nodes. Also, the data cache function of TensorFlow (`tf.data.Dataset.cache`) is used to improve the bandwidth of data loading during training.

For the open division, the following accuracy improvement techniques were applied: (1) use linear learning rate decay scheduler, (2) apply data augmentation, (3) disable dropout layers. Training time scaled up to batch size 4,096 with local batch size one for the open division on Fugaku after the hyperparameter tuning. Figure 3 shows the scaling of time to solution and the number of epochs for the open division. The scaling of time to solution is limited to 4,096, because the number of epochs to convergence increases as the global batch size increase.

Since the accuracy could not reach the target using the batch size larger than 4096 even after the hyperparameter tuning, model parallelism is necessary to scale beyond 4,096 processors. Therefore, a hybrid approach utilizing data and model-parallelism is implemented for CosmoFlow on Fugaku. Model parallelism in TensorFlow is implemented by extending Mesh TensorFlow so that multi processes of both data and model parallelism are enabled. Model parallelism is applied in Conv3d layers by spatially partitioning input tensors in two dimensions. Figure 4 illustrates an example of the spatial partitioning for two processes. The hybrid parallelism enables scaling the number of CPUs up to 8,192 with 4x4 spatial partitioning for the closed division and 16,384 with 4x1 spatial partitioning for the open division (2.62× and 1.98× speedup of training throughput by model-parallelism compared to Fugaku-512/an extrapolation thereof to batch size 4,096). There is still room for improving the scaling efficiency further by reducing the communication overhead.

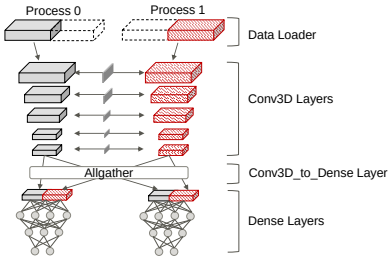


Figure 4: Spatial partitioning for CosmoFlow.

4.2.2 ABCI:. For both of the benchmarks, data reformatting by compressing and archiving multiple files was effective to reduce data staging time. Data shuffling was applied for intra-node GPUs after each epoch, since the dataset is too large to fit on local storage and a partial dataset is shared only intra-node after data staging.

For CosmoFlow, the following performance optimizations were applied to improve training and evaluation throughput: (1) improve data loader bandwidth using NVIDIA Data Loading Library (DALI), (2) apply mixed-precision training, (3) increase validation batch size. Training time scales up to batch size 512 with local batch size one for the closed division. For the open division, after the same hyperparameter tuning techniques mentioned in section 4.2.1 were applied, batch size 2,048 with local batch size one was optimal. Using a larger batch size than 2,048 did not achieve additional speedup because network bandwidth degraded due to congestion and the number of epochs increases with an increase in the batch size.

For DeepCAM, page-locked memory (a.k.a pinned memory) is used to improve memory bandwidth, and four additional worker processes were forked for data loading to improve I/O bandwidth. The distributed data shuffling were applied for intra-node GPUs, and hyper-parameters were tuned, including the warmup steps to reduce the number of epochs to convergence. Training time scales up to batch size 2,048 with local batch size two for closed and open divisions on ABCI after hyperparameter tuning. Especially tuning the warmup steps was effective to reduce the number of

epochs to convergence. For the open division submission on ABCI, the Gradient Skipping (GradSkip) technique, one of the Content-Aware Computing (CAC) techniques developed by Fujitsu, was also applied. GradSkip avoids updating weights in some layers in the training process, by finding layers which have little effect on accuracy, based on automatic analysis of the content of data during the training process.

4.2.3 Cori/Cori-GPU:. Submissions on the Cori supercomputer at NERSC utilized both the primary KNL partition as well as the Cori-GPU testbed. System details are available at [2, 3].

CosmoFlow was trained on Cori KNL on 512 nodes in the closed division and 1024 nodes in the open division. For the open division submission, an additional learning rate decay was added in order to enable convergence at global batch size 1024. The implementation used Intel-optimized TensorFlow with MKL-DNN for optimized performance on the Intel processors. Runtime settings for inter- and intra-parallelism threads, OpenMP threads, and affinity were tuned for maximal throughput. Shifter containers were used to launch training, which prevented scalability issues in shared library loading from the parallel file system at scale. These results show that large CPU systems like Cori can still be useful for training computationally-expensive deep learning models. CosmoFlow was additionally trained on the Cori GPU system on 8 nodes (64 V100 GPUs). Horovod with NCCL-based allreduce was used to achieve efficient data-parallel training. Additionally, node-local SSDs were used to store local partitions of the full dataset. The staging time from the Cori scratch filesystem to the node-local SSDs was considerably longer than other submissions with data-staging, indicating there is further room for optimization in future MLPerf HPC submission rounds.

DeepCAM was similarly trained on the Cori GPU system utilizing 64 V100 GPUs. The implementation in PyTorch utilized the NVIDIA Apex library for automatic mixed precision and used NCCL for optimized distributed data-parallel training.

4.2.4 Piz Daint: On Piz Daint [20], we focused on two data-parallel submissions in the closed division of CosmoFlow with 128 and 256 GPUs, one GPU per node. By using Sarus [27], a container engine with near-native performance for Docker-compatible containers, we were able to rapidly test and tune distributed training with Horovod and NCCL for fine-grained communication to obtain near optimal weak scaling in the range of 100-1000 nodes as shown in Figure 5. A low cycle time, tensor fusion threshold and the usage of the hierarchical, tree-based allreduce implementation proved to be key to achieve this performance.

To find the optimal batch size at a fixed node count, throughput scaling due to increased (local) GPU data-parallelism (Figure 5) has to be traded off against epoch scaling (Fig. 2 b). In particular, we find from Figure 5 that the throughput ratio for a local batch size 4, 2, and 1 relative to a maximum local batch size of 8 that fits on the P100's memory roughly coincides with the strong scaling efficiency, which at a batch size of 1,024 is 91%, 64%, and 35%. With the epoch scaling as demonstrated in Figure 2 b), and a measured 59 epochs to converge for batch size 128, this causes a local batch size of 2 to give the fastest time to convergence for both of our configurations. Interestingly, this is even the case on 256 nodes, where the +70% throughput increase due to higher GPU-parallelism

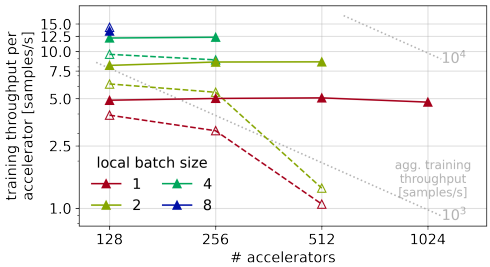


Figure 5: Weak-scaling of training throughput on Piz Daint before (hollow symbols, dashed lines) and after (filled symbols, solid lines) Hovorod/NCCL-optimizations in the batch size region 128-1024.

Table 7: Memory bandwidth measurements

Benchmark	System	Tool	Memory Bandwidth (GB/sec)
CosmoFlow	ABCI	Nvprof	335.4
CosmoFlow	Fugaku	Perf	95.0
CosmoFlow	Summit	Nsight	233.1
CosmoFlow	ThetaGPU	Nsight	194.5
DeepCAM	ABCI	Nvprof	153.1
DeepCAM	Summit	Nsight	254.7

from local batch size 1 to 2 outweighs the +62% increase in epochs to convergence when moving from batch size 256 to 512, leaving a narrow 5% improvement in the time to solution. On the other hand, increasing the local batch size on 128 nodes from 2 to 4 is expected to increase the time to solution by 13%. To further reduce time to solution by strong scaling, we expect an alternative approach to data-parallelism (which would only give 11% improvement for scaling out by another factor of 2) to be more efficient.

Curiously, in Figure 2 c), we find that throughput scales faster than ideal from 128 to 256 GPUs - +8% compared to what is expected for training and a full +167% for evaluation, respectively. As a result, the time to solution is 12% faster on 256 than expected based on the epoch scaling. This scaling is a result of being able to cache the data set in RAM with 256 nodes, whereas with 128 nodes, parallel file system I/O becomes a bottleneck, which more directly impacts evaluation and could be alleviated using near-compute storage.

In summary, we have identified fine-grained communication together with the addition of near-compute storage as key optimizations for CosmoFlow on Piz Daint.

5 WORKLOAD CHARACTERIZATION

In this section, we present the workload characterization of these benchmark applications on various HPC systems. We measure performance metrics to help understand their memory, network and I/O behaviours. It is to be noted that these metrics are captured in additional runs separate from v0.7 submissions. For CosmoFlow, we used 65,536 training samples and 16,384 validation samples for 2 epochs with local batch size of 1. For DeepCam, we used all data samples (121,266 for training and 15,158 for validation validation samples) for 2 epochs with local batch size of 2.

Table 8: Network bandwidth (BW) measurements

Benchmark	System	Tool	BW (GB/sec)	size (MB)
CosmoFlow (512 GPUs)	ABCI	Horovod Timeline	3.41	19.97
CosmoFlow (512 CPUs) *	Fugaku	Mpitrace	0.75	21.71
CosmoFlow (256 GPUs)	Piz Daint	Horovod Timeline	1.86	2.21
CosmoFlow (510 GPUs)	Summit	Horovod Timeline	2.24	22.0
CosmoFlow (128 GPUs)	ThetaGPU	Horovod Timeline	1.95	15.20
DeepCAM (512 GPUs)	ABCI	Timer-based	3.73	37.77
DeepCAM (510 GPUs)	Summit	Timer-based	4.50	225.0

* without model parallelism

Table 9: I/O bandwidth measurements

Benchmark	System	Tool	I/O Bandwidth (GB/sec)
CosmoFlow	ABCI	Nvprof	1.65
CosmoFlow	Fugaku	Timer-based	2.57
CosmoFlow	Summit	Darshan	1.46
CosmoFlow	ThetaGPU	Darshan	1.98
CosmoFlow	Piz Daint	Darshan	8.08
DeepCAM	ABCI	Darshan	2.36

5.1 Memory Bandwidth

We measure memory traffic of these benchmark implementations to estimate how much bandwidth is used for memory reads and writes to the off-chip DRAM on respective systems. More concisely, we measure the accelerator memory bandwidth (except Fugaku?) aggregated across the system. Global memory bandwidth is usually influenced by the underlying cache implementations and may not reflect the memory traffic in its entirety. Hence, we measure DRAM read and write throughput. Table 7 lists the average bi-directional bandwidth (read and write) across different systems.

On ABCI, we used Nvidia `to` calculate the average memory bandwidth of all kernels based on the elapsed time for each CUDA kernel and the memory bandwidth between L2 cache and HBM memory. Since Fugaku does not have GPUs, we used `Perf` [18] to extract read and write memory bandwidths measured at 1ms intervals and the average bandwidths are calculated for each. While `nvprof` measures the bandwidth of CUDA kernel time only, `perf` measures the bandwidth of the training interval at regular intervals. On ThetaGPU and Summit we used Nvidia `Nsight compute` [17] to extract the memory bandwidth of all kernels using the metric `dram__bytes.sum.per_second`.

Observations: From Table 7, it can be observed that the measured memory bandwidth is, in general, higher on GPU-based systems than CPU-based systems owing to better caching mechanisms on the former. We observe the memory bandwidths on Fugaku appear to be smaller than those of ABCI by at least 1.6X times. We expect this is because the CPU system can make use of the L1/L2 cache on CPU more efficiently. In fact, the A64FX CPU has a 32MB L2 cache, which is larger than a 6.1MB L2 cache on the V100 GPU.

5.2 Network Bandwidth

Distributed implementations of deep learning applications typically spend significant time in collective communication calls. Optimizing worker node communication is critical for high-throughput. To understand this behavior, we profiled the heavy collective communication calls, AllReduce operations across implementations. This included calls to `MPI_AllReduce` or `NCCL_AllReduce` based on the communication substrate used. Figure 6(a-b) presents the communication patterns (percentage communication time) of CosmoFlow while Figure 6-(c) shows similar measurements for DeepCAM.

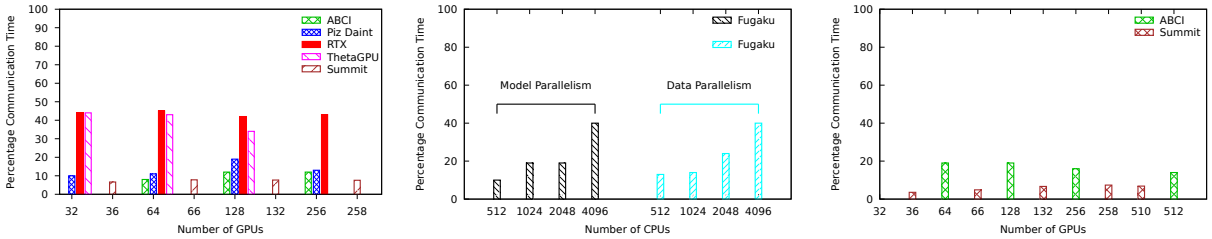
Since CosmoFlow's reference implementation uses Horovod, we used `Horovod Timeline` [8] to obtain the average network bandwidth for collective communications as `mpitrace` [15] failed for us to correctly capture overlapping communication and computation. The average network bandwidth is calculated based on the NCCL time obtained from the Horovod timeline, excluding the waiting time for data fusions. On the other hand, DeepCAM does not use MPI communication through Horovod in the reference implementation and instead uses NCCL communication through `NVIDIA Apex` [16]. Therefore, synchronization operations and timers are inserted before and after the collective communications of NVIDIA Apex to measure the communication time. Then we calculate the average communication bandwidth from the amount of actual data transferred. On Fugaku, as we used Mesh-TensorFlow for CosmoFlow, Horovod timeline cannot be used for the performance measurement and we use mpitrace and `mpiP` [14] together to calculate the average communication bandwidth.

Observations: The percentage of the application training time spent in collective communications with either MPI or NCCL to the total application time is roughly 10-40% as we scale across GPUs on the evaluated systems. On Fugaku, for the data parallel execution, the computation time scales with the number of CPUs, while the MPI communication time does not. Besides, for the model parallel execution, the scalability of the computation time is lower than that of the data parallel execution. This is because model parallelism is applied only to the `conv3d` layer and not to the fully-connected layer. Also, the communication time increases as the degree of model parallelism is increased. This is due to the communication overhead caused by the data transfer in the halo region when performing spatial partitioning in the `conv3d` layer.

5.3 I/O Bandwidth

As the MLPerf HPC benchmarks have massive input data sets, it is important to understand the I/O performance. Table 9 shows the average I/O bandwidth per worker on different systems. We used `Darshan` [28] to get the average I/O bandwidth that captures all I/O-related activity, such as types and number of files and aggregate performance combined with shared and unique files worked by all ranks on certain systems. On ABCI, Darshan cannot measure the I/O volume accurately since our implementation of CosmoFlow used NVIDIA DALI which partly performs mmap-based I/O. Hence, we used `Nvprof` to measure the time of the kernel (`TFRecordReader`) that is performing I/O to calculate the average I/O bandwidth. On Fugaku, we insert timers before and after the data loads, and calculated from the elapsed time and the amount of data.

Observations: It is challenging to accurately capture the time spent in I/O due to the overlap in computation with the I/O activity. However, from Table 9, it can be observed that the measured I/O bandwidth is similar among the systems, and we can expect that I/O bandwidth is high enough to hide I/O behind computation. For example, for DeepCAM on ABCI, I/O bandwidth is 2.36 GB/s per process, using 256 processes with a full training dataset consisting of 7.7TB. In this case, estimated I/O time per epoch is 7700[GB] /



(a) CosmoFlow : GPU (b) CosmoFlow : CPU (c) DeepCAM : GPU

Figure 6: Communication patterns in CosmoFlow and DeepCAM applications.

256 / 2.36[GB/s] = 12.8 seconds, while measured average run time per epoch that includes the I/O time is 99.6 seconds.

6 KEY INSIGHTS

While MLPerf HPC benchmarks have and will continue to provide insights for individual systems on performance, additional insights can be drawn from the v0.7 results and experience that inform the wider HPC community on challenges and best practices in supporting scientific ML workloads.

- Data movement and read performance are important pieces of the overall workflow performance on massive datasets, especially for systems with accelerators. Accelerated storage solutions like on-node SSDs are critical, but one also has to optimize the data-staging part of the pipeline as well.
- Model convergence at scale with large batch sizes is still a challenge, limiting how much we can accelerate ML training. Submitting teams struggled to push CosmoFlow beyond a batch size of 512 in the closed division rules. Additional studies are needed to develop best practices for scaling ML training on scientific datasets. This challenge also reinforces the need for hybrid parallel training methods, e.g. in order to scale CosmoFlow training beyond 512 nodes on Fugaku.
- Model specific tuning is effective to scale training. For CosmoFlow, batch size is increased from 512 to 4,096 by optimizing learning rate scheduler, applying data augmentation, and disabling dropout layers. For DeepCAM, batch size is increased up to 2,048 by optimizing warmup steps and multi step learning scheduler.
- Performance tuning for throughput is also beneficial. For data staging throughput, data reformatting by compressing and archiving multiple files is effective. For training and evaluation throughput, improve data loader bandwidth using NVIDIA DALI, applying mixed-precision, and increasing validation batch size, applying data shuffling only for intra-node GPUs are found effective.
- Efficiently using HPC networks is critical to get good performance on supercomputers - existing frameworks may need specific tuning for fine-grained communication to effectively overlap communication and computation
- *to-do: add run-to-run variability*

7 CONCLUSION

In this paper we presented MLPerf HPC, a benchmark suite aimed at representative scientific machine learning applications. The initial release included two benchmark applications, CosmoFlow and DeepCAM with terabytes of input data sets. We presented the results and analysis of initial submissions from leadership supercomputers globally. Later, we discussed the application characteristics in terms of memory, I/O bandwidth, and network bandwidth. In the next release of the benchmark suite, we aim to address the limitations of the existing benchmarks and add applications from state-of-art models, such as transformers from science domains.

REFERENCES

[1] [n.d.]. AI Bridging Cloud Infrastructure (ABCI). https://abci.ai/en/about_abci/.
[2] [n.d.]. Cori GPU Nodes. https://docs-dev.nersc.gov/cgpu/.
[3] [n.d.]. Cori System. https://docs.nersc.gov/systems/cori/.
[4] [n.d.]. CosmoFlow Datasets. https://portal.nersc.gov/project/m3363/.
[5] [n.d.]. DeepBench. https://github.com/baidu-research/DeepBench.
[6] [n.d.]. ExaLearn Project. https://petreldata.net/exalearn/.
[7] [n.d.]. General MLPerf Submission Rules. https://github.com/mlcommons/policies/blob/master/submission_rules.adoc.
[8] [n.d.]. Horovod Timeline. https://horovod.readthedocs.io/en/stable/timeline_include.html.
[9] [n.d.]. HPE Deep Learning Benchmarking Suite. https://github.com/HewlettPackard/dlcookbook-dlbs/.
[10] [n.d.]. HPL-AI Mixed-Precision Benchmark. https://icl.bitbucket.io/hpl-ai/.
[11] [n.d.]. MLCommons. https://mlcommons.org/en/.
[12] [n.d.]. MLPerf HPC Benchmark Suite. https://github.com/mlcommons/hpc.
[13] [n.d.]. MLPerf Logging Library. https://github.com/mlcommons/logging.
[14] [n.d.]. mpiP profiling tool. https://github.com/LLNL/mpiP.
[15] [n.d.]. MPItrace tool. https://github.com/IBM/mpitrace.
[16] [n.d.]. Nvidia Apex Extension. https://github.com/NVIDIA/apex.
[17] [n.d.]. Nvidia Nsight Compute Profiler. https://developer.nvidia.com/nsight-compute.
[18] [n.d.]. Perf profiling tool. https://perf.wiki.kernel.org/index.php/Main_Page.
[19] [n.d.]. The Supercomputer Fugaku. https://www.r-ccs.riken.jp/en/fugaku/project/outline.
[20] [n.d.]. The Supercomputer Piz Daint. https://www.cscs.ch/computers/piz-daint/.
[21] [n.d.]. TFRecord. https://www.tensorflow.org/tutorials/load_data/tfrecord.
[22] [n.d.]. ThetaGPU. https://www.alcf.anl.gov/alcf-resources/theta.
[23] [n.d.]. Top500 List: November 2020. https://www.top500.org/lists/2020/11/.
[24] R. Adolf, S. Rama, B. Reagen, G. Wei, and D. Brooks. 2016. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581275
[25] Jan Balewski. [n.d.]. CosmoFlow. https://bitbucket.org/balewski/cosmoflow.
[26] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler. 2019. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 66–77. https://doi.org/10.1109/IPDPS.2019.00018
[27] Lucas Benedicic, Felipe A Cruz, Alberto Madonna, and Kean Mariotti. 2019. Sarus: Highly Scalable Docker Containers for HPC Systems. In *International Conference on High Performance Computing*. Springer, 46–60.
[28] Philip H. Carns, Robert Latham, Robert B. Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 Characterization of petascale I/O workloads.. In *CLUSTER*. IEEE Computer Society, 1–10. http://dblp.uni-trier.de/db/conf/cluster/cluster2009.html#CarnsLRILR09
[29] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. 2017. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. arXiv:1606.00915 [cs.CV]
[30] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. arXiv:1610.02357 [cs.CV]
[31] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *SIGOPS Oper. Syst. Rev.* 53, 1 (July 2019), 14–25. https://doi.org/10.1145/3352020.3352024
[32] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. [n.d.]. Dawnbench: An end-to-end deep learning benchmark and competition. ([n. d.]).
[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]
[34] Tony Hey, Keith Butler, Sam Jackson, and Jeyarajan Thiyagalingam. 2020. Machine learning and big scientific data. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378 (03 2020), 20190054. https://doi.org/10.1098/rsta.2019.0054
[35] Zihan Jiang, Lei Wang, Xingwang Xiong, Wanling Gao, Chunjie Luo, Fei Tang, Chuanxin Lan, Hongxiao Li, and Jianfeng Zhan. 2020. HPC AI500: The Methodology, Tools, Roofline Performance Models, and Metrics for Benchmarking HPC AI Systems. arXiv:2007.00279 [cs.PF]
[36] Volodymyr Kindratenko, Dawei Mu, Yan Zhan, John Maloney, Sayed Hadi Hashemi, Benjamin Rabe, Ke Xu, Roy Campbell, Jian Peng, and William Gropp. 2020. HAL: Computer System for Scalable Deep Learning. In *Practice and Experience in Advanced Research Computing* (Portland, OR, USA) *(PEARC '20)*. Association for Computing Machinery, New York, NY, USA, 41–48. https://doi.org/10.1145/3311790.3396649
[37] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]
[38] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. 2018. Exascale Deep Learning for Climate Analytics. arXiv:1810.01993 [cs.DC]
[39] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, P. Prabhat, and M. Houston. 2018. Exascale Deep Learning for Climate Analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 649–660. https://doi.org/10.1109/SC.2018.00054
[40] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W. Hwu. 2020. XSP: Across-Stack Profiling and Analysis of Machine Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 326–327. https://doi.org/10.1109/IPDPS47924.2020.00042
[41] S. Mahon, S. Varrette, V. Plugaru, F. Pinel, and P. Bouvry. 2020. Performance Analysis of Distributed and Scalable Deep Learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 760–766. https://doi.org/10.1109/CCGrid49817.2020.00-13
[42] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn J. Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor W. Lee. 2018. CosmoFlow: using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 65:1–65:11. http://dl.acm.org/citation.cfm?id=3291743
[43] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 336–349. https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf
[44] Prabhat, Oliver Rübel, Surendra Byna, Kesheng Wu, Fuyu Li, Michael Wehner, and Wes Bethel. 2012. TECA: A Parallel Toolkit for Extreme Climate Analysis. *Procedia Computer Science* 9 (2012), 866–876. https://doi.org/10.1016/j.procs.2012.04.093 Proceedings of the International Conference on Computational Science, ICCS 2012.
[45] Evan Racah, Christopher Beckham, Tegan Maharaj, Samira Kahou, Mr. Prabhat, and Chris Pal. 2017. ExtremeWeather: A large-scale climate dataset for semi-supervised detection, localization, and understanding of extreme weather events. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 3405–3416. http://papers.nips.cc/paper/6932-extremeweather-a-large-scale-climate-dataset-for-semi-supervised-detection-localization-and-understanding-of-extreme-weather-events.pdf
[46] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. 2020. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 446–459. https://doi.org/10.1109/ISCA45697.2020.00045
[47] Andrew Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. 2020. Improved protein structure prediction using potentials from deep learning. *Nature* 577 (01 2020), 1–5. https://doi.org/10.1038/s41586-019-1923-7
[48] S. Sharma, Chung-Hsing Hsu, and Wu-chun Feng. 2006. Making a case for a Green500 list. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 8 pp.–. https://doi.org/10.1109/IPDPS.2006.1639600
[49] Dan Stanzione, John West, R. Todd Evans, Tommy Minyard, Omar Ghattas, and Dhabaleswar K. Panda. 2020. Frontera: The Evolution of Leadership Computing at the National Science Foundation. In *Practice and Experience in Advanced Research Computing* (Portland, OR, USA) *(PEARC '20)*. Association for Computing Machinery, New York, NY, USA, 106–111. https://doi.org/10.1145/3311790.3396656
[50] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. 2020. AI for Science. (2 2020). https://doi.org/10.2172/1604756
[51] S S Vazhkudai, B R de Supinski, A S Bland, A Geist, J Sexton, J Kahle, C J Zimmer, S Atchley, S H Oral, D E Maxwell, V G Vergara Larrea, A Bertsch, R Goldstone, W Joubert, C Chambreau, D Appelhans, R Blackmore, B Casses, G Chochia, G

Davison, M A Ezell, E Gonsiorowski, L Grinberg, B Hanson, B Hartner, I Karlin, M L Leininger, D Leverman, C Marroquin, A Moody, M Ohmacht, R Pankajakshan, F Pizzano, J H Rogers, B Rosenburg, D Schmidt, M Shankar, F Wang, P Watson, B Walkup, L D Weems, and J Yin. [n.d.]. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. ([n. d.]). https://www.osti.gov/biblio/1489443

[52] Yu Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *CoRR* abs/1907.10701 (2019). arXiv:1907.10701 http://arxiv.org/abs/1907.10701

[53] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large Batch Training of Convolutional Networks. arXiv:1708.03888 [cs.CV]

[54] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. arXiv:1904.00962 [cs.LG]

[55] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 88–100. https://doi.org/10.1109/IISWC.2018.8573476