MLPerf™ HPC: A Holistic Benchmark Suite for Scientific Machine Learning on HPC Systems

Steven Farrell *, Murali Emani †, Jacob Balma ‡, Lukas Drescher §, Alexandr Drozd ¶, Andreas Fink ††, Geoffrey Fox ‡‡, David Kanter **, Thorsten Kurth †‡, Peter Mattson ‡‡‡, Dawei Mu †††, Amit Ruhela †‡‡, Kento Sato §, Koichi Shirahata ‡‡§, Tsuguchika Tabaru ‡‡¶, Aristeidis Tsarlis ‡‡‖, Jan Balewski *, Ben Cumming §, Takumi Danjo ‡‡¶, Jens Domke ‡‡‖, Takaaki Fukai ‡‡¶, Naoto Fukumoto ‡‡§, Tatsuya Fukushi ‡‡¶, Balazs Gerofi ¶, Takumi Honda ‡‡¶, Toshiyuki Imamura ‡‡¶, Akihiko Kasagi ‡‡§, Kentaro Kawakami ‡‡¶, Shuhei Kudo ‡‡¶, Akiyoshi Kuroda ‡‡¶, Maxime Martinasso ‡‡§, Satoshi Matsuoka ‡‡¶, Henrique Mendonça ‡‡§, Kazuki Minami ‡‡¶, Prabhat Ram ‡‡¶, Takashi Sawada ‡‡§, Mallikarjun Shankar ‡‡, Tom St. John ‡‡, Akihiro Tabuchi ‡‡¶, Venkatram Vishwanath †‡, Mohamed Wahib †‡†, Masafumi Yamazaki ‡‡¶, Junqi Yin ‡‡¶

*Lawrence Berkeley National Laboratory, USA † Argonne National Laboratory, USA ‡ Maxwell Labs, USA § Swiss National Supercomputing Centre, Switzerland ¶ RIKEN Center for Computational Science, Japan ‡‡ University of Virginia, USA ** MLCommons, USA ‡‡‡ NVIDIA, Switzerland ‡‡‖ Google, USA †† National Center for Supercomputing Applications, USA ‡‡‡‡ Texas Advanced Computing Center, USA ‡‡§§ Fujitsu Limited, Japan ‡‡¶¶ Oak Ridge National Laboratory, USA ‡‡¶¶¶ Microsoft, USA ††† Cruise, USA †††† National Institute of Advanced Industrial Science and Technology, Japan

Abstract—Scientific communities are increasingly adopting machine learning and deep learning models in their applications to accelerate scientific insights. High performance computing is 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. 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, driven by the MLCommons™ Association. We present the results from the first submission round including a diverse set of some of the world’s largest HPC systems. We develop a systematic framework for their joint analysis and compare them in terms of data staging, algorithmic convergence and compute performance. As a result, we gain a quantitative understanding of optimizations on different subsystems such as staging and on-node loading of data, compute-unit utilization and communication scheduling enabling overall > 10× (end-to-end) performance improvements through system scaling. Notably, our analysis shows a scale-dependent interplay between the dataset size, a system’s memory hierarchy and training convergence that underlines the importance of near-compute storage. To overcome the data-parallel scalability challenge at large batch-sizes, we discuss specific learning techniques and hybrid data-and-model parallelism that are effective on large systems. We conclude by characterizing each benchmark with respect to low-level memory, I/O and network behaviour to parameterize extended roofline performance models in future rounds.

Index Terms—Deep Learning, Benchmarks, Supercomputers, Scientific Applications

I. INTRODUCTION

Scientific applications are leveraging the potential of machine learning (ML) and, especially, deep learning 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 applications with respect to the type and volume of data and the resulting model complexity. Recently, ML-driven applications have led to major insights and scientific discoveries that would have taken years using traditional methods. A recent breakthrough in addressing one such grand challenge in biology was the development of an ML model that solved the protein folding problem with the AlphaFold tool by Alphabet’s Deepmind [1]. Large-scale experiments will yield unprecedented volumes of scientific data. The authors of [2] highlight the challenges faced with understanding the exponential growth of experimental data and present opportunities of using machine learning techniques to advance science. The AI for Science report [3] 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 ML 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. Implementation of ML models on
supercomputers at full scale poses challenges that are not typically exposed at small-scale, such as the I/O impact of large-scale datasets. 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 I. TOP500 [4] 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. Green500 [5] ranks supercomputers based on their energy efficiency. The list reports the performance per rated watt using the LINPACK benchmark at double precision format. Several benchmarking efforts have previously aimed to characterize performance of machine learning workloads, including Deep500 [6], HPCA1500 [7], and HPL-AI [8]. The largest dataset used across these attempts is obtained from the Extreme Weather Dataset [9] of about 1.65 TB. Other benchmarks aimed at analyzing model performance include DAWNbench [10], Deepbench [11], Fathom [12], ParaDNN [13], HPE DLBS [14], and XSP [15]. The challenges and limitations of existing benchmarking efforts drive the need to develop a benchmark suite with science applications that run at scale. In this paper, we present MLPerf HPC, a benchmark suite aimed to address the limitations of prior efforts. This benchmark suite is a part of MLPerf, driven by MLCommons [16], an open engineering consortium that aims to accelerate machine learning innovation through MLPerf benchmarks, public datasets, and best practices. MLPerf Training benchmarks [17] aim to measure the performance of training models while MLPerf Inference benchmarks [18] 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) Introduce the MLPerf HPC benchmark suite, the newest ML training benchmark suite from the MLCommons consortium, and the first to specifically focus on scientific ML applications relevant for HPC systems (section II). We will describe the first two benchmark applications, CosmoFlow and DeepCAM, as well as the benchmarking methodology and process (section III).

2) Present results from the inaugural MLPerf HPC submission round in 2020 (section IV). These results feature measurements from leading supercomputing platforms around the world, innovations in scalable model-and-data-parallel training and learning algorithms, and the largest scale MLPerf submission to date.

3) Develop a systematic framework for the joint analysis of the publicly available MLPerf HPC submission results to clearly understand and compare submissions in terms of data staging, algorithmic convergence and system compute performance in a condensed set of plots that enable new insights (subsection IV-A).

4) Gain a quantitative understanding of how system-level optimizations lead to > 10× improvements in end-to-end performance for both benchmarks (closed division), what constrains these submissions from scaling further and discuss alternative learning techniques that enable further improvements by a factor of 1.1-3.4× (open division, subsection IV-B).

5) Introduce techniques to characterize memory, network, and I/O behaviours of the benchmark applications across all involved systems, relate them to MLPerf HPC submission results and, hereby, lay the groundwork for a future characterization of the submissions in extended roofline performance models [19] (section V).

II. MLPerf HPC BENCHMARK SUITE

The MLPerf HPC benchmarks are holistic in the sense that they capture critical features of emerging scientific machine learning workloads: massive data volume, large complex models, and training schemes that incorporate data-parallelism, model-parallelism, and hyper-parameter optimization. The goal is to drive innovation in HPC system and software design for machine learning workloads, especially those applications that 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. 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 standard set of benchmarks for informing the design of tomorrow’s large-scale high performance computers. The MLPerf HPC benchmarks are also unique in that they offer performance characterization capability for state-of-the-art practices in scientific discovery. The emerging domain of coupling simulations with AI motivates the first two benchmarks included in the suite, CosmoFlow and DeepCAM. The reference implementations of these applications are available at [20]. Both benchmarks incorporate massive datasets based on simulations and predict important parameters with unprecedented accuracy in their respective domains - cosmology and extreme weather. Details for each benchmark is provided in the following subsections.

Future benchmarks for the suite will be community driven, aiming to share similar characteristics with CosmoFlow and DeepCAM while also expanding the relevance of the benchmark suite to other scientific regimes where AI is quickly becoming an important tool for discovery, such as computational biology, materials science and personalized medicine.
The target quantities are four cosmology parameters, \(128 \times 128 \times 128\), into smaller cubes of size \(n \times n \times n\) and so, similar to what is done in [21], the samples are split into smaller cubes of size \(128^3\) with four red-shift channels. The target quantities are four cosmology parameters, \(\Omega_M, \sigma_8, n_s, H_0\), 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. The baseline global batch size is set to 64. To scale above this configuration, the reference implementation multiplies the baseline learning rate by the factor of batch size increase, respectively its square-root.

The target quality is chosen to be mean-absolute-error (MAE) \(< 0.124\) 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 train.

**A. CosmoFlow**

The CosmoFlow benchmark is based on the work by Mathuriya et. al. [21], continued by the ECP ExaLearn project [22]. 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 [23] binned into 3D volumetric histograms of size \(512^3\) with four channels representing different red-shift values. These massive volumes present considerable challenges for training models due to large memory footprint, and so, similar to what is done in [21], the samples are split into smaller cubes of size \(128^3\) with four red-shift channels. The target quantities are four cosmology parameters, \(\Omega_M, \sigma_8, n_s, H_0\), 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 [24] 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 \(i\)th 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 a baseline learning rate schedule consisting of an initial learning rate of 0.001 which is dropped to \(2.5 \times 10^{-4}\) at 32 epochs and \(1.25 \times 10^{-4}\) at 64 epochs. The baseline global batch size is set to 64. To scale above this configuration, the reference implementation multiplies the baseline learning rate by the factor of batch size increase, respectively its square-root.

**B. DeepCAM**

DeepCAM [26] implements a convolutional encoder-decoder segmentation architecture trained on CAM5 climate simulation data [27] with TECA generated heuristic segmentation masks [28] 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 [29] and was awarded the ACM Gordon Bell Prize in 2018 [30]. Since then, the model was developed into its current form: the ResNet-50 [31] encoder backend was replaced with an Xception [32] network, batch normalization was re-introduced and LAMB [33] replaced the original ADAM/LARS optimizer [34], [35]. 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 [36] with various filter sizes, and 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 \(1152 \times 768\) 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%.

It is critical to understand what differentiates these benchmarks from typical commercial 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 $768 \times 1152$ pixels and 16 channels, which is substantially larger than standard vision datasets like ImageNet, where the average image is $469 \times 387$ pixels with at most 3 or 4 channels. Moreover, the massive dataset sizes, 5.1 TB for CosmoFlow and 8.8 TB for DeepCAM, are over an order of magnitude larger than ImageNet (150GB) and introduce significant I/O challenges that expose storage and interconnect performance limitations. These characteristics offer unique performance differences from industrial machine learning workloads which must be addressed to optimize future systems where ML-augmented science applications are expected to play a critical role.

III. BENCHMARKING PROCESS

The MLPerf HPC benchmarking 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 and user-centric view of performance as MLPerf Training benchmarks and report time to train as the primary metric. This choice creates a metric that is directly relevant to users and customers that 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, we introduced a rule to include data staging in the measured benchmark time to capture the impact of data movement for massive scientific datasets on large HPC parallel file systems and node-local accelerated storage.

A. Measurement

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

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.

2) Timing rules: At the start of a run, the benchmark dataset must reside on the parallel file system of the HPC center and on-node caches must be reset. We do not require resetting system-level caches because it is difficult to perform consistently across different systems and is often extremely disruptive. The benchmark timer begins as soon as the dataset is touched, which includes staging into node-local storage such as an on-node SSD or RAM, if capacity is sufficient. The timer stops when the convergence criteria, as described in the rules, is met (Table II).

3) Run results: ML model training is inherently stochastic due to random initialization, dataset shuffling, etc. Therefore, to get an accurate measurement of the expected time to train, submitters must run the benchmarks 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 measured times.

4) Logging: The benchmarks use the mlperf-logging library [37], 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.

B. 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 publication. A submission can be made to either open or closed divisions. A submitter can make as many submissions in either division and each submission entry can vary in the amount of system resources used.

1) Structure: Reproducibility is a key goal for MLPerf HPC. Accordingly, 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 [38].

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 perform “hyper-parameter 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 discourages excessive hyperparameter tuning by submitters and avoids giving unfair advantages to teams with greater computational resources.

4
IV. 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 HPC systems listed in Table III, for training large-scale scientific problems. The results are summarized in Table IV for both closed and open divisions.

At first glance, we observe a system scale range of up to a factor of 8-16× in closed division for both benchmarks and results showing > 10× improvements in time to train across this range. Furthermore, we can see the effect of innovations in open division by comparing each of these submissions to the fastest submission on the same system in closed division. For instance, on the CPU-based system Fugaku, there is an 3.38× improvement in open division for CosmoFlow while for the GPU-based system ABCI, there are speedups of 2.61× and 1.12× in open division for CosmoFlow and DeepCAM respectively.

In this section, we present a detailed analysis of the submissions in a framework that allows to clearly understand and separate the effects of data-staging, algorithmic convergence and system compute throughput (training and evaluation) in time to solution from the submitted logs (subsection IV-A) and complement this with a set of highlights on implementations from few systems (subsection IV-B).

A. Analysis

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

1) Discussion of data staging time: The data staging time, $T_{staging}$, is shown in Table V for the systems where it was measured. We observe that it is very different for the two benchmarks on ABCI - by interpolation to 1,024 GPUs, staging is handled more than 5× faster (in absolute time) for CosmoFlow than for DeepCAM. This difference comes not only from the fact that DeepCAM’s data set is 73% larger than CosmoFlow’s, but also from the data compression ratio, which is 88% for CosmoFlow compared to only 23% for DeepCAM.

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

$$T_{solution} = T_{staging} + T_{compute} + T_{extra}$$

(1)

As a consequence, compression was not applied in data staging for DeepCAM.

Fig. 1. Relative breakdown of time to train normalized to range [0-1], into staging (green), evaluation (orange) and training (blue). Lower three entries on y-axis are for DeepCAM, rest are for CosmoFlow.
With $T_{compute} = T_{epoch} \cdot \#epochs$ and $T_{extra} \approx 0$ (Fig. 1), where $T_{epoch}$ is the average epoch time, we get

$$T_{solution} \approx \frac{T_{staging}}{T_{staging/epoch} + \#epochs}$$

The ratio $T_{staging/epoch}$, thus, quantifies the relative overhead in units of compute epochs that staging adds to time to solution irrespective of convergence\footnote{This number depends on the batch size through $T_{epoch}$, though.} or the exact amount of data used\footnote{As long as it does not traverse a capacity limit in the memory hierarchy closer to compute units than the staging target (e.g. RAM capacity if staging to on-node SSD).}. Since it relates the compute to staging throughput, it is, however, dependent on the model’s computational complexity. As DeepCAM is more compute-intensive than CosmoFlow, the relative overhead in “extra compute epochs” shown in the last column of Table V is reduced to a factor of 2.5 – 3.5× that of CosmoFlow from what is expected purely from dataset compressibility. The reason for DeepCAM on ABCI still having 10× the share of staging compared to CosmoFlow, is that CosmoFlow requires 4× more epochs to converge (Table VI), which causes all its submissions to have marginal share of staging (< 5% in Figure 1).

Finally, Fugaku’s staging’s times are the highest at 8,192 processors due to 16-way replication of the data set (4-way for 16,384). This is an overhead of model-parallelism and could be avoided by partitioned reading and broadcasting of data across MPI ranks within each model-parallel group.

These findings show that the overhead of staging is highly application-specific, depending on data compressibility, but also a model’s computational complexity and convergence and generally affects smaller systems with fewer epochs to converge more than larger ones.

2) Analysis of compute time: This subsection presents an analysis of the time spent in training and evaluation, $T_{compute}$ in eq. (1), after data staging is completed. It represents ≥ 90% of time to solution in CosmoFlow and ≥ 80% in DeepCAM (Figure 1).

(a) Compute analysis for CosmoFlow: We observe that the number of epochs to convergence in $T_{compute} = T_{epoch} \cdot \#epochs$ is primarily an algorithmic property of SGD (holding the data and model fixed), and as such, only dependent on the optimizer and choice of hyperparameters, 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 II, $T_{epoch}$ only depends on the value of the batch size hyperparameter. Therefore, we provide an analysis of the epoch throughput $T_{epoch}^{-1}$ and number of epochs to converge separately for each of the submissions by means of Figure 2.

Submission parameters: 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 submission is shown in Figure
Fig. 2. Compute analysis of CosmoFlow with figures on a) parameter choice, b) epoch scaling (dependent variable on x-axis), c) throughput scaling (arrows in ABCI-512 explain how to read y-axes, distribution is given for the combined/epoch throughput, curved lines illustrate averaging of training and evaluation to combined/epoch throughput), 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).

2 (a). The ratio of batch size to number of compute units gives the amount of samples processed by a single compute unit on average per train step. Together with the parallelism column in Table IV that characterizes the data-parallel unit per train step, we can observe that all CosmoFlow submissions with a ratio \( \geq 1 \) chose a purely data-parallel implementation - Piz Daint being the only with a local batch size of 2, all others 1 - whereas those with a ratio < 1, i.e. Fugaku-8192 and Fugaku-16384, used model-parallelism in addition within each data-parallel unit of 16 and 4 CPUs respectively, resulting in a hybrid form of model- and data-parallelism that will be discussed in subsection IV-B1. We note that submissions with the same batch size to number of compute units ratio (constant along diagonal lines) that use the same data-parallel unit (Table IV) are related by (data-parallel) weak scaling. From another point of view, this ratio roughly behaves inversely proportional to the communication intensity, i.e. the amount of communication per computation, per train step for a given type of parallelism\(^1\), so that regions of different communication intensity can be identified in Figure 2 (a).

Epoch scaling: 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 train steps to solution, which puts a limit on data-parallel scalability. That is, once an increase in the batch size leads to a larger number of train steps to solution, a system can no longer train a model faster by growing the compute resources proportionally (ignoring caching effects in the memory hierarchy). The submissions \( \star ABCI-2048 \) and \( \star Fugaku-16384 \) are close to this limit (cf. Figure 4(a)) and the specialized techniques to converge at these very large batch sizes will be discussed in greater detail in subsection IV-B1. The remaining submissions all closely follow the reference implementation described in subsection II-A. This turns out to scale efficiently to a batch size of 256 with only \( 1.3 \times \) more epochs to converge for \( 4 \times \) batch size increase from 64, but past this point becomes significantly harder to train and less stable (\( 1.6 \times \) epochs for doubling the batch size). Closed division

\(^1\)Assuming communication to be roughly constant per train step with data-parallel scale-out and the amount of computation scaling with the batch size.
submissions were limited to a maximum batch size of 512 due to convergence issues at batch size 1,024 that were only overcome by Cori-KNL-1024 in open division with a slight modification of the learning rate schedule (section IV-B3).

**Throughput scaling:** In Figure 2 (c), compute throughput is shown as a function of the number of compute units and, implicitly, the batch size through the shared x-axis with Figure 2 (a). For each submission, we plot sample throughput (left axis) for training (▲) and evaluation (●) as well as the combined sample throughput (# samples/T\_epoch, distribution with mean at ○) according to the split of the data set (80% training and 20% evaluation) at the abscissa corresponding to the number of compute units (illustrated on the ABCI-512). On the diagonal we find lines of constant per-compute-unit throughput, commonly used to analyze scaling efficiency. Dividing the combined throughput by the overall number of samples in the data set, we obtain the epoch throughput, T\_epoch, which can be read off for the distribution from the right scale. Note that throughput is only algorithmically relevant together with a specified batch size, so that Figure 2 (c) and (a) are meant to be read together. The resulting pair of plots allows us to understand the scaling efficiency of training, evaluation, and combined throughput for submissions that relate through weak (same data-parallel unit in Table IV and same diagonal in Figure 2 (a)) or strong (same batch size in Figure 2 (a)) scaling, or extrapolation thereof, and compare different systems with each other.

Interestingly, we observe that 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, i.e. PizDaint-256, ABCI-512 and *ABCI-2048, benefit from caching the dataset in RAM. As a result, these systems spend very little (< 5%) of their time in evaluation (Figure 1). This is different for the CPU-based systems that are used at larger processor count and, thus, higher RAM capacity. To reduce the I/O-related performance penalty, smaller systems could selectively cache the evaluation dataset in RAM.

At the other end of the scale, we can observe the network effects on training throughput. Specifically, the difference in scaling efficiency between training and evaluation for submissions related by weak-scaling shows that training at scale increasingly becomes network-bound whereas evaluation with its low communication overhead almost scales ideally. This is the case for e.g. ABCI-512 and *ABCI-2048 - the system with the highest measured throughputs in this round - with 63.2% efficiency for training vs. 98.2% for evaluation and similarly CoriKNL-512 and *CoriKNL-1024 (92.6% for training vs. 100.3% for evaluation).

A further interesting point is the relation of HAL-64, CoriGPU-64 and ABCI-512 that all have the same accelerator type but a different CPU-to-GPU ratio. Without the optimizations on ABCI described in IV-B2, its per-GPU training performance is similar to HAL-64. Both of these systems have one CPU per 2 GPUs and the same number of GPUs per node. In contrast, Cori-GPU represents a more consolidated system with 8 GPUs per node, which reduces the load on the network, but also with smaller CPU resources per GPU at one CPU per four GPUs. This difference can play a role in implementations with significant CPU-overhead to keep the GPUs busy and here in parts explains the uniform difference seen in training and evaluation throughput between HAL and Cori-GPU.

**Compute time:** Figure 2(d) shows T\_compute - the outcome of the competition between epoch and throughput scaling when growing the system. To obtain it, we join the epoch scaling (Figure 2b) and throughput scaling (Figure 2c/a) of each submission on the batch size along the shared axes. The compute time T\_compute = T\_epoch \cdot \#epochs is constant along diagonal lines indicated. We observe 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 (similarly for Cori-KNL-1024 and PizDaint-256). As a further insight, we are able to trace back the run-to-run variation in time to solution, which is 2.8% for *ABCI-2048, *Fugaku-16384 and 11.1% for all other CosmoFlow submissions, to the number of epochs rather than system throughput. This is confirmed by a logarithmic principle component analysis (log-PCA, Figure 3), which shows a first component with strong horizontal alignment in all submissions. Furthermore, we see that Frontera-RTX-64 exhibits especially high relative throughput variability, which is explained by the parallel Lustre filesystem from where data is loaded continuously during training and that is shared by thousands of users at any time. Finally, the technique of grouping throughput- and epoch-scaling plots together in this setting 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 additional throughput measurements in Figure 2(c) at batch size with known epochs to convergence or approximately** by data-parallel extrapolation of throughput along diagonal lines of

**The curved lines from training (▲) and evaluation (●) rooted at the combined mean (○) illustrate the averaging by holding the point cloud together.**

**Ignoring network effects.**
constant throughput per compute unit\footnote{Weak scaling: batch size/# compute units = const., ignoring effects of dataset size} and joining that value with the known epoch scaling from Figure 2(b) on the batch size. Further details on CosmoFlow’s results are presented in section IV-B.

(b) \textit{Compute analysis for DeepCAM}: Figure 1 shows that more than a third (36.3\%) of time to solution is spent in the evaluation phase on Cori-GPU compared to only 4\% (closed) and 1\% (open) on ABCI. The reason for this is not primarily the different evaluation throughput (Table VI), but that evaluation is triggered after a fixed number of training steps instead of once per epoch and delayed in open division. As a consequence, Cori-GPU calculates the IOU score $8 \times$ and $35 \times$ more often than ABCI’s closed and open submissions respectively. We omit a graphical analysis of DeepCAM similar to CosmoFlow here due to the lack of space (it is available from the linked artifact in the appendix).

(c) \textit{Compute analysis comparison}: Table VI summarises the compute characteristics of both applications in the benchmark suite on Cori-GPU and ABCI. DeepCAM requires only 20-25\% the number of epochs to converge of CosmoFlow, but the epochs growth rate as a function of the batch size is similar for both applications. DeepCAM, however, exhibits much more stable convergence (1.7\% per-run std deviation of time to solution vs. 11.1\% for CosmoFlow submissions following the reference implementation). 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 ($1.4 \times$ vs. $5.7 \times$). Comparing the resource footprint, we find that to reach convergence, the compute budget (time to solution in hours $\times$ number of accelerators/processors) is $1.5 - 2.6 \times$ larger for CosmoFlow than for DeepCAM, with correspondingly higher time to solution that is $2.6 - 2.9 \times$ that of DeepCAM for closed division. Notably, though, we see for both benchmarks that despite a relatively large increase in number of accelerators by $8 - 16 \times$, the compute budget with the right optimizations can be controlled and a decrease in $\sim 10 \times$ time to solution is reliably possible.

B. \textit{Highlights}

In this section, we present details of the implementations and present highlights from v0.7 submissions.

1) Fugaku: Submissions on the Fugaku supercomputer to CosmoFlow utilized hybrid data- and model-parallel execution. To reduce staging time, data reformattting by compressing and archiving multiple files was effective. 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 to 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.

\footnote{Using $T_{solution} \propto T_{staging}/T_{epoch} + \#epochs$ from equation 1.}
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 is implemented by extending Mesh TensorFlow so that multi-processing can be applied for both data and model parallelism, and the hybrid parallelism is applied to Conv3d layers, that have a high spatial locality, by partitioning input tensors spatially in two dimensions (Figure 4(b)).

To compare the hybrid data-and-model parallelism to pure data parallelism, Figure 5 shows a breakdown of the elapsed time for the first two epochs in CosmoFlow. The communication share increases as the number of CPUs increases in both cases. But whereas the scalability of data parallelism is limited mainly by the increase of the number of epochs (Figure 4(a)/2b), that of the hybrid parallelism is constrained by the increase of communication time (Figure 5(b)). As a result, the hybrid parallelism enabled scaling the number of CPUs in the submissions up to 8,192 with 4x4 spatial partitioning for closed division and 16,384 with 4x1 spatial partitioning for open division (2.62× and 1.98× speedup of training throughput by model-parallelism compared to Fugaku-512, resp. an extrapolation thereof to batch size 4,096, Figure 2(c)). There is still room for improving the scaling efficiency of the hybrid parallelism further by reducing the communication overhead. In particular, we expect that parallelizing the halo-exchange of the Conv3D-layers within the model-parallel group with the evaluation of the layer in the non-halo region will give a significant speedup.

2) ABCI: For both of the benchmarks, data reformattting by compressing and archiving multiple files was effective to reduce data staging time. Data shuffling was applied only among 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. When applying the three optimizations sequentially on a single GPU, DALI is effective for both training (1.26X speedup) and evaluation (1.69X), adding mixed precision is effective only for training (1.77X), and increasing validation batch size further improves evaluation (3.03X). With data-parallelism, training time scales up to batch size 512 at a local batch size one for closed division. For open division, after the hyperparameter tuning techniques mentioned in section IV-B1 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 significantly (Figure 4(a)). We have already seen a manifestation of this in the weak-scaling analysis of ABCI-512 and +ABCI-2048 in subsection IV-A2 when comparing training and evaluation scaling efficiency (Figure 2c).

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. Hyperparameters were tuned 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 after hyperparameter tuning. Especially tuning the warmup steps was effective to reduce the number of epochs to convergence. For the open division submission, the Gradient Skipping (GradSkip) technique [47] was applied that 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 training. Effectively, GradSkip skips the gradient calculation of these layers in the backward pass. For our submissions, we apply it in two steps. After an average of 53.8% train steps, we skip the first 62 out of 301 layers (12.4% speedup) and after another 20.1% train steps, we skip the first 260 layers (22.1% speedup over no GradSkip).

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 [41], [42].

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 by a factor of 0.5 at 96 epochs was added to the schedule in section II-A 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

| Benchmark | Submission | Batch size | # Epochs | # GPUs | Training throughput/# acc. (samples/second) | Evaluation throughput/# acc. (samples/second) | Time to solution (minutes) | Compute budget (h : acc) |
|-----------|------------|------------|----------|--------|---------------------------------|---------------------------------|----------------|---------------------|
| CosmoFlow | Cori-GPU-64 | 64         | 53.8 ± 4.85 | 64     | 12.07 ± 0.09                    | 21.17 ± 0.56                    | 364.73 ± 32.77 | 389.04             |
|           | ABCI-512   | 512        | 100.06 ± 13.50 | 512 | 26.59 ± 0.90                     | 51.88 ± 0.68                     | 34.42 ± 4.03     | 293.03             |
|           | +ABCI-2048 | 2,048      | 98.50 ± 2.56 | 2,048 | 16.79 ± 0.23                    | 49.21 ± 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 ± 0.63    | 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 ± 0.14                      | 4.67 ± 0.82                      | 10.49 ± 0.23     | 178.95             |

### TABLE VI
Scaling of required epochs, throughput, time to solution and compute budget in CosmoFlow vs. DeepCAM.
show that large CPU systems like Cori can still be useful for training computationally-expensive deep learning models.

On an 8-node Cori-GPU system (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. 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) *Piz Daint*: Submissions on Piz Daint [39] at CSCS focused on two data-parallel configurations in the closed division of CosmoFlow with 128 and 256 GPUs, one GPU per node. Sarus [48], a container engine with near-native performance for Docker-compatible containers, was used 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 (Figure 6). A low cycle time, tensor fusion threshold and the usage of the hierarchical, tree-based all-reduce implementation proved to be key to achieve this performance. Single node optimizations within Tensorflow had a comparatively smaller effect on training throughput, except for tuning intra-/inter-op parallelism threads [49].

![Figure 6. Weak-scaling of training throughput for CosmoFlow on Piz Daint before (hollow symbols, dashed lines) and after (filled symbols, solid lines) applying fine-grained communication in the batch size region 128-1024.](image)

To find the optimal batch size at a fixed node count, throughput scaling due to increased local batch size and, thus, GPU data-parallelism has to be traded off against epoch scaling (Fig. 2 b). Specifically, Figure 6 shows that the throughput ratio for a local batch size 4, 2, and 1 relative to a maximum local batch size of 8 which is 91%, 64%, and 35% and roughly coincides with the strong scaling efficiency at a batch size of 1,024. Due to the particular closed division epoch scaling in the range of 100-1000 nodes (Figure 6). A low cycle time, tensor fusion threshold and the usage of the hierarchical, tree-based all-reduce implementation proved to be key to achieve this performance. Single node optimizations within Tensorflow had a comparatively smaller effect on training throughput, except for tuning intra-/inter-op parallelism threads [49].

Observations: Using an additional FP-32 memory bandwidth measure on ABCI (Table VII), we observe that the memory bandwidth of CosmoFlow on Fugaku is 3.85X smaller than on ABCI without mixed precision, even though the maximum effective memory bandwidth in a stream benchmark is similar for the two. Taking into account the throughput relations at 512 compute units in the submission results (Figure 2c) and the 1.4X speedup of mixed-precision training over...
FP32 (section IV-B2), we can compare the systems in their 
caching efficiency. It turns out that FP32 on ABCI has a similar 
(90 %) memory traffic per train step as Fugaku, whereas 
mixed-precision training transfers only 50 % as much memory, 
which underlines its cache-friendly implementation.

![Figure 7(a) CosmoFlow: GPU](image1)
![Figure 7(b) DeepCAM: GPU](image2)

**Fig. 7.** Communication patterns in CosmoFlow and DeepCAM applications.

**B. Network Bandwidth**

To understand the networking behavior, we profiled the 
heavy collective communication calls, incl. AllReduce 
operations, across all implementations. Figure 5 and 7(a) presents 
the share of time spent in communication for CosmoFlow and 
Figure 7(b) shows similar measurements for DeepCAM.

Since CosmoFlow’s reference implementation uses 
Horovod, we used Horovod Timeline [52] to obtain the 
average network bandwidth for collective communications as 
mpitrace [53] was unable 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. DeepCAM uses NCCL communication through 
NVIDIA Apex [54] in the reference implementation. 
Since Horovod timeline and mpitrace cannot be used here, 
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 Mesh-TensorFlow was used 
for CosmoFlow, Horovod timeline cannot be used here and 
we use mpitrace and mpiP [55] together to calculate the 
average communication bandwidth (results in Table VII).

**Observations:** Between 10-40% of total application time 
is spent in collective communications as we scale across 
GPUs. On Fugaku, we observe that for model-parallelism the 
scalability of the computation time is lower than that for the 
data-parallel execution. This is because model parallelism is

### Table VII

| Benchmark System | Memory Tool | Memory BW (GB/sec) | Network Tool | # units | Network BW (GB/sec) | Size (MB) | I/O Tool | I/O BW (GB/sec) |
|------------------|-------------|-------------------|--------------|---------|-------------------|---------|---------|----------------|
| CosmoFlow ABCI†  | Nvprof      | 335.4             | Horovod TL   | 512 GPUs| 3.41              | 19.97   | Nvprof  | 1.65          |
| Fugaku†          | Perf        | 110.8             | Mpirace      | 512 GPUs| 0.75              | 21.71   | Darshan | 0.51          |
| Piz Daint        | Nvprof      | -                 | Horovod TL   | 256 GPUs| 1.86              | 2.21    | Darshan | 1.46          |
| Summit           | Nsight      | 233.1             | Horovod TL   | 510 GPUs| 2.24              | 22.0    | Darshan | 1.98          |
| ThetaGPU         | Nsight      | 194.5             | Horovod TL   | 128 GPUs| 1.95              | 15.20   | Darshan | 2.36          |
| DeepCAM ABCI     | Nvprof      | 153.1             | Timer-based  | 512 GPUs| 3.73              | 37.77   | Nvprof  | 254.7         |
| Summit           | Nsight      | 254.7             | Timer-based  | 510 GPUs| 4.50              | 225.0   | Nvprof  | 110.8         |

† mixed-precision used on ABCI (memory bandwidth with FP32-training: 427.1 GB/sec), no model-parallelism used on Fugaku measurements

C. I/O Bandwidth

As the MLPerf HPC benchmarks have massive input data 
sets, it is important to understand the I/O performance. Table 
VII shows the average I/O bandwidth per worker on different 
systems. We used Darshan [56] 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 cer-
tain systems. On ABCI, Darshan cannot measure the I/O 
volume accurately since the 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:** From Table VII, it can be observed that the 
measured I/O bandwidth is similar for systems with on-node 
storage§§ and we can expect that it 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 7.7 TB. In this case, 
estimated I/O time per epoch is 7,700 GB / 256 processes / 
2.36 GB/s = 12.8 seconds, while measured average run time 
per epoch that includes the I/O time is 99.6 seconds.

VI. KEY INSIGHTS AND CONCLUSION

To summarize, we presented MLPerf HPC, a benchmark 
suite aimed at representative scientific machine learning ap-
lications with two applications, CosmoFlow and DeepCAM. 
We presented the results of initial submissions from lead-
ership supercomputers and developed a framework for the

§§This excludes Piz Daint (cf. section IV-B4) as well as Frontera-RTX 
with 64 GPUs in the submission due to insufficient on-node SSD capacity.
systematic analysis of time to solution in terms of different components from data staging, algorithmic convergence and system compute throughput. This serves the critical need in the HPC community to understand large-scale scientific ML workloads from a holistic perspective. Furthermore, we introduced a set of techniques for workload characterization in terms of I/O, memory and network performance metrics to enable the parameterization of extended roofline models and, thereby, relate MLPerf HPC application performance to hardware capabilities in future rounds.

The key insights from this round are:

- Data staging adds a highly variable overhead across applications (< 1% to 20% of time-to-train) that depends on both model and dataset characteristics. Where effective, compression and archiving of multiple files together should be done, especially on smaller systems, where convergence in general is achieved in fewer epochs.

- Accelerated storage solutions like on-node SSDs are critical for I/O-performance with large datasets. The results for CosmoFlow showed a narrow range of scale where GPU-based systems operated most efficiently by being able to fit the dataset in RAM and not yet experiencing prohibitive overhead from epoch scaling. If RAM capacity is insufficient, selectively caching the evaluation dataset in RAM should be attempted and performance can be further enhanced by improving data loader bandwidth and restricting data shuffling to intra-node (DeepCAM).

- Mixed-precision training and increasing the validation batch size significantly increase compute performance

- Efficient scheduling of communication is crucial for both data- and model-parallelism - while fine-grained communication is required in some data-parallel frameworks, model-parallel scalability heavily depends on overlapping halo exchanges with local layer-wise computations.

- Scaling to large batch sizes is challenging, requiring model-specific techniques such as special learning rate schedules, data augmentation and disabling dropout-layers that can exhibit a complicated interplay with convergence. This informs the need for efficient strong scaling methods, such as the hybrid model-and-data parallelism on Fugaku.

**Future Work:** In future releases of the benchmark suite, we aim to add new benchmarks for greater diversity and coverage of scientific ML workloads, including state-of-art models such as transformers and graph neural networks, as well as consider applications from emerging focus areas such as AI-driven simulations. We also plan to expand the set of collected metrics to enhance performance models and, thus, increase utility and relevance to HPC users.

**Acknowledgment**

This research was funded in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

**References**

[1] A. Senior, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Žídek, A. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, “Improved protein structure prediction using potentials from deep learning,” Nature, vol. 577, pp. 1–5, 01 2020.

[2] T. Hey, K. Butler, S. Jackson, and J. Thiyalalasingam, “machine learning and big scientific data,” p. 20190054, 03 2020.

[3] R. Stevens, V. Taylor, J. Nichols, A. B. Maccabe, K. Yelick, and D. Brown, “AI for Science,” 2 2020. [Online]. Available: https://www.top500.org/lists/2020/11/2021.

[4] S. Sharma, Chung-Hsing Hsu, and Wu-chun Feng, “Making a case for a Green500 list,” in Proceedings 20th IEEE International Parallel Distributed Processing Symposium, 2006.

[5] I. Ben-Nun, M. Besta, S. Hübner, A. N. Ziogas, D. Peter, and T. Hoeffer, “A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning,” in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 66–77.

[6] Z. Jiang, L. Wang, X. Xiong, W. Gao, C. Luo, F. Tang, C. Lan, H. Li, and J. Zhan, “HPC AI500: The Methodology, Tools, Roofline Performance Models, and Metrics for Benchmarking HPC AI Systems,” 2020.

[7] “HPL-AI Mixed-Precision Benchmark,” https://icl.bitbucket.io/hpl-ai/, 2021.

[8] R. Adolf, S. Rama, B. Reagen, G. Wei, and D. Brooks, “Fathom: reference workloads for modern deep learning methods,” in 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10.

[9] Y. Wang, G. Wei, and D. Brooks, “Benchmarking TPU, GPU, and CPU Platforms for Deep Learning,” CoRR, vol. abs/1907.10701, 2019. [Online]. Available: http://arxiv.org/abs/1907.10701

[10] Li, A. D., H. Wang, C. Qin, A. ˇZ´ıdek, A. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, “Improved protein structure prediction using potentials from deep learning,” Nature, vol. 577, pp. 1–5, 01 2020.

[11] T. Hey, K. Butler, S. Jackson, and J. Thiyalalasingam, “machine learning and big scientific data,” p. 20190054, 03 2020.

[12] R. Stevens, V. Taylor, J. Nichols, A. B. Maccabe, K. Yelick, and D. Brown, “AI for Science,” 2 2020. [Online]. Available: https://www.top500.org/lists/2020/11/2021.

[13] S. Sharma, Chung-Hsing Hsu, and Wu-chun Feng, “Making a case for a Green500 list,” in Proceedings 20th IEEE International Parallel Distributed Processing Symposium, 2006.

[14] I. Ben-Nun, M. Besta, S. Hübner, A. N. Ziogas, D. Peter, and T. Hoeffer, “A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning,” in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 66–77.

[15] Z. Jiang, L. Wang, X. Xiong, W. Gao, C. Luo, F. Tang, C. Lan, H. Li, and J. Zhan, “HPC AI500: The Methodology, Tools, Roofline Performance Models, and Metrics for Benchmarking HPC AI Systems,” 2020.

[16] “HPL-AI Mixed-Precision Benchmark,” https://icl.bitbucket.io/hpl-ai/, 2021.

[17] R. Adolf, S. Rama, B. Reagen, G. Wei, and D. Brooks, “Fathom: reference workloads for modern deep learning methods,” in 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10.

[18] Y. Wang, G. Wei, and D. Brooks, “Benchmarking TPU, GPU, and CPU Platforms for Deep Learning,” CoRR, vol. abs/1907.10701, 2019. [Online]. Available: http://arxiv.org/abs/1907.10701

[19] Li, A. D., H. Wang, C. Qin, A. ˇZ´ıdek, A. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, P. Kohli, D. Jones, D. Silver, K. Kavukcuoglu, and D. Hassabis, “Improved protein structure prediction using potentials from deep learning,” Nature, vol. 577, pp. 1–5, 01 2020.

[20] “MLPerf HPC Benchmark Suite,” https://github.com/mlcommons/hpc, 2021.
APPENDIX

ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

SUMMARY OF THE EXPERIMENTS REPORTED

We ran MLPerf™ HPC benchmarks, CosmoFlow and DeepCAM on several supercomputers such as Cori, Fugaku and Piz Daint with Tensorflow, Horovod and PyTorch. The details are listed in sections III and IV in the paper.

The baseline implementations are available at https://github.com/mlcommons/hpc. The logging package can be installed as the instructions listed in this URL.

```bash
# Install the package into your python environment.
git clone -b hpc-0.5.0 https://github.com/mlperf-hpc/logging.git mlperf.Logging
pip install [--user] -e mlperf-logging

# Test compliance of a specific mlperf hpc log file
python -m mlperf_logging.compliance_checker -- ruleset hpc-0.5.0 $logfile
```

a) CosmoFlow: The dataset we use for this benchmark comes from simulations run by the ExaLearn group and hosted at NERSC at https://portal.nersc.gov/project/m3363/.

The latest pre-processed dataset in TFRecord format is in the cosmoUniverse_2019_05_4parE.tf folder, which contains training and validation subfolders. There are currently 262144 samples for training and 65536 samples for validation/testing. The combined size of the dataset is 5.1 TB. Example submission scripts are in scripts and YAML configuration files are in configs directories. To run a code at NERSC, use sbatch -N 64 scripts/train_cori.sh and modify the scripts accordingly to run on other systems.

b) DeepCAM: The dataset for this benchmark comes from CAM5 simulations and is hosted at NERSC. The samples are stored in HDF5 files with input images of shape (768, 1152, 16) and pixel-level labels of shape (768, 1152).

The globus endpoint for the dataset is at https://tinyurl.com/3hn229c. The splitting scripts to split the dataset to train, test, and validation are under src/utils. Example submission scripts are at src/deepCam/run_scripts. These can be modified to run on other systems.

The v0.7 submissions are hosted at https://github.com/mlcommons/hpc_results_v0.7. There is a separate directory for each submitting organization with the details of its submission. It contains the following three sub-directories:

- **benchmarks**: This directory contains the benchmark implementations and manual for system setup used in that organization's submissions. In particular, the detailed steps including the scripts to configure a system and run the benchmark are mentioned in a README file in (submitter)/(benchmark-name)/implementations/submission-entry/ for each submission. The code used by a submission is referenced from there and available from (submitter)/(benchmark-name)/implementations/implementation-entry/ (multiple submissions can use the same code) and the system used is detailed at (submitter)/systems/(submission-entry.json).

- **results**: This directory lists all the submission results of an organization as log files, with one folder at (submitter)/results/(submission-entry)/(benchmark-name)/ for each submission. For example, https://github.com/mlcommons/hpc_results_v0.7/tree/main/Fujitsu/benchmarks/cosmoflow/implementations/fugaku_16384xA64FX_tensorflow_open contains the log files for the closed division submission to CosmoFlow on Fugaku with 16,384 CPUs. The implementation is available from https://github.com/mlcommons/hpc_results_v0.7/tree/main/Fujitsu/benchmarks/cosmoflow/implementations/implementation_fugaku_open and the system description from https://github.com/mlcommons/hpc_results_v0.7/blob/main/Fujitsu/systems/fugaku_16384xA64FX_tensorflow_open.json.

- **systems**: Here the details of the system hardware and software for each submission is listed in a JSON format. In the paper, Table III lists the overall configuration of the systems while Table IV lists that used for the v0.7 submissions. For example, https://github.com/mlcommons/hpc_results_v0.7/tree/main/LBNL/systems lists out Cori and Core-GPU configurations used for different submissions.

Further details on the directory structure of the v0.7 submission results can be obtained from https://github.com/mlcommons/policies/blob/master/submission_rules.adoc.

The analysis of the v0.7 submissions presented in section IV-A can be reproduced for both CosmoFlow and DeepCAM (for DeepCAM not shown in the paper) with the framework available at https://github.com/lukasgd/mlperf_hpc_results_v0.7_visualization (results are in the branches hpc_results_v0.7_cosmoflow and hpc_results_v0.7_deepcam).

The full instructions to run the workload characterization experiments in Section V (memory, network and I/O measurements) on the systems in Table VII are provided on https://github.com/memani1/mlperf_dealpc-workload-characterization.

c) Author-Created or Modified Artifacts: Persistent ID: https://github.com/mlcommons/hpc

Artifact name: MLPerf HPC Benchmark Suite Reference Implementation
Persistent ID: https://github.com/mlcommons/hpc_results_v0.7
Artifact name: MLPerf HPC Benchmark Suite Submissions v0.7
Persistent ID: https://github.com/lukasgd/mlperf_hpc_results_v0.7_visualization
Artifact name: MLPerf HPC Benchmark Suite Submissions v0.7 Results Visualization
Persistent ID: https://github.com/memani1/mlperf_dealpc-workload-characterization
Artifact name: Workload characterization of MLPerf HPC benchmarks