Asynchronous I/O Strategy for Large-Scale Deep Learning Applications

Sunwoo Lee*, Qiao Kang*, Kewei Wang*, Jan Balewski‡, Alex Sim†,
Ankit Agrawal*, Alok Choudhary*, Peter Nugent†, Kesheng Wu†, and Wei-keng Liao*

*ECE Department, Northwestern University
{slz839, qkt561, kwf5687, ankitag, choudhar, wkliao}@ece.northwestern.edu
†Lawrence Berkeley National Laboratory
{asim, penugent, kwu}@lbl.gov
‡ National Energy Research Scientific Computing Center
balewski@lbl.gov

***Abstract*—Recently, many scientific applications adopt deep learning methods to solve their classification or regression problems. However, for data-intensive scientific applications, I/O performance can be the major performance bottleneck. In order to effectively solve important real-world problems using deep learning methods on High-Performance Computing (HPC) systems, it is essential to address the poor I/O performance issue in parallel neural network training. In this paper, we propose an asynchronous I/O strategy that can be generally applied to deep learning applications. Our I/O strategy employs an I/O-dedicated thread per process, that performs I/O operations independently of the training progress. The I/O thread reads many training samples at once to reduce the total number of I/O operations per epoch. Given the fixed amount of training data, the fewer the I/O operations per epoch, the shorter the overall I/O time. The I/O operations are also overlapped with the computations using the double-buffering method. We evaluate our I/O strategy using two real-world scientific applications, CosmoFlow and Neuron-Inverter. Our experimental results demonstrate that the proposed I/O strategy significantly improves the scaling performance without affecting the regression performance.**

***Index Terms*—I/O, Deep Learning, Parallelization**

I. Introduction

Recently, a variety of scientific applications adopt deep learning methods to solve their classification/regression problems [1]–[4]. However, training deep and large networks is an extremely time-consuming task that can take hours or even days. Especially for the data-intensive scientific applications, efficient scaling of training is critical to fully utilize High-Performance Computing (HPC) platforms and effectively tackle large-scale problems.

In deep learning-based scientific applications, I/O time can take up a large portion of training time. When collecting experimental data, they are usually stored as a few large files such that each file contains many data samples. Thus, in data parallel training, many processes can access the same file simultaneously causing I/O congestions. In addition, it is a common practice that the training samples are randomly shuffled during training, which results in having many small random accesses. The expensive I/O cost implies that the compute resources are waiting for the data to be ready staying idle. Thus, in order to achieve a good scaling efficiency, it is essential to minimize the I/O cost. However, while the statistical efficiency of training algorithms and the communication cost in parallel training have been widely studied, the I/O performance is overlooked and has not been well studied.

Several deep learning applications have acknowledged that the parallel training of their networks suffer from the expensive I/O cost [3], [5]–[8]. The researchers have put much effort into improving I/O performance as follows. Mathuriya et al. prefetched the training dataset into *Burst Buffer* so that the mini-batches are rapidly read from the SSD-based storage servers [3]. Zhu et al. designed I/O pipelining for TensorFlow [9], which takes advantage of Remote Direct Memory Access (RDMA) for data shuffling [8]. Pumma et al. improved the I/O performance of Caffe [10] by re-designing the LMDB I/O library [6]. Although these works effectively improve the I/O performance, they either rely on special hardware-assisted features or tackles the performance issues existing in a specific software library.

In this paper, we propose an asynchronous I/O strategy that can be generally applied to data parallel neural network training. Our study specifically focuses on the I/O performance of loading data from disk space to CPU memory space. First, to minimize the number of I/O operations, our strategy allocates a large memory buffer and reads many training samples at once. Given a fixed dataset size, the fewer the I/O operations, the shorter the overall I/O time. In addition, we adopt double buffering method to overlap the I/O time with the training time. We implement the I/O overlap by employing an I/O-dedicated thread per process, which fills in the two I/O buffers one after another. By adjusting the I/O buffer size, users can make a good trade-off between the I/O performance and the memory footprint. Our I/O strategy also affects the degree of randomness in data shuffling depending on the I/O buffer size. We will discuss our empirical study of such an impact on the regression performance.

We evaluate our proposed I/O strategy using two real-world scientific applications, CosmoFlow and Neuron-Inverter. Both applications solve domain-specific regression problems using

a large amount of experimental data that cause expensive I/O cost during training. We report and analyze the scaling performance using two different supercomputers, Summit at Oak Ridge National Laboratory (ORNL) and Cori at National Energy Research Scientific Computing (NERSC). Our experimental results demonstrate that the I/O time can be effectively reduced by applying our proposed I/O strategy to the large-scale deep learning applications.

II. Background

A. Convolutional Neural Networks

Convolutional Neural Network (CNN) is a type of artificial neural network that contains convolution layers [11]. The convolution layers have a special connection pattern such that each neuron at one layer is connected to a subset of the neurons at the previous layer. Such connection pattern enables exploitation of spatially-local correlation in the input data. Each convolution layer can be followed by a pooling layer. Depending on the model architecture, the networks can have a few fully-connected layers at the end of the model.

CNNs are popularly used to solve computer vision or natural language processing problems. Many scientific applications also employ CNNs when the input data is known to inherently have spatially-local correlation. In this work, we study two real-world scientific applications that solve the domain-specific regression problems using deep CNNs.

B. Training Algorithms with Data Shuffling

The most popular training algorithm for neural networks is mini-batch Stochastic Gradient Descent (SGD) and its variants such as Adam [12], AdaGrad [13], and AdaDelta [14]. We will call mini-batch version of SGD 'SGD' for short. SGD iteratively updates the model parameters using gradients approximated from a random subset of training samples (called mini-batch). The algorithm stops the training when either the gradients become sufficiently small or the achieved accuracy becomes acceptable by users.

In deep learning applications, 'epoch' is usually defined as iterations for traversing over all the given training samples once. It is a common practice to shuffled the training samples every epoch so that the mini-batches consist of differen training samples. Approximating gradients from different mini-batches can be considered as inherently injecting noise to the training, and it likely results in achieving better generalization performance. However, such a random data access pattern can cause an expensive I/O cost. Especially for scientific applications with large-scale experimental data, the global data shuffling can cause an extremely expensive I/O cost.

C. Synchronous SGD with Data Parallelism

Synchronous SGD with data parallelism is the most popular parallelization strategy for deep learning applications. In data parallel training, each mini-batch is evenly distributed to all workers and they are independently processed. Once all the workers locally compute the gradients from the given training samples, the gradients are averaged across all the workers using inter-process communications. Typically, the gradients are averaged using *allreduce* communications. If the gradients are averaged every iteration, all the workers can update the model parameters always using the globally synchronized gradients. This approach is called 'synchronous' SGD. Although there are many alternative parallel training algorithms, such as asynchronous SGD or local SGD, we focus on synchronous SGD with data parallelism considering its popularity and effectiveness on achieving the high accuracy.

D. CosmoFlow

CosmoFlow is a deep learning tool for Cosmology data analysis, developed by Lawrence Berkeley National Laboratory and Intel. Given a 3-dimensional distribution of masses in the evolved Universe, CosmoFlow estimates the initial condition of the Universe. Mathuriya et al. proposed a 3-D CNN solution to this large-scale multi-value regression problem and studied the scaling performance on Cori KNL nodes [3]. CosmoFlow is incorporated in the MLPerf HPC benchmark suite - an industry standard for measuring machine learning performance on large-scale HPC systems [15]. The most computationally challenging aspect of the CosmoFlow is ingestion of the input 3D cubes of size from 128^3 up to 1024^3, which can easily exceed the memory space available on GPU.

E. Neuron-Inverter

Neurons are the fundamental units of computation in brain. Their electrical properties arise from the spatial densities of the diverse ion channels along the membrane. Neuron-Inverter is a project to develop a deep learning tool for inferring such channel density values from empirical recordings of single neurons. It will allow to construct realistic biophysical neuronal models and give insights into the etiology of neurological diseases such as Autism and Epilepsy [16], [17]. Ben-Shalom at al. demonstrated use of 1-D CNN to regress the time series data of neuron action potential to ion channels densities [16]. For Neuron-Inverter, the data samples are relatively small, but the dataset contains an enormous number of samples ($O(10^8)$) causing extremely I/O intensive neural network training.

III. Asynchronous I/O Strategy for Data-Parallel Training

In this section, we describe our I/O strategy for data-parallel neural network training. We begin with a description of our I/O strategy that enables large contiguous read operations. Then, we explain how to overlap the I/O time with the computation time by employing double-buffering method. We will use a few notations as follows: N is the total number of training samples, B is the I/O buffer size with respect to the number of samples, M is the mini-batch size, and P is the number of processes.

A. Asynchronous I/O Strategy

1) I/O pattern in training: Before we discuss our proposed I/O strategy, we define the I/O pattern in neural network

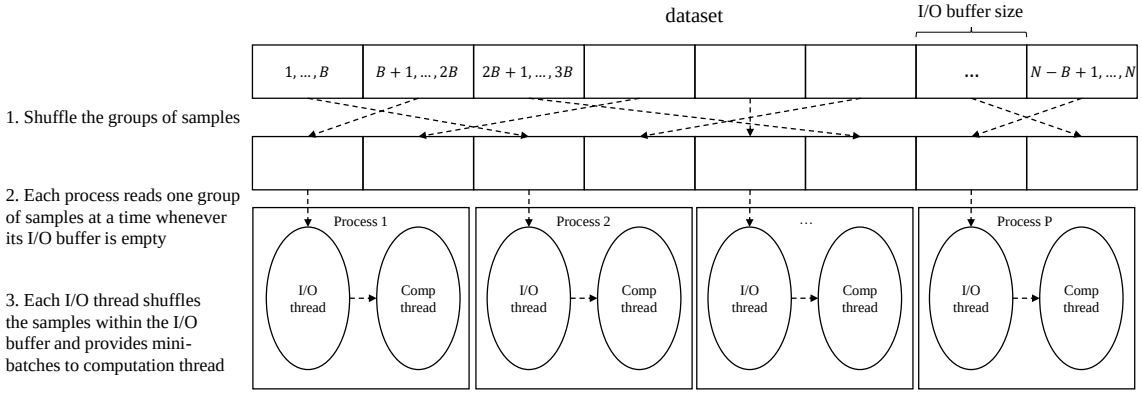

Fig. 1. An example illustration of the proposed I/O strategy for deep learning. Each process has an I/O dedicated thread. First, the dataset is partitioned to $\frac{N}{B}$ groups and randomly shuffled, where N is the number of samples and B is the I/O buffer size per process. Second, each process is assigned with $\frac{N}{BP}$ groups, where P is the number of processes. Then, during the training, each process reads one group of samples at a time using an I/O-dedicated thread.

training. Given N training samples, M samples are extracted as a mini-batch and processed one batch after another. Thus, at each epoch, $\frac{N}{M}$ mini-batches are read from the dataset in total. The training is typically performed for multiple epochs until the training loss converges. Therefore, the described I/O operations are repeatedly perform until the end of the training. In data parallel training, each mini-batch is evenly distributed to all P processes. So, each process reads $\frac{M}{P}$ samples per iteration. As scaling up, the number of iterations per epoch is fixed to $\frac{N}{M}$ regardless of the number of processes.

In neural network training, the data samples are usually shuffled every epoch. Each mini-batch most likely consists of different samples every iteration. So, when a mini-batch is extracted from the dataset, we assume random M samples are read from the files. During the parallel training, each process reads random $\frac{M}{P}$ samples at each iteration and repeats this $\frac{N}{M}$ times per epoch.

2) Asynchronous I/O with I/O-dedicated thread: In many scientific applications, the experimental data is usually stored as a small number of large files. This data organization allows researchers to better manage, transfer, and analyze the data. However, considering the I/O pattern in neural network training, this approach makes multiple processes access a single file at the same time causing I/O congestion. In addition, the data samples are assumed to be shuffled in neural network training, and it makes the random data access pattern. Such random small I/O accesses from multiple processes can cause a significantly expensive I/O cost.

To efficiently perform I/O operations in parallel neural network training, we propose to asynchronously read a large number of samples at once. Figure 1 presents our I/O strategy. Instead of globally shuffling all the individual data samples, the dataset is partitioned to $\frac{N}{B}$ groups and they are shuffled every epoch. Then, each process is assigned with random $\frac{N}{BP}$ groups of samples and extract the local mini-batches from the given groups. For each local mini-batch, the data samples are shuffled again inside of the I/O buffer.

The described I/O strategy has two advantages as follows. First, a large contiguous region of a file can be read at once. If each file contains more than B samples, the entire group can be read by a single read operation. Given the same amount of total data, a cheaper I/O cost can be expected by having fewer I/O operations. Second, the number of processes that access the same file is dramatically reduced. Since our strategy enforces each process to read the samples only from the assigned groups, up to $\frac{K}{B}$ processes may access the file, where K is the number of samples in the file. If all the individual samples are globally shuffled, up to $\max(K, P)$ processes can access the same file. By reducing the number of processes per file, the read operations can more effectively take advantage of the cache effect. In practice, most of the large-scale HPC systems allow to have the I/O buffers that are large enough to read one file at a time achieving the optimal I/O performance.

The proposed I/O strategy makes a trade-off between the I/O cost and the memory footprint. By loading a large amount of data at once, the dataset can be processed with fewer read operations, and thus the overall I/O time is likely reduced. However, each process should consume a large amount of memory space for the I/O buffer. The larger the buffer size, the fewer the I/O operations. The modern HPC platforms usually have a large amount of memory space in each node. For instance, each GPU node of Summit supercomputer at Oak Ridge National Laboratory has 512 GB memory space. Considering such a rich memory space, the extra memory consumption of our I/O strategy can be justified in practice.

Note that the proposed I/O strategy affects the data shuffling. Since the groups of samples are shuffled across the processes, instead of the individual samples, the degree of randomness is sacrificed compared to the global shuffling. At the worst case, if the number of groups is the same as the number of processes, the proposed strategy becomes the local shuffling. In this study, we empirically found that the regression performance was not degraded even at the worst case. The I/O buffer sizes decides the number of groups $\frac{N}{B}$. In the evaluation section, we will report and analyze the impact of the buffer size on the I/O performance as well as the regression performance.

B. Double-Buffering for I/O Overlap

When training a neural network with SGD, batches of data samples are processed one after another. Thus, while one mini-batch is being processed, the next mini-batch can be pre-loaded without having data dependency. In this way, the SGD iteration time can be reduced by overlapping the I/O time with the computation and communication time in parallel training. TensorFlow, one of the most popular deep learning software frameworks, also supports data prefetching feature that enables I/O overlapping. However, if the data sample size is small, reading one batch at a time may cause a large number of random small read operations.

We propose to use double-buffering to effectively overlap the I/O operations with the computations, based on the asynchronous I/O strategy described in Section III-A2. First, each process allocates two I/O buffers. Then, the I/O thread of the process monitors the two buffers. If any of the them is empty, the I/O thread reads one group of data samples to fill in the buffer. During the whole training, the I/O thread asynchronously monitors and fills in the empty buffers. The computation thread consumes the data in one buffer after another. For each mini-batch, each computation thread reads random $\frac{M}{P}$ samples from the current buffer. Once all the samples are extracted from one buffer, the computation thread switches to the other buffer. If both buffers are empty, the computation thread waits until the I/O thread fills in one buffer with new data.

Figure 2 presents the described asynchronous double-buffering for I/O overlapping. Once the computation thread starts to consume the data from one buffer, it takes $\frac{BP}{M}$ iterations to use all the data in the buffer. If the I/O buffer size B is larger than the local batch size $\frac{M}{P}$, one read operation can be overlapped with the computations for processing multiple mini-batches. In the ideal case, if the read operation takes a shorter amount of time than the computation time for $\frac{BP}{M}$ iterations, the entire I/O time can be hidden behind the computation time.

The double buffering method increases the memory footprint. While the computation thread consumes the data from one buffer, the I/O thread fills in the other buffer simultaneously. So, each process requires to allocate two memory buffers each of which can hold B data samples. Typically, it is a common practice for deep learning applications that one process is assigned on one GPU so that the process fully utilizes the given GPU resources including the GPU memory space. In modern HPC systems, each compute node usually contains $4 \sim 8$ GPUs. Thus, the proposed I/O strategy can increase the memory footprint by $8B \sim 16B$. We suggest maximizing the buffer size considering the available memory space in the system to minimize the number of read operations. In Section IV, we will analyze the impact of the I/O buffer size on the overall performance.

C. Implementation Details

Most of the popular deep learning software frameworks such as TensorFlow or PyTorch support Python programming environment. In this work, we also implemented the deep learning solutions for CosmoFlow and Neuron-Inverter using TensorFlow using Python language. The proposed asynchronous I/O strategy requires one I/O thread per process. However, Python *threading* package does not actually support the instruction-level parallelism due to the Global Interpreter Lock (GIL). In order to avoid such a limitation, we used Python *multiprocessing* package that implements the shared-memory programming model using processes. At the initialization time, each MPI process creates a 'thread' using *multiprocessing* package and two memory buffers shared between them. The main thread extracts mini-batches from the shared memory buffers while the child thread asynchronously fills in the two buffers reading new data samples. Note that our implementation is based only on the off-the-shelf Python packages without using any deep learning framework-dependent software features. Our implementation also does not rely on any hardware-assisted features.

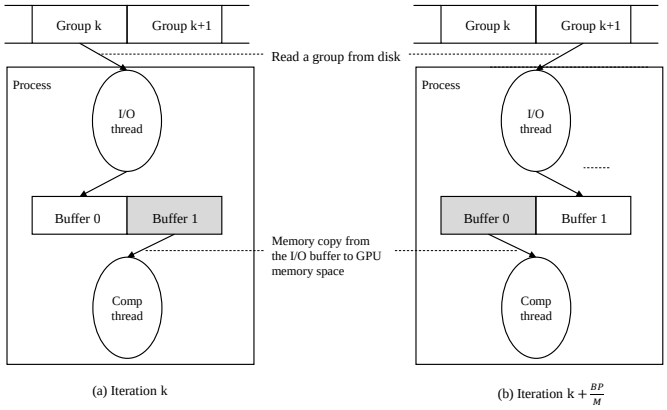

Fig. 2. Double-buffering for I/O overlapping. While the I/O thread fills in one buffer, the computation thread consumes the data in the other buffer. The producer (I/O thread) asynchronously fills in any buffer that is empty. The consumer (Computation thread) extracts $\frac{M}{P}$ random samples from the buffer as a local mini-batch, and switches the buffer once the current buffer is all consumed.

IV. Evaluation

In this section, we evaluate the proposed asynchronous I/O strategy using two scientific applications, CosmoFlow and Neuron-Inverter.

A. Experimental Settings

Systems – We use two different HPC platforms for the experiments, Cori GPU machines and Summit. Cori is a Cray XC40 supercomputer at National Energy Research Scientific Computing Center (NERSC). We use Cori GPU machines [18] that consists of 18 nodes. Each node has two sockets of Intel Xeon Gold 6148 (Skylake) CPUs, 8 NVIDIA V100 GPUs and 384 GB memory space. Summit is an IBM AC922 system that consists of 4,608 nodes [19]. Each node has two sockets of IBM Power9 CPUs, 6 NVIDIA V100 GPUs, and 512 GB memory space.

TABLE I
FOUR I/O STRATEGIES STUDIED IN THIS PAPER.

	Prefetch	I/O pattern	I/O overlap
TF	no	one sample per read	no
TF-Pre	yes	one sample per read	TF thread pool
Async I/O	no	multiple samples per read	I/O-dedicated thread
Async I/O (DB)	yes	multiple samples per read	I/O-dedicated thread

Software – On Cori, we used TensorFlow 2.2.0 and Horovod 0.19.0 for all the experiments. On Summit, we used IBM Watson Machine Learning Community Edition 1.7.0-3 that supports TensorFlow 2.1.0 and Horovod 0.19.0. For the I/O-dedicated thread, we used Python *multiprocessing* package. The source code of CosmoFlow[1] and Neuron-Inverter[2] will be released once this paper is accepted.

Datasets – CosmoFlow [3] is a large-scale Cosmology parameters regression problem. The training data for the CosmoFlow are simulated mass distributions of the Universe for different initial conditions. The *4parE* dataset was generated with four different initial condition parameters uniformly varied by 10% around their nominal values. The range of the 4 varied parameters was scaled to be within (-1,1). The same initial condition Universe was evolved into 4 different redshifts. The Universes were binned into cubes with 512 bins in all 3 dimensions. Then, each Universe is reshaped into a 12-channel cube of size $128 \times 128 \times 128$ by concatenating subsets of the original cube on the channel dimension. So, each sample size is $128 \times 128 \times 128 \times 12$ and the label size is 4. The data were packed as 5-dimensional Numpy arrays of dtype unit16 and stored in HDF5 files. The dataset consists of 64 HDF5 files in total and each file contains 128 samples. The overall training data size is ~ 384 GB.

Neuron-Inverter is another regression problem that estimates the input-output neuronal mechanism [20]. The *Ontra2* dataset consists of 101 files each which contains $\sim 610K$ data samples. The input data is neuron spikes measured at 3 different locations as 1-dimensional time series of size 1600. The output data is 19 electrical properties (conductances) determined for different compartments of neuron. So, each sample size is 1600×3 and the corresponding label size is 19. All the samples from each cell is stored as a single HDF5 file. Given 101 files, we used 64 files for training. Each data point is a 4-byte floating point numbers and the overall training data size is ~ 680 GB.

Parallel File System Settings – On Cori, the input files are stored on Lustre parallel file system. For each file, the stripe size is set to 1 MB and the stripe count is set to 1. On Summit, the input files are located on IBM Spectrum Scale parallel file system called Alpine. Alpine does not enable users to adjust the stripe settings.

Neural Networks – For CosmoFlow, we used a slightly modified version of Livermore Big Artificial Neural Network (LBANN) [21]. The network has 7 3-D convolution layers followed by 3 fully-connected layers. The overall number of parameters is 9.4 millions. For Neuron-inverter, we designed a 1-D CNN that consists of 4 1-D convolution layers followed by 5 fully-connected layers. The network has 3.2 millions parameters in total. Both networks are deep and large CNNs designed to solve the application-specific regression problems. The detailed model architecture can be found in the open-source that will be opened once the paper is accepted.

[1]https://github.com/NU-CUCIS/tf2-cosmoflow
[2]https://github.com/NU-CUCIS/tf2-neuroninverter

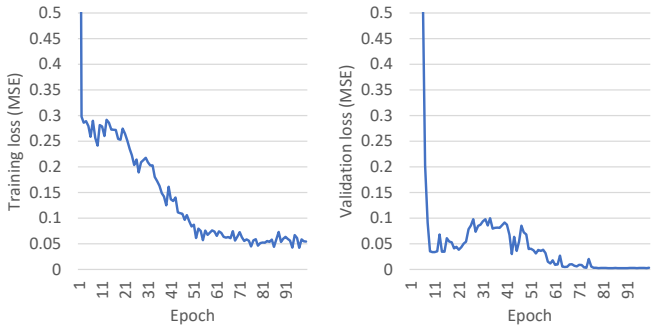

Fig. 3. The learning curves of CosmoFlow. The global batch size is 256 and the learning rate is 0.002. We used Adam optimizer and the training is performed for 100 epochs. Using Mean Squared Error (MSE) metric, the achieved validation loss is 0.002344.

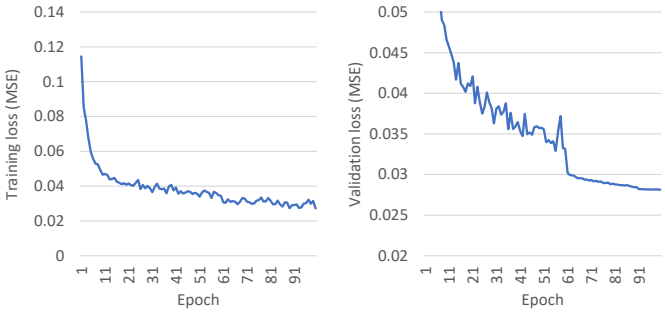

Fig. 4. The learning curves of Neuron-Inverter. The global batch size is 32,768 and the learning rate is 0.0005. We used Adam optimizer and the training is performed for 100 epochs. Using Mean Squared Error (MSE) metric, the achieved validation loss is 0.028126.

I/O Strategies – We compare four different I/O strategies that are summarized in Table I. *TF* is the baseline that uses `tf.data` API. It reads one training sample at a time without data prefetching. This setting exposes the entire I/O time. *TF-Pre* uses `tf.data` API with prefetching feature. The prefetching is adopted following the suggestions in TensorFlow official guideline [22]. We used `AUTOTUNE` option supported by TensorFlow. *Async I/O* is the proposed asynchronous I/O strategy without double buffering. Similarly to the baseline, all the I/O time is exposed. *Async I/O (DB)* is the proposed asynchronous I/O strategy with double buffering.

B. Regression Performance

We first report the regression results achieved with the best-tuned hyper-parameter settings. Although this paper focuses on the I/O performance of parallel neural network training,

TABLE II
THE AVERAGE EPOCH TIMING BREAKDOWN FOR COSMOFLOW ON CORI GPU MACHINES. THE TIMINGS ARE ALL IN SECOND.

Number of GPUs	I/O strategy	Exposed I/O	Comp	Comm
32	TF	46.03	38.33	0.10
	TF-Pre	28.91	38.31	0.11
	Async I/O	43.18	38.01	0.11
	Async I/O (DB)	10.71	37.95	0.10
64	TF	27.65	19.30	0.11
	TF-Pre	14.39	19.32	0.12
	Async I/O	21.82	19.84	0.12
	Async I/O (DB)	4.00	19.13	0.11

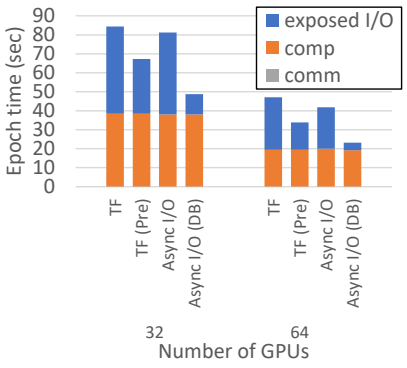

Fig. 5. Strong scaling performance of CosmoFlow on Cori GPU machines. We measured the average epoch time. The timing breakdown shows the exposed I/O time, computation time, and communication time, separately. The timings are averaged across 3 epochs. The model does not fit to fewer than 32 GPUs. We scale the training up to the case where each process works on one file. We see that the *Async I/O (DB)* has significantly reduced the exposed I/O time and outperforms the other settings.

TABLE III
THE AVERAGE EPOCH TIMING BREAKDOWN FOR COSMOFLOW ON SUMMIT. THE TIMINGS ARE ALL IN SECOND.

Number of GPUs	I/O strategy	Exposed I/O	Comp	Comm
32	TF	4.12	17.23	0.11
	TF-Pre	1.99	18.39	0.10
	Async I/O	2.29	18.25	0.10
	Async I/O (DB)	0.00	18.73	0.11
64	TF	3.46	9.10	0.14
	TF-Pre	1.13	9.59	0.15
	Async I/O	1.71	9.60	0.14
	Async I/O (DB)	0.00	9.67	0.15

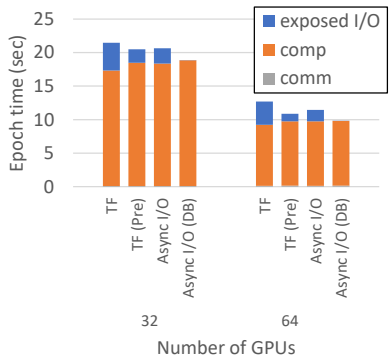

Fig. 6. Strong scaling performance of CosmoFlow on Summit. We measured the average epoch time. The timing breakdown shows the exposed I/O time, computation time, and communication time, separately. The timings are averaged across 3 epochs. The model does not fit to fewer than 32 GPUs. We scale the training up to the case where each process works on one file. We see that the *Async I/O (DB)* has near-zero I/O time, which demonstrates the effectiveness of the proposed I/O strategy.

we present these regression results to provide useful insights to the deep learning users and the domain scientists.

1) CosmoFlow: We trained LBANN model on 4ParE dataset for 100 epochs using Adam optimizer. The global batch size is 256 and the initial learning rate is 0.002. The learning rate is decayed by a factor of 10 twice, after 50 epochs and 75 epochs. The loss function is Mean Squared Error (MSE). Figure 3 presents the training loss (left) and the validation loss (right). With the best-tuned hyper-parameters, we could achieve the validation loss of 0.002344.

2) Neuron-Inverter: We trained the 1-D CNN model we designed on Ontra dataset for 100 epochs using Adam optimizer. The global batch size is 32,768 and the initial learning rate is 0.0005. The learning rate is decayed by a factor of 10 twice, after 60 epochs and 90 epochs. Figure 4 presents the training loss (left) and the validation loss (right). We achieved the validation loss of 0.028126. Note that, since each file has different numbers of samples, some processes oversample the data when they have smaller files than the other processes. So, the number of processed samples can be slightly larger than the number of the actual samples.

C. CosmoFlow Scaling Performance

We present the strong scaling performance of CosmoFlow on Cori and Summit and then analyze the I/O performance. In all the charts, we report 'exposed I/O time' rather than the total I/O time. Since the I/O operations are asynchronously performed by a separate thread, the directly measured I/O time may contain unexpected overhead such as context switching time. So, we measure the waiting time of the main thread instead and consider it as the exposed I/O time.

1) Cori GPU Nodes: Table II and Figure 5 present the strong scaling performance of LBANN training on Cori GPU nodes. Note that the model does not fit to the memory space when running on fewer than 32 GPUs. Given 64 HDF5 training files, we set the buffer size to 128 samples (a single file size), and thus the training can scale up to 64 GPUs. First, we see that *TF*'s I/O time takes up a larger portion than the computation time within each epoch. *TF-Pre* reduces the exposed I/O time by preloading the data in background, however, most of the I/O time is still exposed. *Async I/O*'s I/O time is slightly shorter than that of *TF* thanks to the reduced number of I/O operations. With the doulbe buffering, *Async I/O (DB)* effectively reduces the exposed I/O time and achieves the shortest epoch time among all the I/O strategies. When using 64 GPUs, *Async I/O (DB)* shows a significantly reduced epoch time (23.25 sec) compared to that of *TF-Pre* (33.84 sec).

2) Summit GPU Nodes: We perform the same CosmoFlow scaling experiments on Summit. Table III and Figure 6 present

TABLE IV
THE AVERAGE EPOCH TIMING BREAKDOWN FOR NEURON-INVERTER ON CORI GPU MACHINES. THE TIMINGS ARE ALL IN SECOND.

Number of GPUs	I/O strategy	Exposed I/O	Comp	Comm
16	TF	53384.49	390.93	4.35
	TF-Pre	53100.29	388.48	4.24
	Async I/O	199.99	387.28	4.05
	Async I/O (DB)	0.00	390.93	4.94
32	TF	25899.36	201.29	2.93
	TF-Pre	25703.93	203.42	3.09
	Async I/O	113.42	198.82	2.48
	Async I/O (DB)	0.00	200.19	3.13
64	TF	13798.48	97.05	1.99
	TF-Pre	13664.83	96.05	2.24
	Async I/O	77.95	97.13	2.83
	Async I/O (DB)	0.00	98.99	2.19

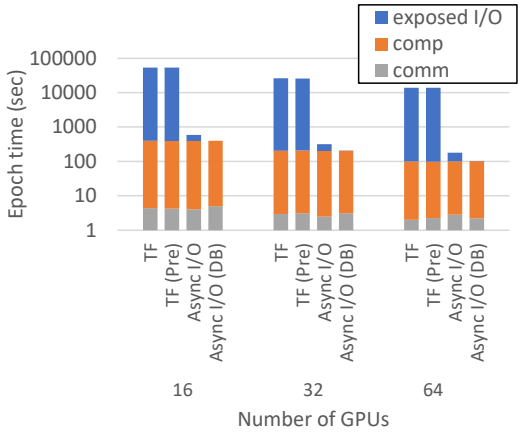

Fig. 7. Strong scaling performance of Neuron-Inverter on Cori GPU machines. The y-axis is epoch time in log scale. The timing breakdown shows the exposed I/O time, computation time, and communication time, separately. The timings are averaged across 3 epochs. The model does not fit to fewer than 16 GPUs. The proposed asynchronous I/O strategy dramatically reduces the I/O time.

TABLE V
THE AVERAGE EPOCH TIMING BREAKDOWN FOR NEURON-INVERTER ON SUMMIT. THE TIMINGS ARE ALL IN SECOND.

Number of GPUs	I/O strategy	Exposed I/O	Comp	Comm
16	TF	45677.48	765.38	2.03
	TF-Pre	40259.47	777.39	2.00
	Async I/O	54.96	764.44	2.13
	Async I/O (DB)	0.00	775.94	2.39
32	TF	24499.22	400.33	2.39
	TF-Pre	22095.84	403.94	3.39
	Async I/O	26.23	399.93	2.16
	Async I/O (DB)	0.00	405.01	2.19
64	TF	10820.49	215.59	4.20
	TF-Pre	10660.43	217.02	4.03
	Async I/O	14.52	215.32	3.95
	Async I/O (DB)	0.00	215.30	4.23

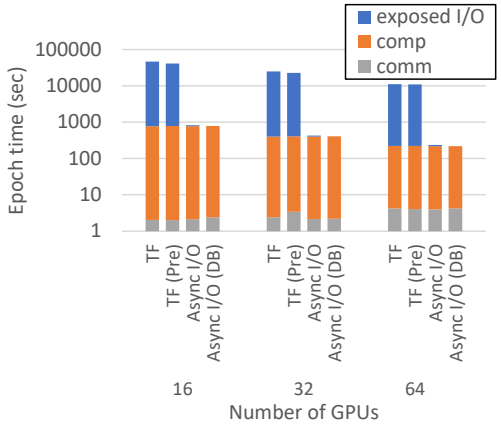

Fig. 8. Strong scaling performance of Neuron-Inverter on Summit. The y-axis is epoch time in log scale. The timing breakdown shows the exposed I/O time, computation time, and communication time, separately. The timings are averaged across 3 epochs. The model does not fit to fewer than 16 GPUs. The proposed asynchronous I/O strategy dramatically reduces the I/O time.

the scaling performance of CosmoFlow on Summit. Overall, the timing breakdown shows the similar performance results to that on Cori GPU nodes. Due to the different hardware configurations and environmental settings, the baseline (*TF*) I/O time is much shorter than that on Cori. While *TF-Pre* hides only a part of the I/O time behind the computation time, our proposed I/O strategy hides the entire I/O time and it results in having near-zero I/O time.

D. Neuron-Inverter Scaling Performance

We also study the I/O performance of Neuron-Inverter application. Compared to CosmoFlow, Neuron-Inverter dataset has a significantly smaller sample size. Thus, we can expect a different impact of I/O strategies on the performance.

1) Cori GPU Nodes: Table IV and Figure 7 present the scaling performance of Neuron-Inverter on Cori GPU machines. The model does not fit to fewer than 16 GPUs. Thus, we present the strong scaling performance from 16 processes (GPUs). First of all, the baseline takes an enormous amount of I/O time compared to the proposed I/O strategy. This huge difference of I/O time comes from the small random access pattern. As described in Section III-A1, each mini-batch consists of random M samples. The training sample size in Neuron-Inverter dataset is $\sim$ 19 KB. Given the file size of 10 GB $\sim$ 12 GB, reading such a small sample at a time can cause a significantly expensive I/O cost. Although TensorFlow's prefetching feature hides a part of the I/O time behind the computation time, the total I/O time is several orders of magnitude larger than the computation time, and thus the I/O overlap does not make a meaningful difference. In contrast, our proposed I/O strategy reads a large number of samples at once significantly reducing the number of I/O operations. In this experiment, we used the buffer size as the largest file size among all the given input files. So, each read operation prefetches about 610K training samples into the memory space at once. We see taht such a coarse-grained I/O operations considerably reduce the total I/O time.

2) Summit GPU Nodes: Neuron-Inverter scaling performance on Summit is similar to that on Cori. Table V and Figure 8 present the performance results on Summit. The timing breakdown shows that the I/O time takes up most of the epoch time in baseine (*TF*). Since the I/O time is several orders of magitude longer than the computation time, the

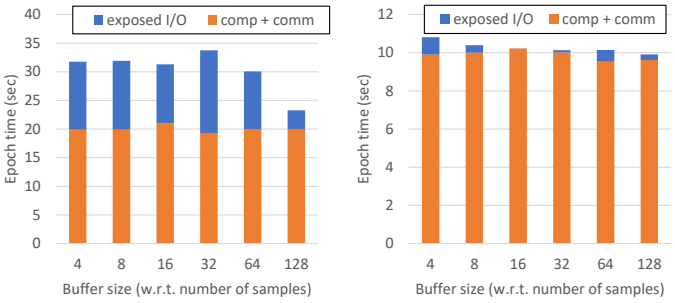

Fig. 9. CosmoFlow timing breakdown comparison with different I/O buffer sizes. The buffer size shown in x-axis is with respect to the number of samples. The left and right charts show the performance on Cori and Summit, respectively. On Cori, the exposed I/O time is effectively reduced when the buffer size is 128. On Summit, the exposed I/O time is already so small that there is not much room for improvement.

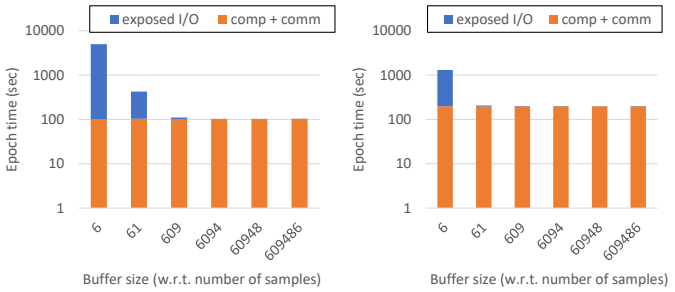

Fig. 10. Neuron-inverter timing breakdown comparison with different I/O buffer sizes. The maximum buffer size is decided by the number of samples in an input file ($\sim$ 610K). Then, we reduced it by a factor of 10. The left and right charts show the performance on Cori and Summit, respectively. The exposed I/O time is dramatically reduced as the buffer size increases. Cori requires at least the buffer size of 609 to entirely overlap the I/O time while Summit requires the buffer size of 61. Note that the y-axis is in log scale.

TensorFlow's prefetching feature does not make a meaningful difference. In contrast, our proposed asynchronous I/O strategy dramatically reduces the I/O time and the double buffering method hides it behind the computation time. So, *Async I/O (DB)* shows near-zero exposed I/O time. One notable thing is that the computation time is longer than that on Cori. Because Summit and Cori have different hardware configurations as well as software packages, the performance cannot directly compared between them. Remember that the computation time on Summit was much shorter than that on Cori for CosmoFlow, as shown in Table II and III. This result demonstrates that Summit can provide a better computational power when each sample size is sufficiently large.

E. Impact of Buffer Size

The proposed asynchronous I/O strategy allows to read more samples at once as the buffer size increases, and thus the total number of I/O operations is reduced. Given the same total data size, a cheaper I/O cost can be expected by performing fewer I/O operations. We analyze the impact of the I/O buffer size on both the I/O performance and the regression performance.

1) Impact on I/O Performance: We first report the impact of the I/O buffer size on the I/O performance for both applications. Figure 9 presents the impact of the I/O buffer size on the performance of CosmoFlow. The timing breakdowns are measured from CosmoFlow running on 64 processes (GPUs) with different buffer sizes. Since the global batch size is 256, the minimum buffer size is the local batch size, 4 samples. We set the buffer size up to 128 samples, which means a single file is read at once. Figure 10 shows the performance of Neuron-Inverter with different I/O buffer sizes. The y-axis of Figure 10 is in log scale. For Neuron-Inverter, the maximum buffer size is 609486 which is the maximum number of samples in a file. When reducing the buffer size, we scaled down it by a factor of 10. In both figures, the left chart shows the performance measured on Cori GPU nodes and right chart shows that on Summit.

We can get two insights from Figure 9 and 10. First, the buffer size should be large enough to minimize the latency overhead. Especially when the sample size is small, the latency overhead can cause an extremely expensive I/O cost. The Neuron-Inverter I/O time is several orders of magnitude larger than the computation time on both Cori and Summit. However, such a long I/O time is effectively reduced when the buffer size is larger than 609 samples (about 11 MiB). Second, if the I/O buffer is large enough to minimize the latency overhead, the overlapping plays a key role in improving the scalability. For example, CosmoFlow shows a minor improvement by increasing the buffer size on Cori because the sample size is already so large that the latency overhead does not take up a large portion of the epoch time. By applying the double buffering, most of the I/O time could be hidden behind the computation time and it resulted in achiving a considerably reduced epoch time. These experimental results demonstrate the effectiveness of the proposed asynchronous I/O strategy in large-scale deep learning applications.

2) Impact on Regression Performance: Recently, Meng et al. explained the impact of shuffling methods on the convergence rate of SGD training [23]. For non-convex optimization such as neural network training, the local shuffling does not harm the convergence rate if the following condition is satisfied.

$$S < \frac{n}{M}, \tag{1}$$

where S is the number of epochs, n is the total number of training samples, and M is the number of non-overlapped subsets of the training dataset. The number of training epochs for convergence is mostly affected by the training data, however, the condition shown above is most likely satisfied in scientific applications. In our experiments, we found that the above condition is most likely satisfied in both applications. For example, when scaling up CosmoFlow using 64 processes, the number of training samples n is 8192 and the minimum number of data groups is 64. So, if the training converges in fewer than $\frac{n}{M} = 128$ epochs, the local shuffling is not expected to degrade the regression performance. We found that the training loss converges in $80 \sim 100$ epochs with

the appropriate learning rate decay settings. For the Neuron-Inverter, the above condition also easily holds because of the large number of training samples. In practice, the above condition is most likely true especially for the large-scale scientific applications that have a large amount of experimental data.

F. Comparison with Previous Works

1) TensorFlow Prefetching: We have compared out proposed I/O strategy with TensorFlow's prefetching feature (*TF-Pre*) in all the experiments. TensorFlow's `tf.data.Dataset` module calls a user-defined callback function to read each mini-batch. The callback function is usually implemented such that it reads one training sample at a time. Given a batch size of B, the callback function is internally called B times to build up a single mini-batch. The prefetching feature enables to pre-load multiple samples or even mini-batches in advance. Although it allows the I/O overlapping, due to the I/O callback that reads one sample at a time, the underlying I/O operations become to have the expensive small random access pattern. So, this comparison demonstrates that the I/O granularity is important in data-intensive deep learning applications.

2) DeepIO: Zhu et al. proposed to employ double buffering method for overlapping the I/O time with the training time [8] similarly to our proposed I/O strategy. There are two primary differences between *DeepIO* and our I/O strategy.

Data Shuffling – First, the two I/O strategies shuffle the data samples in a different way. *DeepIO* globally shuffles the training samples only within the pre-loaded samples in the memory buffer. When a remote sample is required, the worker obtains it using an RDMA read operation. Our approach is generally applicable to any deep learning applications without using such a hardware-assisted feature. Our proposed I/O strategy first globally shuffles the contiguous groups of samples, and then the workers independently read the assigned groups. Although it causes random access pattern, each group is most likely large enough to avoid the small random access overhead. Then, each worker locally shuffles the samples again within its memory space. Our approach does not cause any extra inter-process communications.

Data read strategy – Second, it is not clearly explained in [8] that how the actual read operations are performed. The principle of our I/O strategy is that we reduce the total I/O time by merging many small random reads to a single large contiguous read, and then overlap the large read operation with multiple SGD iterations. Thus, the efficient read strategy is critical in our I/O strategy. We discussed how to efficiently perform I/O operations using a stand-alone I/O-dedicated thread. Assuming G workers run on each node using G GPUs, by having an I/O-dedicated thread per process, G CPU cores within each node will be fully occupied by the I/O threads. Considering the number of CPU cores in each modern HPC compute node, e.g., Summit at ORNL supports up to 128 logical cores on one node, such an overhead is most likely negligible. Our experimental results demonstrate that the I/O-dedicated thread enables to explicitly overlap the I/O time with the training time without a significant overhead.

V. Conclusion

In this paper, we proposed an asynchronous I/O strategy for large-scale deep learning applications. We applied our I/O strategy to the real-world scientific applications and validated the effectiveness. Our experimental results demonstrate the importance of careful adjustment of I/O granularity. In order to minimize the overall I/O time, each I/O operation should read a sufficiently large amount of data at once. In addition, we discussed how to effectively overlap the I/O time with the training time by employing an I/O-dedicated thread per process. Without using hardware-assisted features, our implementation using off-the-shelf software features virtually eliminated the I/O time. Considering the increasing available memory space in modern HPC systems, communication-based data shuffling and the corresponding I/O strategy can be an important future work.

Acknowledgment

acknowledgment

References

[1] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M. M. A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov *et al.*, "Deep learning at 15pf: supervised and semi-supervised classification for scientific data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–11.

[2] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica *et al.*, "Exascale deep learning for climate analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 649–660.

[3] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook *et al.*, "Cosmoflow: Using deep learning to learn the universe at scale," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 819–829.

[4] W. Dong, M. Keceli, R. Vescovi, H. Li, C. Adams, E. Jennings, S. Flender, T. Uram, V. Vishwanath, N. Ferrier *et al.*, "Scaling distributed training of flood-filling networks on hpc infrastructure for brain mapping," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 52–61.

[5] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Towards scalable deep learning via i/o analysis and optimization," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2017, pp. 223–230.

[6] ——, "Parallel i/o optimizations for scalable deep learning," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 720–729.

[7] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "Fanstore: Enabling efficient and scalable i/o for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.

[8] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-aware i/o pipelining for large-scale deep learning on hpc systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 145–156.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[13] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[14] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[15] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[16] R. Ben-Shalom, C. M. Keeshen, K. N. Berrios, J. Y. An, S. J. Sanders, and K. J. Bender, "Opposing effects on nav1. 2 function underlie differences between scn2a variants observed in individuals with autism spectrum disorder or infantile seizures," *Biological psychiatry*, vol. 82, no. 3, pp. 224–232, 2017.

[17] P. W. Spratt, R. Ben-Shalom, C. M. Keeshen, K. J. Burke Jr, R. L. Clarkson, S. J. Sanders, and K. J. Bender, "The autism-associated gene scn2a contributes to dendritic excitability and synaptic function in the prefrontal cortex," *Neuron*, vol. 103, no. 4, pp. 673–685, 2019.

[18] NERSC. Cori gpu nodes. [Online]. Available: https://docs-dev.nersc.gov/cgpu/

[19] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell *et al.*, "The design, deployment, and evaluation of the coral pre-exascale systems," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 661–672.

[20] R. Ben-Shalom, J. Balewski, A. Siththaranjan, V. Baratham, H. Kyoung, K. G. Kim, K. J. Bender, and K. E. Bouchard, "Inferring neuronal ionic conductances from membrane potentials using cnns," *bioRxiv*, p. 727974, 2019.

[21] Y. Oyama, N. Maruyama, N. Dryden, P. Harrington, J. Balewski, S. Matsuoka, M. Snir, P. Nugent, and B. Van Essen, "Toward training a large 3d cosmological cnn with hybrid parallelization," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2019.

[22] Google. Better performance with the tf.data api. [Online]. Available: https://www.tensorflow.org/guide/data_performance/

[23] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019.