# Preprocessing Pipeline Optimization for Scientific Deep Learning Workloads

Khaled Z. Ibrahim and Leonid Oliker
*Applied Mathematics & Computational Research Division*
Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA
{kzibrahim, loliker}@lbl.gov

***Abstract*—Newly developed machine learning technology is promising to profoundly impact high-performance computing, with the potential to significantly accelerate scientific discoveries. However, scientific machine learning performance is often constrained by data movement overheads, particularly on existing and emerging hardware-accelerated systems. In this work, we focus on optimizing the data movement across storage and memory systems, by developing domain-specific data encoder/decoders. These plugins have the dual benefit of significantly reducing communication while enabling efficient decoding on the accelerated hardware. We explore detailed performance analysis for two important scientific learning workloads from cosmology and climate analytics, CosmoFlow and DeepCAM, on the GPU-enabled Summit and Cori supercomputers. Results demonstrate that our optimizations can significantly improve overall performance by up to 10× compared with the default baseline, while preserving convergence behavior. Overall, this methodology can be applied to various machine learning domains and emerging AI technologies.**

***Index Terms*—Deep Neural Network, Pytorch, TensorFlow, Preprocessing, Compression, CosmoFlow, DeepCAM**

## I. Introduction

Scientific machine learning (SciML) is promising to radically transform the frontiers of scientific discovery. The combination of high-performance computing systems, unprecedented data sets from scientific facilities, and new generation of predictive algorithms have the potential for significant breakthroughs that will impact almost every scientific domain. Additionally, the commercial success of machine learning methods is driving hardware vendors to incorporate special architectural designs to accelerate key learning computations. Examples include NVIDIA GPUs [1], Google TPUs [2], GraphCore [3], Cerebras [4] and Sambanova [5].

However, despite the tremendous computational potential of these hardware technologies, the cost of data movement often remains a significant bottleneck. This has become increasingly exacerbated in recent years, where computational nodes can deliver multi-peta flop half-precision arithmetic, which dwarfs the data transfer bandwidths at various levels of the deep memory hierarchy. For instance, to amortize the overhead of moving the data from a node-local NVMe disk to the GPU-accelerator, the workloads need *three orders of magnitude* of computational flops for each moved byte, which is unattainable for even high arithmetic intensity workloads such as deep neural networks (DNN).

To mitigate the cost of this disparity, applications should carefully minimize data movement and orchestrate data migration in a way that reduces or hides their cost. This study addresses this challenge by exploring techniques to reduce the data movement from the storage system to the deep learning compute engine. This is accomplished via the development of domain-specific encoding and decoding plugins, which have the advantages of reducing the required data movement, thus allowing a larger set to fit in the memory system, and accelerating data prepossessing execution on the learning engine. Our encoders also apply fusion and reordering of the decompression steps with application-specific preprocessing computations to reduce computation and data movement. Additionally, our approach enables a data decoding scheme implemented on the hardware accelerators, further reducing the overall runtime.

We demonstrate the impact of our optimization scheme on two important scientific deep learning workloads from the HPC MLPerf benchmark [6], CosmoFlow [7] and DeepCAM [8], in the domains of cosmology and climate analytics. Results show that our methodology for optimized preprocessing significantly reduces communication requirements and improves performance by up to 10×, while preserving convergence behavior. Overall our contributions include:

- Analyzing the contents of samples data for two scientific deep learning workloads, CosmoFlow and DeepCAM, to devise a compressed format suitable for execution on GPU-accelerated architectures.
- Implementing specialized preprocessing plugin to improve the speed of feeding data to the deep learning pipeline for the two applications.
- Demonstrating the preservation (or improvement) of convergence properties.
- Evaluating our implemented schemes on three HPC platforms, using a variety of configurations, and demonstrating speedups of up to 3× and 10× for DeepCAM and CosmoFlow, respectively, compared with the baseline.

## II. Data Movement and Preprocessing Challenge for Deep Learning Workloads

Although deep learning workloads are known to have a high computational intensity, significant time can be spent in migrating the data into the accelerator's memory system. In an HPC environment shown in Figure 1.a, a training sample could originate from a shared file system traversing multiple hops until reaching the accelerator optimized for conducting a training or inference task. For training, a sample is repeatedly accessed, proportional to the number of training epochs. The ability to cache the training set depends on the number of samples assigned to a node and the capacity of the storage or memory hierarchy. For instance, if the samples assigned to a node fit in the host CPU memory, a sample traverses step ❶ & ❷ once, while `step` ❸ & ❹ are repeated throughout the training session. If the dataset per node fits in the node NVMe, but not in memory, the `step` ❷ & ❸ & ❹ are repeated.

A challenge associated with this migration path is the potentially large arithmetic intensity required to hide the memory latency. For example, the NERSC Cori GPU, the NVMe node bandwidth is 3.2GB/s shared across 8 GPU, thus the cost of moving a single byte is 2250× the cost of moving a byte from the GPU's local HBM memory. If the dataset fits in memory, the cost moving data is reduced to approximately 56×. Feeding four GPUs concurrently makes the cost for moving a byte across the PCIe bus 224× that of the accelerator memory. The potential of caching data in the memory systems depends on the sample size and the number of samples assigned to a node. In general, reducing the input sample size, for instance through compression, enables caching more samples in the host CPU memory.

The preprocessing logic, `step` ❸, may involve decompression of a sample or applying some augmentation operators. For instance, in CosmoFlow, the preprocessing involves applying a *log* operator to all points within a sample. In image classification applications, preprocessing of an image may involve rotating, resizing, flipping, etc. Some of these preprocessing steps carry serialization logic and as such can be executed more efficiently on the CPU. However, when possible, efficiency can be increased by offloading preprocessing computations to the accelerators.

Figure 1.b shows the optimized migration path developed in this work. The samples are initially encoded in `step b.`❶ to enable efficient migration and processing on the target accelerator architecture. This enables reducing the data movement, allowing the dataset caching on the nearest level of the memory system, and enabling efficient decoding (or decompressing samples) on the accelerator. Additionally, we perform complex preprocessing operators on the GPU, moving `step a.`❸ to `step b.`❺ for accelerated preprocessing. Comparing Figure 1.`a&b`, a reduced sample size potentially lower the impact of the bottleneck transfer at `step a.`❷ and increase the frequency of performing `step b.`❹, leveraging bandwidth that is one to two orders of magnitude larger. Moreover, offloading the sample decoding and preprocessing to GPUs adds the additional benefit of fewer data moved across the bandwidth-constrained system resources (PCIe, NVLink, etc.). It also leverages the accelerator computational power in preprocessing samples.

## III. Related Work

Machine learning (ML) algorithms are data-intensive applications targeting various domains in our daily life, including

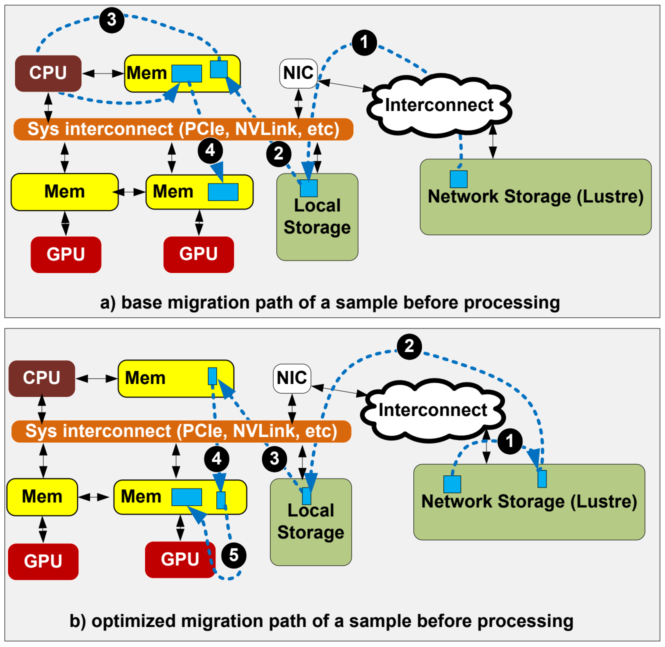

CPU
Mem
NIC
Interconnect
Sys interconnect (PCIe, NVLink, etc)
Mem
Mem
Local Storage
Network Storage (Lustre)
GPU
GPU
a) base migration path of a sample before processing
CPU
Mem
NIC
Interconnect
Sys interconnect (PCIe, NVLink, etc)
Mem
Mem
Local Storage
Network Storage (Lustre)
GPU
GPU
b) optimized migration path of a sample before processing

Fig. 1. In deep learning workload, a sample migrates from the storage system to the highest level of the memory hierarchy, where the capacity allows holding all the samples accessed by the compute node. In the optimized workflow, the samples are encoded to reduce their size and to improve the chance of having them reside in a memory level closer to the accelerator. Encoded samples are then decoded by the accelerator.

image classification, natural language processing, and health care. The use of ML in high-performance scientific computing has a long history [9] and has recently been targeting new areas in scientific computing, including predictive, design, modeling, and analytic. Example applications include the use of ML techniques [9] in the design for new material, cancer research, and protein folding. In particular, the benchmarks we are considering in this study results from community efforts to implement a full scientific machine learning benchmark, MLPerf HPC [6]. Similar to the standard ML, the scientific ML (SciML) landscape is rapidly evolving, and the suite of benchmarks is expected to grow in future releases.

The use of high-level productivity frameworks [10] (*e.g.*, PyTorch, TensorFlow, MXNet, LBANN), mixed precision, vendor libraries (NVIDIA CUDNN, Intel OneDNN), complicates the performance analysis for application developers, who write their model using a relatively high-level declarative language, which improves productivity.

With the evolution of computational architecture, researchers continually revisit the performance benchmarking [11]–[16], modeling [17], [18], and optimizations [19], [20]. Modeling IO [21], in particular, has special importance for these workloads due to the volume of data they typically process. Ultimately, the performance of these applications is defined by the time to a desired accuracy, which intertwines multiple performance contributing factors, including statistical efficiency, hardware efficiency, and runtime efficiency. The significant impact of data movement on the performance of deep-learning workload both within accelerator and at the system level has recently been characterized [22]. Moreover, recent studies [23] shows the importance of optimizing data movement within the accelerator to improve performance.

Compression of data can reduce the volume of data transfer across the memory hierarchy. In particular, compression of scientific data based on floating-point arithmetic has also been studied intensively [24]–[27], using both lossy and lossless compression targeting a range of applications, such as visualization and scientific computation. Unfortunately, they do not provide mixed-precision solutions, specifically targeting 16-bit floating-point representation, and the support on accelerator architecture is limited. Moreover, most compression frameworks do not provide the flexibility to fuse or reorder user-level compute operations with the decompression process.

In this work, we aim to accelerate the data loading pipeline for scientific applications by leveraging custom encoding that enables efficient decoding on accelerator architectures. Our encoders leverage compression techniques interleaved with preprocessing functionalities and exploit domain-specific optimization to improve the efficiency of processing. Moreover, we designed our decoder to provide the deep learning training cycle half-precision input samples, a floating-point format not supported by the decompression frameworks we are aware of.

## IV. Scientific Machine Learning Workloads

In this study we consider two scientific machine learning applications from the MLPerf HPC benchmark suite: DeepCAM, which analyzes 2D weather images to find extreme weather phenomena, and CosmoFlow, which predicts cosmological parameters that describe the evolution of the universe by observing 3D images associated with four redshifts.

The DeepCAM climate benchmark [8] leverages deep learning to predict extreme weather phenomena, such as cyclones, from weather images such as those shown in Figure 2. This code is based on the Exascale Deep Learning for Climate Analytics paper, which won the 2018 SC Gordon Bell prize [28]. The model uses PyTorch framework and Google's Deeplabv3+ [29] to perform semantic segmentation. DeepCAM analyzes images of the size $1152 \times 768$ from the Community Atmosphere Model (CAM5) [30] climate data. A climate sample is composed of 16 images (or channels)—describing temperature, wind speeds, pressure values, humidity, etc.—stored in HDF5 files using 32-bit floating-point format (FP32). The PyTorch framework [31] leverages system-specific libraries for optimized performance. For instance, it uses NVIDIA's NCCL for distributed implementation and NVIDIA's CUDNN for efficient execution of the neural network computation.

The CosmoFlow benchmark aims at predicting four cosmological parameters that describe the evolution of the universe. CosmoFlow neural network uses the TensorFlow [32] framework to implement five layers of 3D convolutional layers and three fully connected layers. For distributed implementation, TensorFlow leverages the NCCL library to provide efficient inter-GPU and inter-node performance through the Horovod [33] library. In addition, TensorFlow installation leverages system optimized CUDNN for efficient execution on NVIDIA GPUs.

The CosmoFlow dataset uses the $128^3$ decomposition of the $512^3$ [34] samples. Figure 3 shows the dark matter evolution (at four redshifts) within a sample. The original data files are stored as HDF5 files, while the decomposed data uses TFRecord format [35] for efficient processing by TensorFlow. The latest release of the dataset [7] provides a compressed variant of the dataset using gzip, which reduces the required storage space by $5\times$.

## V. Compressability of Scientific Data

The process of data compression/encoding could be either domain-specific or general-purpose. In the domain-specific case, such as audio or image format, spatial or temporal properties could be leveraged to achieve fast compression and decoding. The use of general-purpose compression such as gzip in compressing TFRecord [36], on the other hand, focuses on balancing the compression ratio and the compression overhead without a particular target application. For example, the imagenet dataset [37] uses the industry-standard image JPEG [38] format for encoding and decoding images. Vendors provide specialized decoders to allow efficient processing of sample data. For instance, on NVIDIA GPU architectures, nvjpeg library [39] provides an accelerated functionality to decode and apply various operators on images written with this format.

However, Scientific Machine Learning (SciML) workloads lack similar standards for efficient encoding of scientific images. Unfortunately, it is often not possible to leverage industry encoding tools because scientific images can have different requirements. For example, DeepCAM is based on 32-bit floating-point data with 16 channels per image (compared with the standard 8-bit 4 channels representation).

Thus our work explores the potential of developing custom encoding/decoding strategies encompassing compression/decompression fused with computational operators specifically for preprocessing scientific images — enabling efficient decoding on GPUs, supporting FP16 precision to feed data to mixed-precision pipeline, and allowing accelerated operator preprocessing.

### A. DeepCAM Sample Compressibility

The DeepCAM datset is composed of samples containing 16 channel images. Each image is $1152 \times 768$ 32-bit floating-point values. As shown in Figure 2, the DeepCAM images within a sample typically have large areas with smooth changes. The target of the machine learning process is to find extreme weather phenomenona, which are typically associated with abrupt changes. Our analysis of these samples recognized that the $x$-direction contains the smoothest changes in values. We leveraged this smooth change to construct differential encoded segments within a line using a lossy compression technique, which effectively removes noises resulting from sensor measurement of smooth areas without significantly

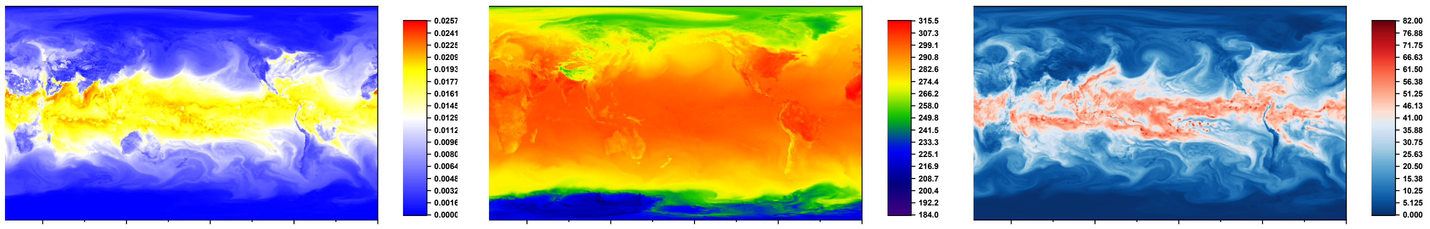
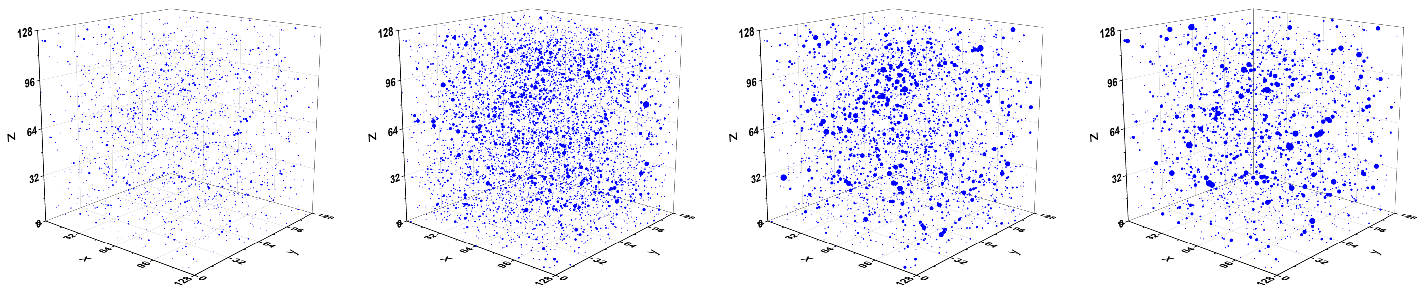


Fig. 2. Three of the sixteen channels (describing temperature, wind speeds, pressure values, and humidity at different altitudes) within a sample for the DeepCAM ML application, which analyzes 2D weather images for extreme weather phenomena. We observe spacial locality in most of these images, especially along the latitude, except at extreme weather phenomenona.



(redshift 3.0) (redshift 1.5) (redshift 0.5) (redshift 0.0)

Fig. 3. A sample of CosmoFlow with universe at four different redshifts, where the size of the dot is proportional to the $log(count\ of\ dark\ matter\ particles + 1)$. CosmoFlow analyzes dark matter images to predict cosmological parameters that describe the evolution of the universe. We notice progressive clustering with localized evolution as we approach the redshift of 0.0 (today).

affecting the quality. Areas with abrupt changes are kept uncompressed in our scheme because they potentially carry interesting climate phenomena.

Figure 4 illustrates the differential encoding scheme that we developed for DeepCAM. A sequence of values with smooth transitions has pivot value, relative to which encoding is done. We record the sequence of differences each value from its neighboring value. The exponent of these differences is clustered into groups of close values. The group of values is defined by an arbitrary number of bits, 3 in our case. The minimum is of these exponents is stored for each segment. We use 4 bits for the mantissa and reserve a bit for sign, with such a $\Delta$ of an 8-bit per difference. Our scheme allows better handling of floating-point denormalization because the exponent is interpreted depending on the meta-data associated with each segment. However, for lines with abrupt transitions or where the number of segments is large, we do not compress these lines. We also have a special encoding for the case where all neighboring values are similar. Depending on the content of the line, we use the encoding that enables the best space-saving. Additionally, we use metadata that enables independent decoding of lines, thus enabling efficient execution on accelerator architectures.

The decode process involves software emulated addition for floating-point numbers. While we emit half-precision (FP16) values, the computation is conducted in single-precision (FP32) precision resulting in a slightly lossy encoding. Overall we notice roughly 3% of the values with larger than 10% error,

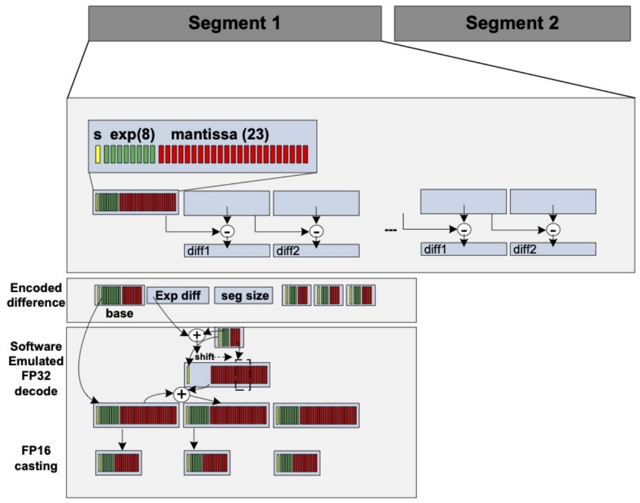


Fig. 4. The floating-point encoding/decoding mechanism that we used in DeepCAM. A line of values is split into segments. A sequence of values close in the range is encoded using differential encoding. We store the head value of a segment and encode the difference using special encoding.

primarily for small values close to zero due to floating-point denormalization.

## B. *CosmoFlow Sample Compressibility*

The CosmoFlow dataset [34] results from from the simulation [40] of the evolution of cosmological N-body dark matter using pyCOLA-based implementation [41] under initial conditions generated from the MUSIC [42] framework. The simulation output is particle counts in a 3D $512^3$-voxel histogram for 4 different redshifts. The governing cosmological parameters of interest are used as the labels associated with these samples. The full dataset represents the simulation of 10017 universes for different values of four cosmological parameters varied uniformly over a 30% spread of the mean values for four redshift snapshots per universe. Samples are decomposed into $128^3$ voxel sub-volume to fit into GPU memories.

To develop a domain-specific encoding/decoding scheme, we first analyzed the contents of the CosmoFlow samples, an example of which is shown in Figure 3. We observed that the number of unique values per sample is relatively small, in the range of few hundreds, as shown in Figure 5. The frequency associated with these unique values follows a power-law distribution, Figure 5.a. Our analysis of the full dataset shows that the particles count associated with the four redshifts, under the same cosmological parameters, are highly coupled, making the number of unique groups of values significantly less than the permutation of all unique values. For instance, in the samples shown in Figure 5.b&c, with 558 unique values, only 36944 unique groups of four values exist out of a potential $1.2 \times 10^{11}$ possibilities. These data attributes result from spatial and temporal evolution of the dark-matter particle distribution. Having few groups of these values simplifies the development of an encoding scheme using localized lookup tables. The use of these tables resulted in a compression factor of roughly 4×, compared with a 5× compression using the general-purpose gzip. However, the advantage of our scheme is that, unlike gzip, our decoder can be efficiently implemented on accelerators, such as NVIDIA GPUs.

Another important optimization enabled by our analysis is that complex preprocessing operations, such as computing a $log$ operator to the particle count within a point, are applied to the unique set of values within the sample. CosmoFlow samples have 8M values, but the number of unique values within that data is three orders of magnitude smaller. As such in the decoding phase, applying the $log$ operator before decompression is advantageous. Analyzing the full dataset, roughly 0.5M samples, we found the unique set of values and permutations vary from one sample to the other, and therefore implement a lookup table for each sample. For larger than $128^3$ decompositions, multiple lookup tables are required.

## VI. Integration of Decoder/Encoder into the Deep Learning Pipeline

To offload the decoder to the accelerator, we leveraged the NVIDIA data loading library DALI [43]. DALI provides a mechanism for loading and preprocessing data and supports multiple frameworks including PyTorch, TensorFlow, and MXNet. The preprocessing is either simple operators provided by the framework, or a complex function provided by a third-party library, such as JPEG decoding via nvjpeg [39]. Images, audio, and video codecs typically leverage domain-specific knowledge of the spatial and temporal locality for data to provide multiple levels of compression ratio, retrieval quality, and playback performance.

For our study we implemented two variants for decoding (decompression overlapped with preprocessing computation logic) the DeepCAM and CosmoFlow samples, one for the CPU and another for the GPU. Our decoders support multiple compression formats, ranging from simple lookup tables to differential encoding. We use keys of width 1 or 2 bytes for lookup tables, with lookup values of 8 bytes. These operations are highly parallelizable since there are no dependencies between threads due to the use of single key width per table. Similarly, for repeated values, we efficiently parallelize the broadcasting of constants. Thus the only difference between the CPU and GPU implementations is that the work to construct a sample on GPUs is split between threads preserving coalesced access to the memory. In contrast, on the CPU we assign different samples to different threads.

For differential encoding, the loop carried dependencies complicate the GPU implementation. Our GPU version uses hierarchical parallelism, where we assign a warp[1] of threads a copy or broadcast tasks and assign tasks that create control divergence to different warps. This hierarchical assignment allows keeping as many warps active as possible, while threads within a warp cooperatively execute inner loop tasks.

Our DALI plugins feed the sample data into the Pytorch and Tensorflow frameworks with no model changes. Thus only the data feeding module in both applications needs to be modified, while the model and its interface to the data feeder is maintained. Our data-feeder plugins provide FP16 samples, which are compatible with the automatic mixed-precision engine for PyTorch and TensorFlow. We rely on auto-casting to support mixed precision, and performed our experiments using PyTorch v.1.8.0+ and Tensorflow 2.5+[2].

## VII. Experimental Setup

We use Summit [44], a GPU supercomputer at Oak Ridge National Laboratory composed of 4,600 nodes. Each node has two IBM POWER9 CPUs with 512 GB memory and six NVIDIA V100 GPUs each with 16 GB memory connected to the CPU with NVLink. Each node has two dual-rail EDR InfiniBand connected to the infiniband switches. Each node also has an attached 1.6TB NVME storage.

We also used two special clusters within NERSC Cori system [45]. The first is cluster is Cori-V100, which has two Intel Xeon Gold 6148 CPU, with 384 GB DDR4 memory. Each CPU is accelerated with four V100 GPUs. While the GPUs are interconnected using NVLink, the connection between the CPU and GPU is through a PCIe 3 switch. Each node has four dual-rail EDR InfiniBand NIC, and has an attached 1 TB NVME storage. The second is based two AMD EPYC CPU

[1]For improved efficiency, a warp of threads in NVIDIA GPU execute instructions in a lock-step fashion.

[2]In our experiments, the availability of optimized libraries differed by the system. As such, the choice of optimized libraries are system dependant, especially as the use of containers is limited to specific system architectures. For instance, on OLCF we used opence 1.5.0 with the set of packages optimized for it.

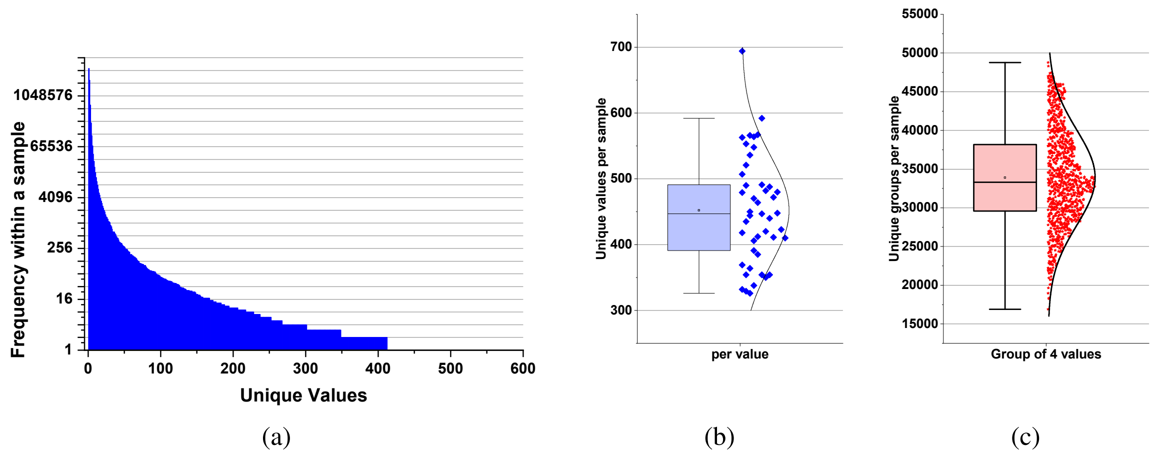


Fig. 5. Each CosmoFlow sample has a power-law distribution of the frequency of appearance for each value, as shown in (a). The number of unique values in the order of hundreds, as shown in (b), but varies from one sample to another. (c) Locality makes the number of unique groups small, enabling the use of localized lookup tables for encoding. For instance, for groups of four values, corresponding to the four redshifts in the simulation, we have few tens of thousands of permutations that can indexed with 16-bit integers.

TABLE I
SYSTEM ARCHITECTURE FOR EVALUATED SYSTEMS

| | Summit | Cori V100 | Cori A100 |
|---|---|---|---|
| Host Processor (CPU) | IBM P9 | Intel Xeon Gold 6148 | AMD EPYC 7742 |
| CPU Freq (Ghz) | 3.1 | 2.4 | 2.25 |
| Host Memory (GB) | 512 | 384 | 1056 |
| CPU-GPU Interconnect | NVLink | PCIe Gen 3.0 | PCIe Gen 4.0 |
| GPU | V100 | V100 | A100 |
| GPUs per node | 6 | 8 | 8 |
| L2 Cache (MB) | 6 | 6 | 40 |
| SM | 80 | 80 | 104 |
| Mem Capacity (GB) | 16 | 16 | 40 |
| BW to GPU Mem (TB/s) | 0.9 | 0.9 | 1.6 |
| GPU FP32 TF/s | 15.7 | 15.7 | 19.5 |
| Tensorcore TF/s | 120 | 120 | 312 |
| NVMe Capacity (TB)) | 1.0 | 1.6 | 15.4 |
| NVMe Read BW (GiB/s) | 5.5 | 3.2 | 24.3 |

TABLE II
SOFTWARE ENVIRONMENT FOR COSMOFLOW AND DEEPCAM

| | CosmoFlow | | | DeepCAM | | |
|---|---|---|---|---|---|---|
| | Summit | CoriV100 | CoriA100 | Summit | CoriV100 | CoriA100 |
| Framework | TF 2.5 | TF 2.5 | TF 2.5 | PT 1.10 | PT 1.8 | PT 1.9 |
| torchvision | | | | 0.11.1 | 0.8.1 | 0.10.0 |
| python | 3.8 | 3.8 | 3.8 | 3.8 | 3.8 | 3.8 |
| horovod | 0.21.0 | 0.22.1 | 0.23.0 | | | |
| CUDA | 11.0.221 | 11.2.2 | 11.4.0 | 11.0.3 | 11.2.2 | 11.4.0 |
| CUDNN | 8.0.4 | 8.1.0 | 8.2.4 | 8.1.1 | 8.1.0 | 8.2.4 |
| NCCL | 2.7.8 | 2.8.4 | 2.11.4 | 2.11.4 | 2.8.4 | 2.11.4 |
| DALI | 1.9.0 | 1.9.0 | 1.9.0 | 1.9.0 | 1.9.0 | 1.9.0 |
| gcc | 7.3.0 | 7.3.0 | 8.3.0 | 8.2.0 | 7.3.0 | 8.3.0 |

accelerated, each accelerated by four A100 GPUs. A summary of the characteristics of the compute systems is provided in Table I.

For the workloads, we used the MLPerf version of CosmoFlow, which is written in TensorFlow, and the DeepCAM, which is written in PyTorch. Both applications leverage automatic mixed precision to accelerate computation. The software stack used in this study is composed of multiple layers, starting from the framework layer (TensorFlow, and pyTorch) to system-specific libraries, detailed in Table II. For NVIDIA GPU-based systems, these frameworks leverage NVIDIA optimized libraries such as CUDNN for convolutional layer computation, NCCL for synchronizing the distributed model, etc. We modified only the components required for compatibility with the introduced decoder plugin based on NVIDIA DALI [43]. For Summit, we built our compute environment through cloning the opence-1.5.0 environment.

## VIII. OPTIMIZED PREPROCESSING PIPELINE EVALUATION

In deep-learning workloads, the time to accuracy is a function of the number of epochs required for convergence and the time to perform a single epoch. We, therefore, evaluate our optimized plugin for its impact on convergence and performance.

### A. Convergence Behavior with Plugins

Since our decoded samples are primarily in FP16 format, while the baseline uses FP32, we begin by examining the convergence behavior between these two formats. Note that both of our SciML codes are processed in mixed precision with auto-casting, and each DNN framework performs mixed precision with per task determination of the computation precision. As a first step, we compare the base and decoded samples with respect to the limits of precision. Our CosmoFlow decoder is not lossy when casting to FP16, while for DeepCAM compression is lossy especially for values close to zero affecting approximately 3% of the values per sample. For both applications, we use lossless compression of the labels.

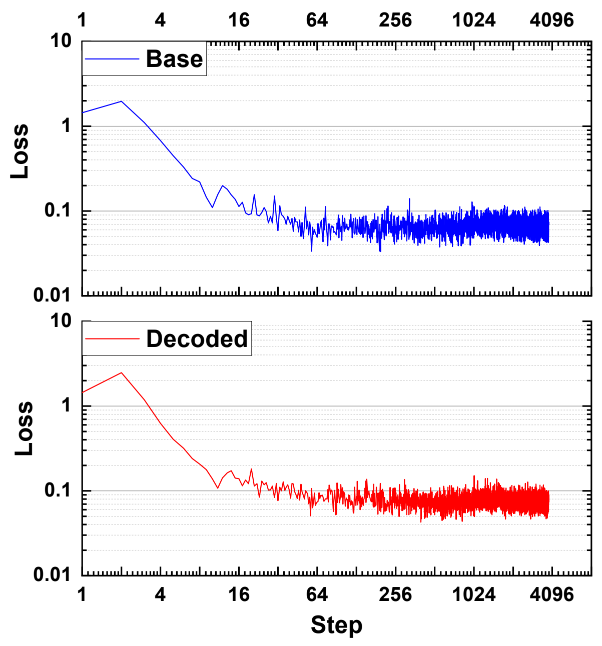


Fig. 6. DeepCAM change in the loss function (for learning samples) with the base and the decoded samples on a single GPU. Our decoded samples show identical convergence behavior to the base case.

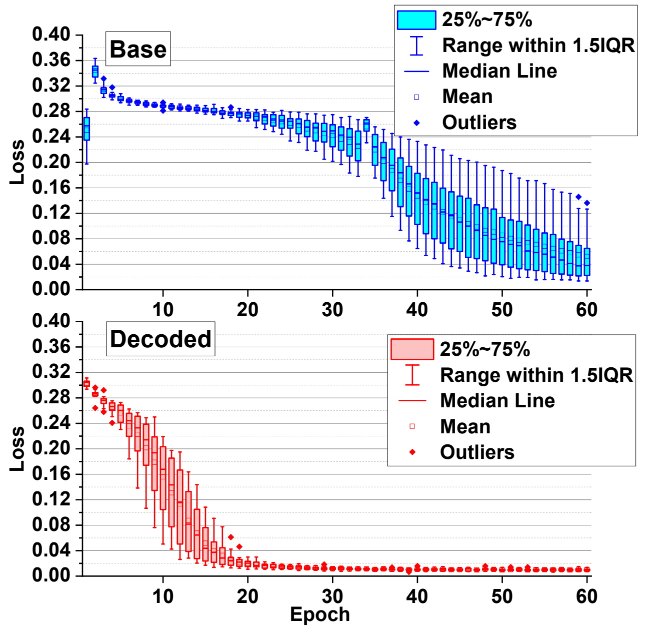


Fig. 7. CosmoFlow change in the loss function (for learning samples) with base and decoded samples on a single GPU. Our decoded samples show better convergence properties, likely due to the framework logic that addresses mixed-precision differently for 16-bit vs. 32-bit floating-point samples.

Figure 6 shows the loss function associated with the training of 1536 samples as the learning steps progress (with two samples processed per step), using a single GPU with the learning schedule parameters of the reference implementation. We observe that the base and decoded samples result in identical convergence behavior. The same behavior is also seen in the the loss function of the validation samples, which is omitted for brevity.

CosmoFlow results, shown in Figure 7, track the convergence for 16 repetitions of the learning tasks with similar learning schedule on a single GPU, using 128 samples. We tracked the loss of learning samples across different epochs (full traversal of the full set of samples), and repeated the experiments following the MLPerf HPC submission guidelines that require results across multiple runs. For CosmoFlow, the convergence is known to vary widely between runs [6]. Such change in convergence behavior is related to the traversal of samples, which uses random shuffling, and internal DNN processing, such as random weight drop-offs.

Figure 7 highlights the surprising property of better convergence for our decoded samples. This advantage is visible not only in the lower loss values, but also in reduced variability. This is likely an artifact of TensorFlow's internal logic that processes FP16 and FP32 differently. However, our study does not leverage these improved convergence properties to gain a performance advantage via reducing the number of epochs for our decoding strategy since we have not fully conducted a hyperparameter optimization to tune for the base or decoded samples. We merely used the same learning schedule (warm-up, learning rate change with rank count and phases, etc.) for both classes of samples. Such parameter search is influenced by the target the level of concurrency in processing samples (global batch size), the learning schedule, the selected optimizer, in addition to the precision of computation.

Thus for the remainder of the discussion, we compare performance per computation step, while fixing the learning schedule and optimizer, and using mixed precision for both sample types. Finding optimal hyperparameters for learning, which is out of the scope of our study, is equally crucial for achieving the best time to solution.

## IX. Performance Evaluation

This section summarizes the performance evaluation of CosmoFlow and DeepCAM in the MLPerf HPC training benchmark suite [6] from the MLPerf benchmark suite [13] on the studied platforms.

### A. *DeepCAM Performance with Optimized Preprocessing*

For the DeepCAM plugin, we present the performance of our decoder on both CPUs and GPUs. Note that the irregular pattern for decoding has the potential for a more efficient CPU implementation. Additionally, some DALI plugins use mixed architecture (CPU+GPU) version to optimize performance.

The baseline for our experiments is the CAM5 dataset distributed. We use two dataset assignments per node, a smaller 1536 samples per node case, which can be cached in the memory system. The bigger data set is 8× larger (12,288) and less likely to fit in memory. These two datasets were chosen because the number of samples assigned to a node in HPC environments depends on the node count and the number of samples used in training.

Our work also explores the performance difference between *staged* data within the attached node storage (local NVMe) versus *unstaged* streaming from the network storage system. We explored this option because some HPC systems have nodes containing locally attached NVMe, while other systems rely solely on shared storage attached to the interconnect. Exploring these design alternatives enables the exploration of architectural configurations outside the studied systems. Moreover, by exploring different batch sizes, we investigate cases where data scientists may choose a small global batch size to improve convergence, thus reducing local batch size per GPU.

As shown in Figure 8, the baseline configuration performs slightly better with batching and data staging, but suffers a significant slowdown of $1.2 - 2.4\times$ for a large dataset. These performance characteristics highlight the performance dependency on data movement. Note that the baseline performance does not improve when migrating from the Cori-V100 (middle) to the faster Cori-A100 system (right), due to similar CPU-GPU bandwidth for both platforms for the sample transfer sizes. While for Cori-based experiments, both cpu-based and gpu-based plugin improves the performance, for Summit only gpu-based plugin improves the performance.

We measured for Cori-V100 node and Cori-A100 a peak PCIe host-to-device transfer bandwidth of 12.4 GB/s, and 24.7 GB/s, respectively. For the range of transfer sizes of 4 to 64 MB, corresponding to our sample sizes, where memory is pageable[3], the bandwidth range is 4-8 GB/s for the V100 node and 6-8 GB/s for the A100 node. Effectively, both nodes have close bandwidths for transferring samples to the accelerator, which hinders the opportunity to benefit from the computational capabilities of the new A100 accelerator.

The decoder plugin results show that our approach significantly accelerates performance, especially for the Cori-A100 where throughput increases by up to $3.1\times$ compared with the baseline. Performance generally improves with increasing batch size, although Cori-A100 system (with larger memory capacity) suffers a small degradation with a batch size of 8. Our profiling analysis indicates that the framework choice of the computational kernels, which is impacted by the batch size, is the cause of such behavior. In general, the GPU version of the decoder outperforms the CPU counterpart, due its efficient implementation as well as the reduced pressure on the system bus in moving data between the host and the GPU memories. As a result, the GPU plugin is up to $1.5\times$ faster than the CPU for unstaged data. Additionally, our plugin also leverages the increased capability of the A100, resulting in a speedup of up to $2.2\times$ compared with the older generation V100. The decode operation overhead is small taking roughly 4% of the processing time per sample.

The performance improvement with the plugin on Summit is quite lower than what we observed on Cori systems. The nvlink connection between the host processor and the accelerator reduces the potential gain with the plugin as the baseline is higher for Summit. At batch size of 4, the 6-V100 Summit node outperforms an 8-V100 Cori node, while expected performance should be around 75%. Meanwhile, the ability of host processor to process the software stack and the level of optimization for the software stack appears to be lower for Summit as compared with CoriGPU. As such, we notice the lower performance of the cpu-based plugin and limited improvement with gpu-plugin (limited to $1.3\times$). With the gpu-plugin, a Summit node delivers at best 60% of the Cori-V100 performance.

Figure 9 breaks down the performance profile for processing a sample on the Cori V100 and A100 nodes. We present key grouped activities for two timelines during the execution, the host CPU timeline and the accelerator GPU timeline. Results show that for the baseline, despite the efficient execution on the A100, the cost of host CPU preprocessing and host–device data movement did not improve. Using our loader plugin, not only improves data transfer time but also speeds up CPU preprocessing while reducing the fluctuations captured during the model synchronization `allreduce` in the back propagation phase.

## B. CosmoFlow Performance with Optimized Preprocessing

For the CosmoFlow plugin, the overall speedup of our GPU decoder implementation achieves up to $10\times$ speedup. We compare performance with an uncompressed tfrecord baseline and the compressed tfrecord, using gzip, which is part of the standard benchmark implementation. This compression version is intended to reduce the impact of the well known CosmoFlow IO bottleneck [19]. Although the gzipped files are roughly 75% the size of our encoded samples, the decompression can only be performed on the host CPU, since there is no existing GPU version for gunzip.

We explore the performance of three systems, OLCF Summit, Cori-V100, and Cori-A100. Similar to DeepCAM, we used two datasets sizes consisting of 128 and 2048 samples per GPU. The constant number of samples per GPU allows us to normalize performance between Cori and Summit. Our experiment ran 100 and 15 epochs for the small and large set, respectively. Our GPU plugin significantly outperforms the CPU implementation, and its results are omitted for brevity.

Figure 10 presents results for the small data set, with batches sizes ranging from 1 to 8. Note that the difference between the Cori-V100 and Summit-V100 and is mainly due to the use of 8- vs 6-GPUs per node for Cori and Summit, respectively. Observe that the base case does not change significantly with the batch size. Additionally, on most systems, the use of gzipped formatting reduces throughput by up to $1.5\times$, where the benefit of reduced IO pressure is negatively offset by increased decompression time.

For the small dataset, our encoder has the greatest impact on Summit, where speedup improves by 5-$8\times$, where samples are likely cached in memory. This improvement is largest for a batch size of one and gets smaller with the batch size increase. Acceleration on Cori-V100 and Cori-A100 is also significant,

[3]Deep learning frameworks typically use pageable memory to avoid running out-of-memory with pinned memory.

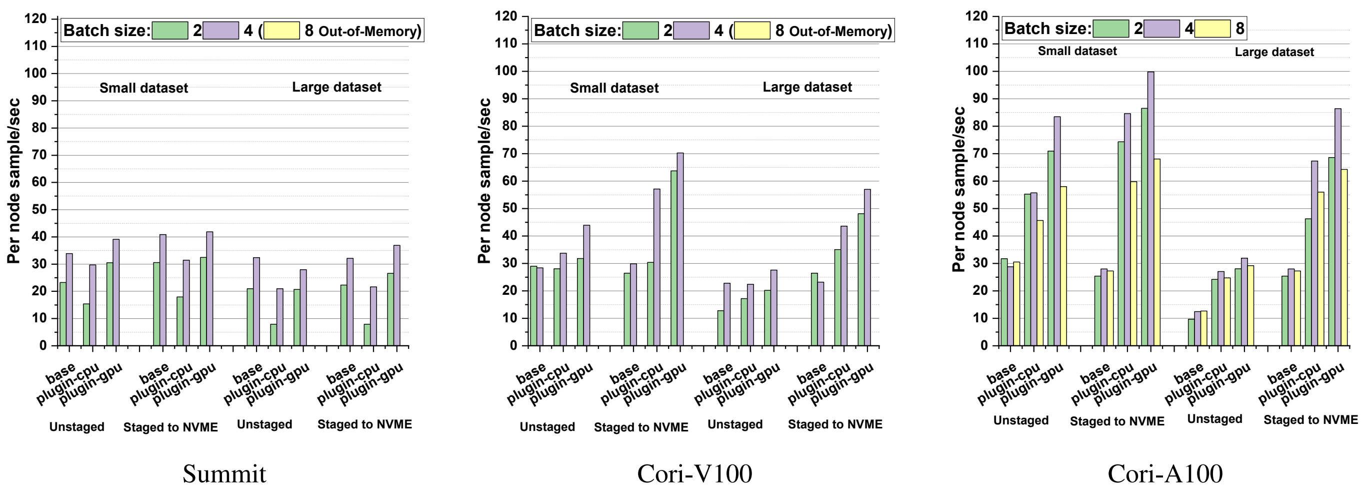


Fig. 8. Improvement of throughput of DeepCAM (in sample/s for full node) using our two decoder plugins (CPU and GPU) on Summit, Cori-V100, and Cori A100 platforms, using a small (1536) and large (12,288) samples per node. *Staged* experiments leverage node-attached NVMe, while *Unstaged* stream data directly from the interconnect. On Cori, our CPU and GPU decoder plugins deliver up to 2.5× and 3× speedup, respectively.

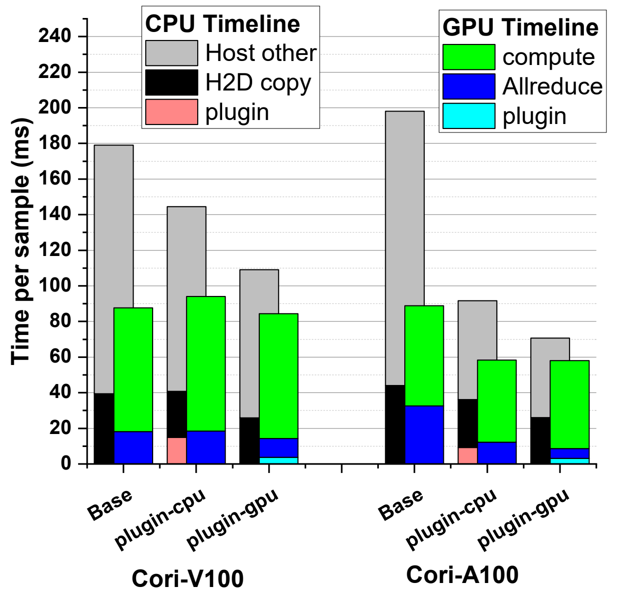


Fig. 9. Time breakdown of DeepCAM performance on the Cori V100 and A100, comparing our CPU- and GPU-plugin implementations, using the small sample set with a batch size of four. Improving preprocessing reduces CPU overhead and the variability captured by the model synchronization during the allreduce operation.

achieving a 3-4× speedup. The decode operation overhead is negligible, taking less than 1% of the total processing time of a sample. The presented improvements are due to the reduction of data movement and GPU offloading of the *log* operator and applying it only to unique values within a sample. For the baseline, we note that the performance for Cori-V100 and Cori-A100 is quite close despite the different hardware capabilities due to the bottleneck of moving the samples files from the file system to the accelerator memory. Finally, staging slightly improves performance for small datasets. For the larger dataset of 2048 samples per GPU in Figure 11, staging improves node performance by up to 1.5× for Cori-V100 and A100. The difference for Summit is within 10%. The speedup for the large dataset is up to an order of magnitude.

Figure 12 presents the Summit and Cori-V100 execution time profile of CosmoFlow, highlighting the time breakdown on the host CPU and GPU for the small data set. Observe that performance is dominated by the CPU preprocessing activities for the baseline, hindering efficient accelerator execution. The data movement cost between the host memory is higher for Cori-V100 because of the PCIe 3.0 bus connecting the CPU and GPU. Summit, on the other hand, uses NVLINK, which roughly provides 3× the bandwidth of the PCIe 3.0. Our analysis also shows that decompression of gzipped TFRecord is more efficient on Cori-GPU systems, but nonetheless slowdowns the overall runtime. Finally, we observe that our developed plugin reduces the preprocessing overhead significantly, revealing the raw performance of the V100&A100 accelerators.

## X. Conclusions

The AI revolution has a profound impact on scientific discovery in important scientific disciplines, including cosmology, material science, and bioinformatics. Unfortunately, there is a growing disparity between the rapidly evolving computational capabilities of specialized hardware accelerators and the ability to feed data for processing through the deep and complex memory hierarchy. To successfully address these challenges, the community needs to develop data reduction and mitigation techniques that can be integrated into existing software ecosystems to leverage broad functionality and portability. We present a methodology that incorporates domain-specific application analysis and architectural insight to develop successful data reduction and accelerator-preprocessing techniques for the DeepCAM and CosmoFlow benchmarks on three HPC platforms.

Our approach successfully preserves the convergence characteristics, even for the cases where we adopt lossy com-

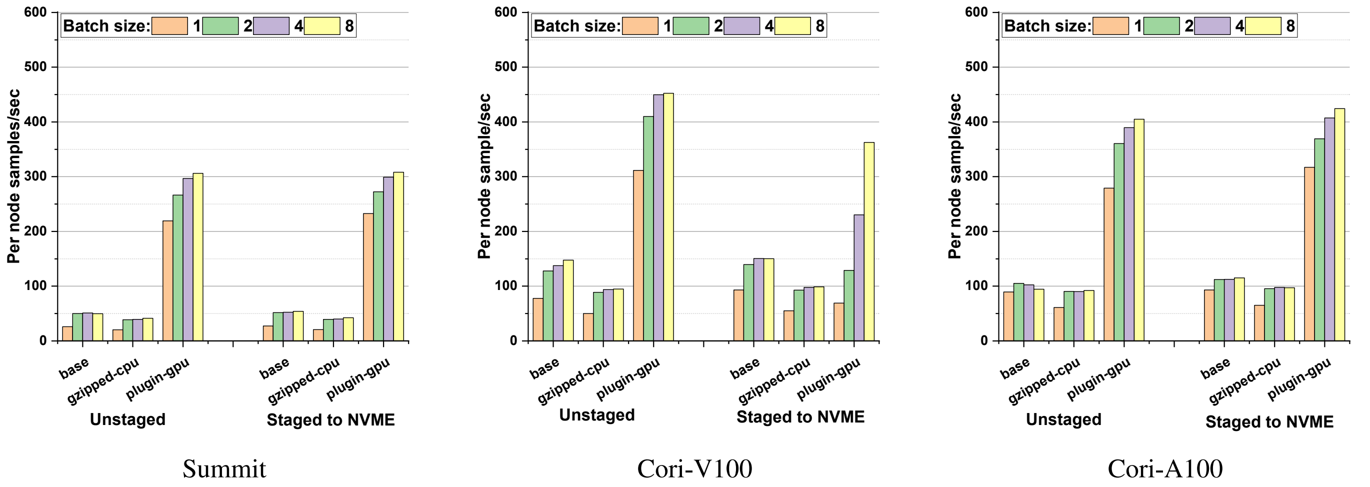


Fig. 10. Performance of CosmoFlow (per node sample/s) using our decoder plugin compared with the baseline and gzip compression, using the small set (128 samples/GPU). Our plugin achieves up to 8× speedup compared with the base, while conventional compression results in up to 50%.

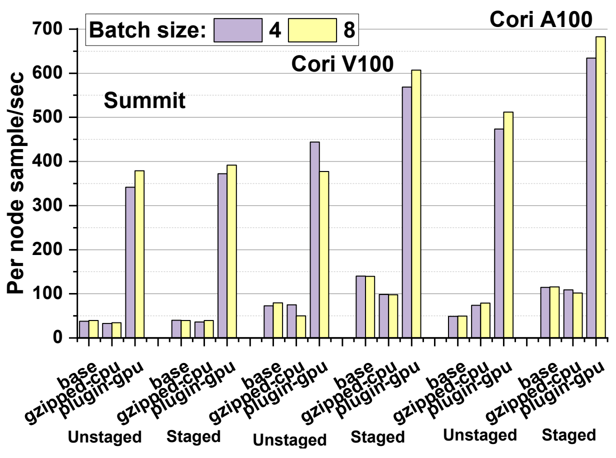


Fig. 11. Improvement of throughput of CosmoFlow (per node sample/s) using a large dataset (2048 samples per GPU) that does not fit in memory using the decoder plugin vs. base and gzip compressed samples on three platforms.

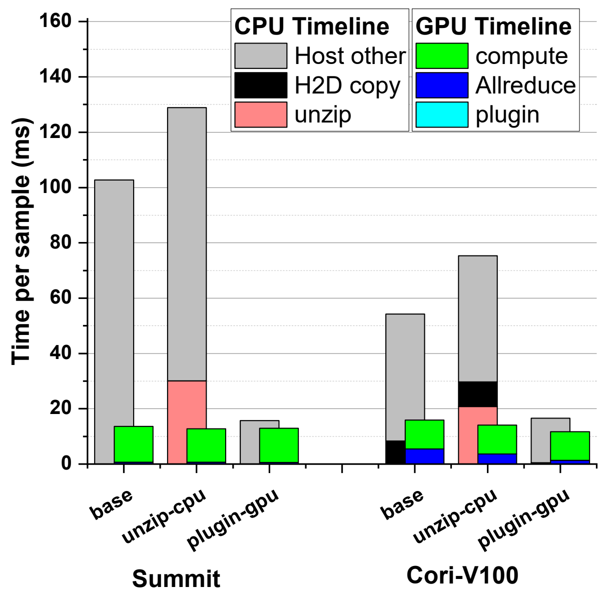


Fig. 12. CosmoFlow execution time breakdown for the small sample set with batch size of four. The base version underutilizes the GPU, while our plugin reduces host CPU preprocessing overhead.

pression. We also leverage the application convergence under mixed-precision processing to achieve a higher compression ratio. Additionally, our study presents detailed experiments that examine a variety of critical parameters, including data and batch size, and the impact of staging on node-local memory to explore system architectures beyond those investigated in this study. The developed custom encoder leveraged the reordering of computation with decompression to apply operators on on unique values of the data, and also the fusion of data transpose with decompression thus achieving higher efficiency for preparing the data for computation.

Overall our detailed analysis of CPU and GPU behavior, shows performance gains of up to 10× relative to the baseline approach. Comparison with traditional gzip compression results in an execution slowdown, highlighting the limitations of using off-the-shelf solutions for complex machine learning workloads and underlying architectures.

We believe that our approach can be used as a template to optimize a wide variety of SciML codes and can be incorporated into a variety of emerging architectural accelerators. Future work will explore other classes of scientific workloads and their datasets as well as emerging system architectures, with a focus on specialized deep learning accelerators.

## Acknowledgements

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231 and used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and the

Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## References

[1] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," *CoRR*, vol. abs/1803.04014, 2018.

[2] N. P. e. a. Jouppi, "In-datacenter performance analysis of a tensor processing unit," ISCA '17, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2017.

[3] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore IPU architecture via microbenchmarking," *CoRR*, vol. abs/1912.03413, 2019.

[4] Cerebras, "The wafer-scale engine (wse)." https://cerebras.net, 2021.

[5] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, "Accelerating scientific applications with sambanova reconfigurable dataflow architecture," *Computing in Science and Engineering*, vol. 23, 3 2021.

[6] "MLPerf Training: HPC." https://mlcommons.org/en/training-hpc-07/, 2021.

[7] "Cosmoflow tensorflow keras benchmark implementation." https://github.com/sparticlesteve/cosmoflow-benchmark, 2021.

[8] "Deep learning climate segmentation benchmark." https://github.com/sparticlesteve/mlperf-deepcam, 2021.

[9] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," 2 2019.

[10] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, A. López García, I. Heredia, P. Malík, and L. Hluch?, "Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey," *Artif. Intell. Rev.*, vol. 52, p. 77–124, June 2019.

[11] "Dawnbench: An end-to-end deep learning benchmark and competition." https://databricks.com/research/dawnbench-an-end-to-end-deep-learning-benchmark-and-competition, 2021.

[12] J. Dongarra, P. Luszczek, and Y. Tsai, "Hpl-ai mixed-precision benchmark." https://icl.bitbucket.io/hpl-ai/, 2021.

[13] P. e. a. Mattson, "MLPerf Training Benchmark," in *Proceedings of Machine Learning and Systems* (I. Dhillon, D. Papailiopoulos, and V. Sze, eds.), vol. 2, pp. 336–349, 2020.

[14] "Deepbench." https://github.com/baidu-research/DeepBench, 2021.

[15] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 66–77, 2019.

[16] 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," *CoRR*, vol. abs/2007.00279, 2020.

[17] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure, "Characterizing deep-learning i/o workloads in tensorflow," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pp. 54–63, 2018.

[18] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia, "Characterizing deep learning training workloads on alibaba-pai," *CoRR*, vol. abs/1910.05930, 2019.

[19] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. V. Essen, "The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism," *IEEE Transactions on Parallel & Distributed Systems*, vol. 32, no. 07, pp. 1641–1652, 2021.

[20] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 235–245, 2019.

[21] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "Dlio: A data-centric benchmark for scientific deep learning applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 81–91, 2021.

[22] K. Z. Ibrahim, T. Nguyen, H. A. Nam, W. Bhimji, S. Farrell, L. Oliker, M. Rowan, N. J. Wright, and S. Williams, "Architectural requirements for deep learning workloads in hpc environments," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 7–17, 2021.

[23] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," 2021.

[24] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[25] D. M. Hammerling, A. H. Baker, A. Pinard, and P. Lindstrom, "A collaborative effort to improve lossy compression methods for climate data," in *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)*, pp. 16–22, 2019.

[26] A. Fox, J. Diffenderfer, J. Hittinger, G. Sanders, and P. Lindstrom, "Stability analysis of inline zfp compression for floating-point data in iterative methods," *SIAM Journal on Scientific Computing*, vol. 42, no. 5, pp. A2701–A2730, 2020.

[27] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a gpu," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), Association for Computing Machinery, 2011.

[28] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, IEEE Press, 2018.

[29] L. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," *CoRR*, vol. abs/1802.02611, 2018.

[30] "Ncar community atmosphere model (cam 5.0)." https://www.cesm.ucar.edu/models/cesm1.0/cam/docs/description/cam5_desc.pdf, 2021.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[32] M. A. et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[33] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow.," *CoRR*, vol. abs/1802.05799, 2018.

[34] "Cosmoflow datasets." https://portal.nersc.gov/project/m3363/, 2021.

[35] "Tfrecord and tf.train.example." https://www.tensorflow.org/tutorials/load_data/tfrecord, 2021.

[36] "Tfrecord compression." https://www.tensorflow.org/api_docs/python/tf/io/TFRecordOptions, 2021.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[38] G. K. Wallace, "The jpeg still picture compression standard," *Commun. ACM*, vol. 34, p. 30–44, Apr. 1991.

[39] "nvjpeg: Gpu-accelerated jpeg decoder, encoder and transcoder." https://developer.nvidia.com/nvjpeg, 2021.

[40] Mathuriya *et al.*, "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 '18, IEEE Press, 2018.

[41] S. Tassev, M. Zaldarriaga, and D. J. Eisenstein, "Solving large scale structure in ten easy steps with cola," *Journal of Cosmology and Astroparticle Physics*, vol. 2013, p. 036–036, Jun 2013.

[42] O. Hahn and T. Abel, "Multi-scale initial conditions for cosmological simulations," *Mon.Not.Roy.Astron.Soc.415:2101-2121,2011*, vol. 415, 11 2011.

[43] "Nvidia data loading library (dali)." https://docs.nvidia.com/deeplearning/dali/, 2021.

[44] Vazhkudaiverman *et al.*, "The design, deployment, and evaluation of the coral pre-exascale systems," 7 2018.

[45] "Cori gpu nodes." https://docs-dev.nersc.gov/cgpu/, 2021.