LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-812691

The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism

Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, B. Van Essen

July 21, 2020

IEEE Transactions on Parallel & Distributed Systems

Disclaimer

The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs with Hybrid Parallelism

Yosuke Oyama* †, Naoya Maruyama†, Nikoli Dryden‡†, Erin McCarthy§†, Peter Harrington¶, Jan Balewski¶, Satoshi Matsuoka‖ *, Peter Nugent¶ and Brian Van Essen†
* Tokyo Institute of Technology, oyama.y.aa@m.titech.ac.jp
† Lawrence Livermore National Laboratory, {maruyama3,vanessen1}@llnl.gov
‡ ETH Zürich, ndryden@ethz.ch
§ University of Oregon, emccarth@cs.uoregon.edu
¶ Lawrence Berkeley National Laboratory, {PHarrington,balewski,penugent}@lbl.gov
‖ RIKEN Center for Computational Science, matsu@acm.org

***Abstract*—We present scalable hybrid-parallel algorithms for training large-scale 3D convolutional neural networks. Deep learning-based emerging scientific workflows often require model training with large, high-dimensional samples, which can make training much more costly and even infeasible due to excessive memory usage. We solve these challenges by extensively applying hybrid parallelism throughout the end-to-end training pipeline, including both computations and I/O. Our hybrid-parallel algorithm extends the standard data parallelism with spatial parallelism, which partitions a single sample in the spatial domain, realizing strong scaling beyond the mini-batch dimension with a larger aggregated memory capacity. We evaluate our proposed training algorithms with two challenging 3D CNNs, CosmoFlow and 3D U-Net. Our comprehensive performance studies show that good weak and strong scaling can be achieved for both networks using up to 2K GPUs. More importantly, we enable training of CosmoFlow with much larger samples than previously possible, realizing an order-of-magnitude improvement in prediction accuracy.**

***Index Terms*—deep learning, convolutional neural network, model-parallel training, hybrid-parallel training**

I. Introduction

Recent advances in deep learning, especially convolutional neural networks (CNNs), have become a subject of significant research interest in emerging scientific workflows in research fields such as cosmology [1], medical image analysis [2], climate analysis [3], and turbulent flow simulations [4]. This is thanks to its potential for extraordinary impact, as demonstrated in image and speech classification [5], [6], playing games [7], and translating natural languages [8], among others. While early successful results have been reported in applying deep CNNs to scientific problems, training highly robust and accurate scientific models can be severely constrained as they tend to need to use much larger samples, such as 3D medical images or simulation outputs. Larger, high-dimensional samples can make deep CNNs even deeper with each layer becoming more compute intensive, making already long training times even longer. Furthermore, increased sample sizes directly increase the memory usage of model training, which is especially problematic in accelerators with limited memory capacity such as GPUs. 3D CNNs can consume tens to hundreds of gigabytes of memory, as exemplified in the cosmology and medical models evaluated in this work (see Section V). This easily exceeds the available memory capacity of typical high-end GPUs. These two major problems, high computational cost and memory usage, can severely reduce the effectiveness of deep learning, especially in scientific domains.

Parallelizing training can alleviate these problems by employing more compute and memory resources. However, the most commonly-used approach, data-parallelism, fails to adequately address the challenges posed by extreme scientific problems due to its limited parallelism. While training samples are distributed over multiple processing elements (PEs) such as GPUs, each PE must still process at least one complete sample, resulting in limited reduction of per-PE memory pressure. Reducing the size of training samples by, e.g., lowering resolution is a common workaround. However, it inevitably loses information in the original data, which can be critical to train highly accurate models. Even if memory is not a problem, the degree of parallelism is still limited by the number of samples per mini-batch, which cannot be arbitrarily increased without adversely affecting the model accuracy [9].

To address these issues in large-scale 3D CNNs, **strong scaling is required**. We present an end-to-end training framework that extends the parallelism beyond the current state of practice for strong scaling. First, we develop a highly efficient hybrid-parallel algorithm for 3D convolutions that exploits parallelism both in the spatial and mini-batch dimensions, allowing one sample to be distributed over multiple PEs. This provides improved performance and is indispensable for breaking the memory barrier in 3D CNNs. Second, to achieve scalable end-to-end performance, **the performance of I/O pipeline must also strong scale** for a fixed number of samples, which may be gigabytes in size. We apply the same hybrid-parallel techniques to I/O, maximizing parallelism, increasing throughput, and minimizing scaling bottlenecks.

We demonstrate end-to-end scalable training by extending an existing DNN framework with our hybrid-parallel compute and I/O algorithms. In our experimental studies, we use CosmoFlow, a regression model for cosmology [1], and the

3D U-Net, a segmentation model for medical images [2], as representative large-sample 3D CNNs. We present comprehensive performance analyses of our algorithms and show that they collectively enable efficient strong scaling for both problems, i.e., parallel speedups without increasing the mini-batch size. Even more importantly, we demonstrate that by training CosmoFlow with much larger samples than previously possible, it is possible to realize **an order-of-magnitude improvement in prediction accuracy**, which can have a tremendous impact in accelerating scientific discoveries. This new capability enabled by our training pipeline is not limited to the particular problem but can be a critical tool for broader ML-enhanced scientific applications.

We summarize our contributions as follows:

- We present an end-to-end approach for strong-scaling training large-sample 3D CNNs. We address the compute, memory, and I/O challenges using hybrid-parallelism.
- We present a prototype implementation of our proposed approach by extending the LBANN framework [10] and demonstrate training on full-resolution samples for CosmoFlow (512^3) and the 3D U-Net (256^3). Our performance results show show good strong and weak scaling on up to 2048 GPUs. Training the CosmoFlow model is 1.77x faster when using 2048 GPUs over 512 GPUs, both using the same mini-batch size of 64. Similarly, a 1.42x speedup is achieved for the 3D U-Net when using 512 GPUs over 256 GPUs.
- We provide detailed model-based performance analyses of both problems in order to give comprehensive understanding of their scaling efficiencies.
- We demonstrate a significant improvement in prediction accuracy by using full-resolution data. The CosmoFlow model trained with 512^3 samples, while requiring at least eight V100 GPUs per sample, realizes ten times lower mean squared error than when trained with 128^3 samples, which was the largest size reported previously.

II. Background

Here, we first describe two fundamental methods to train a single DNN in parallel: data- and model-parallelism. We also describe hybrid-parallel training, a combination of both which couples spatial partitioning and data-parallelism for improved scalability. Then we introduce the CosmoFlow and the 3D U-Net models in more detail.

A. Data-, model-, and hybrid-parallelism

Mini-batch Stochastic Gradient Descent (SGD) is the most widely used technique to optimize the parameters of a given deep neural network, and has the form:

$$W^{(t+1)} = W^{(t)} - \eta^{(t)} \sum_{n=1}^{N} \nabla L \left(x_n; W^{(t)} \right),$$

where $W^{(t)}$ are the network parameters at step t, $\eta^{(t)}$ is the learning rate at step t, N is the mini-batch size, L is the loss function, and x_n is the nth sample.

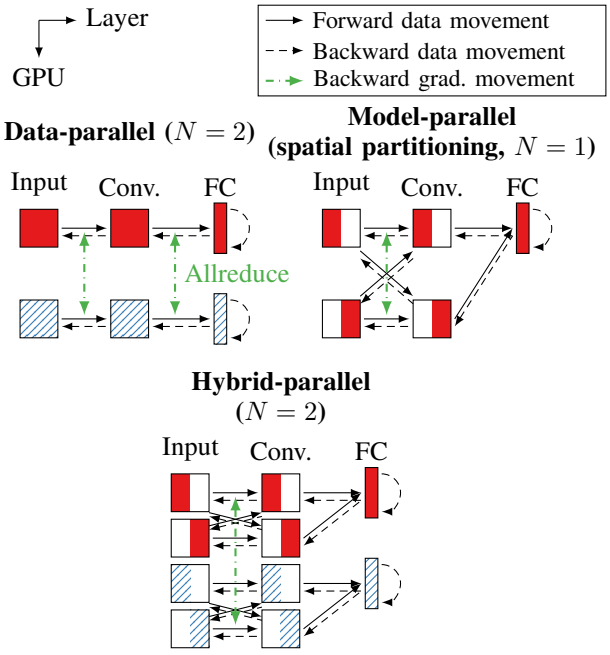


Fig. 1: Three different parallel strategies for deep neural networks. N denotes the mini-batch size. "Data-parallel" and "Hybrid-parallel" compute a mini-batch of two samples (and), while "Model-parallel" splits the spatial dimension of one sample ( ) into two GPUs. Note that data movement within a single process is typically cheap.

1) Data-parallelism: Data-parallel training, which partitions samples to compute ∇L in parallel, is the most widely-used training technique (Figure 1, top left). It takes advantage of the fact that N is typically large enough to efficiently parallelize training on up to thousands of GPUs [11]–[13] and that communication requirements are relatively small compared to the compute requirements for CNNs.

However, data-parallelism is limited by the number of samples in a mini-batch, the memory requirements of training, and how well the model learns.

2) Model-parallelism: On the other hand, model-parallel training (Figure 1, top right) parallelizes the computation of ∇L for each data sample across multiple GPUs. This enables additional parallelism to be exploited, and partitions data across multiple GPUs to reduce memory requirements. In data-parallel training, if the memory requirements (including necessary intermediate activations) exceed the memory capacity of a GPU, training is infeasible. In contrast, with model-parallelism, the memory requirements are roughly inversely proportional to the number of partitions. This strong-scaling advantage is especially attractive for high-dimensional CNNs, since the large input data results in huge intermediate activation tensors during training.

At the same time, model-parallel training requires careful framework design to mitigate overheads. While data-parallel training requires only a single global allreduce per layer

to aggregate parameter updates, model-parallel training can involve many fine-grained communication operations and synchronizations in each layer.

Many strategies exist to exploit model-parallelism in CNNs, including spatial partitioning, channel/filter partitioning, and layer-wise pipelining (see Section VI). In this paper, due to the high-resolution and 3-dimensional input data, we focus on spatial partitioning as a natural way to decompose the computation. Spatial partitioning distributes input and activation tensors by partitioning the height, width, and depth dimensions, similarly to traditional stencil computations. This introduces a halo exchange (an exchange of spatial boundary data between neighbor processes) within convolutional and pooling layers to maintain correctness, although it can be overlapped to hide communication overhead. For 3D data, the improved surface-to-volume ratio of the problem mitigates these communication overheads even further. Existing work on spatial partitioning has been limited to 2D data; here, we extend spatial partitioning to support 3D data.

3) Hybrid-parallelism: "Hybrid-parallelism" is the combination of data-parallelism and model-parallelism (Figure 1, bottom). Hybrid parallelism takes advantage of both the low overhead of data-parallelism to weak scale and of model-parallelism to strong-scale onto more compute resources. It requires careful selection of the relative balance of parallelism to achieve good scalability, as demonstrated in Section V-B.

B. CosmoFlow

CosmoFlow [1] is a project to use deep learning to estimate the values of important cosmological parameters from 3D cosmological simulations. One of the goals in cosmology is to understand and control the underlying systematics in a cosmological survey. As there is only one universe to observe, and the entire universe is needed to make these measurements, constraining the effects of systematics falls to computationally expensive simulations. The CosmoFlow network aims to replicate both the systematics from survey operations as well as those nature forces upon us. Creating surrogates for these simulations is necessary to generate the sheer statistical numbers needed to control the systematics.

Mathuriya et al. conducted thousands of independent N-body dark matter simulations with varied initial cosmological parameters, and constructed a dataset from them. The task is to predict the initial parameters from 3D mass distributions. The original spatial dimensions, 512^3 voxels, required too much memory to train on, so each sample was split into 128^3 voxel sub-volumes which are used as different data samples. As a result, the CosmoFlow network was trained with 99,456 training samples, each a 128^3-voxel 3D histogram of particle counts. It was reported that training with large mini-batches containing 8192 samples did not converge to comparable accuracy as smaller, 2048-sample mini-batches. These two problems are easily solved by our approach, because each sample is distributed among multiple GPUs, avoiding memory limits and allowing scaling without large mini-batches.

The latest CosmoFlow dataset is the "2019_05_4parE" dataset [14]. It contains $10{,}017$ simulated universes, each of which is composed of four channels (redshifts) and is 512^3 voxels, stored as 16-bit integers, along with four cosmological parameters that were used to generate the universe. These are Ω_M, the proportion of matter in the universe; σ_8, the amplitude of mass fluctuations at a distance scale of 8 Mpc/h; n_s, the scalar spectral index of the spatial curvature of a comoving slice of space-time; and H_0, Hubble's constant. The dataset is about 9.77 TiB in size. We normalize the parameters to be in $[-1, 1]$ when training, in line with prior work.

In this paper, we distinguish the datasets with the spatial input sizes, 128^3 to 512^3. We split each dataset into 80%, 10%, and 10% as training, validation, and test datasets respectively. Additionally in this work, we will test the hypothesis that by allowing neural networks to observe entire data samples during training it is possible to learn longer range properties in the data and improve the quality of this type of regression model.

C. The 3D U-Net

The 3D U-Net [2] is a 3D version of the U-Net [15], a 2D CNN for image segmentation. It replaces all 2D operations with 3D operations to perform volumetric segmentation on 3D data. U-Nets have been applied to a wide range of 2D and 3D segmentation applications, such as biological image analysis [2], [15] and CT image analysis [16].

In this paper, we apply the 3D U-Net to the Liver Tumor Segmentation (LiTS) dataset [17], where the task is to segment liver lesions in 3D CT scans. It consists of 131 CT scans for training and 70 for testing. Each is composed of a variable number of 512×512 slices and per-voxel ground-truth labels. To use a consistent input size, we down-sample the non-slice dimensions and up- or down-sample the slice dimension so each sample is 256^3 voxels. We convert the original dataset to equivalent HDF5 files with 16-bit integers.

The most significant difference between the 3D U-Net and CosmoFlow networks is that it uses deconvolution layers to upsample activations to their original size. As the memory requirements for activations is cubic in the layer's spatial dimensions, the 3D U-Net requires a huge amount of memory near both the input and output layers, compared to the CosmoFlow network with the same input size. Furthermore, the CT image and labeled segmentation of each sample in the LiTS dataset are both the same size. Since the labels are not small (e.g. a class label), we must consider the I/O performance of reading them in addition to the inputs. Thus, the 3D U-Net helps demonstrate performance in different regimes than CosmoFlow.

III. Scalable training of 3D CNNs

Scaling up the training of neural networks for large 3D data cubes requires innovation in both spatially distributed convolution/deconvolution, and parallel data ingestion, reuse, and movement. In this section, we discuss each of these in turn. We also propose a performance model to predict layer-wise computation and communication time for a given network and

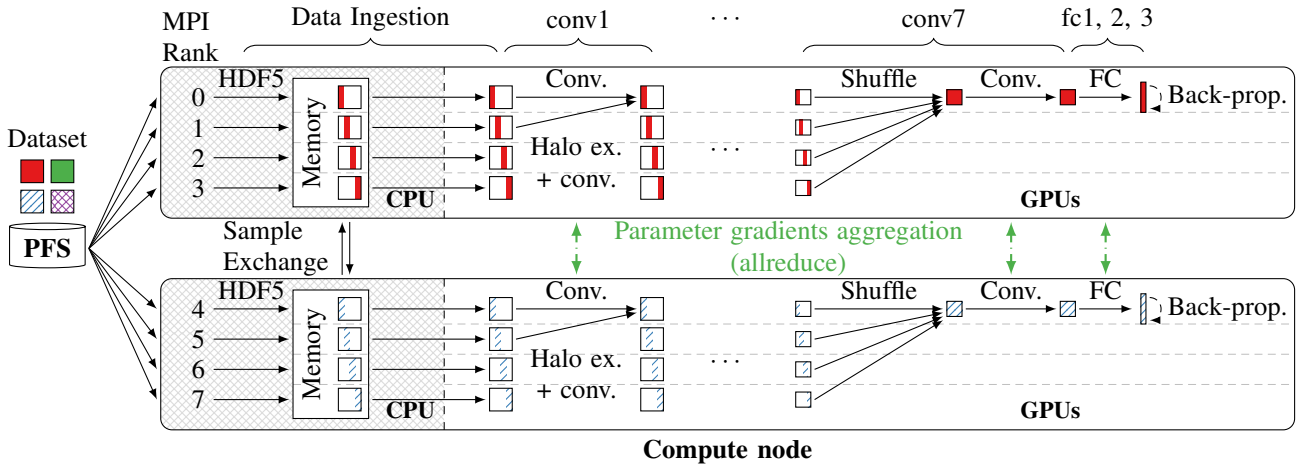


Fig. 2: Overview of hybrid-parallel training. Each node contains four processes which partition a single data sample.

runtime configuration to validate our computational efficiency. We use the Livermore Big Artificial Neural Network Toolkit (LBANN) [10] to implement our approach, as it has already demonstrated good scalability for 2D spatial partitioning [18]. Our training pipeline is summarized in Figure 2.

Part of the contributions described here derive from traditional solutions in HPC systems and software architecture, but are extended to tackle the complexities of training deep neural networks.

Notation. We adopt cuDNN's notation for tensor dimensions, using N, C, D, H, and W to refer to samples, channels, depth, height, and width, respectively. Unless explicitly mentioned, we assume all tensors are fully packed. We use "D-way", "$D \times H$-way", or "$D \times H \times W$-way" to refer to how many ways the depth, height, and/or width are partitioned. We omit N when the remaining GPUs are used for data-parallelism in a hybrid-parallel manner. For example, if the total number of GPUs is 16, 2-way means there are $16/(2 \times 1 \times 1) = 8$ groups each of which split a data sample onto two GPUs in the depth dimension.

A. Hybrid-parallel Implementation of 3D CNNs

The hybrid-parallel training requires partitioning activations in their spatial and sample dimensions over distributed PEs. Once they are partitioned, each layer computation is done locally except for layers that involve data dependencies across partitions, which include convolutions, pooling, and batch normalization. Convolution and pooling have a spatial dependency that can be resolved with halo exchanges. For batch normalization, partial statistics over partitions need to be aggregated with allreduces to correctly compute per-channel statistics for samples.

In this work, we extend an existing library [18] that provides the basic infrastructure for implementing hybrid-parallel 2D CNNs. This extension requires significant additional work to both support 3D data and achieve good performance. We began by adding support for 5D tensors (required for 3D data) to both the library and underlying distributed linear algebra backend.

However, we found several performance issues and missing support for realizing real end-to-end training of 3D CNNs, which had not been reported before. Several operations were designed for 2D data and not well-optimized for large, 3D tensor shapes. For example, we identified that the existing packing and unpacking CUDA kernels for the neighbor communication of boundary regions were sub optimal for our target problems. We developed a suite of new optimized packing/unpacking kernels for common convolutional filters (e.g., 3^3 and 5^3). Such optimized kernels are automatically picked when possible. Similarly, we extend other network components, such as distributed batch-normalization and the cross-entropy loss, with optimized versions for large 3D problems. In general, we find that due to the large data sizes, operations that are normally considered cheap can in fact dominate runtime if not well implemented.

The library also lacked support for many layers necessary for more general CNN architectures, as it was designed primarily for sequential networks. The 3D U-Net, which contains both down- and upsampling branches with residual connections between, required additional features. We developed distributed, hybrid-parallel implementations of deconvolution and support for more flexible distributed tensor manipulations for the residual connections.

Unlike the activations, the layer parameters are relatively small in our networks (e.g., 9.5M for CosmoFlow, see Table I). Thus, we do not expect a significant benefit from partitioning them (e.g., with channel/filter parallelism [19]) as we do the activations. We use standard data-parallel techniques to aggregate parameter updates with allreduces in backpropagation (green arrows in Figure 2).

Note that these extensions do not change the fundamental architecture of the underlying framework, but extend some base classes such as the tensor class and convolutional layer classes to support hybrid-parallelism. Thus, our techniques can

be applied to any other deep learning framework.

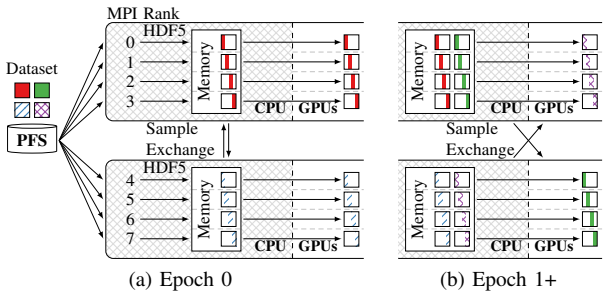
B. I/O performance optimization

Training the CosmoFlow and the 3D U-Net networks requires the ingestion and shuffling of many huge samples, each of which is 1 GiB and 64 MiB in size respectively, and is accessed once per epoch in random order. A key challenge is that in the steady state, ingesting training data from the PFS quickly becomes the dominant portion of the runtime. Furthermore, the 10 TB CosmoFlow dataset is too large to cache in local storage on our compute nodes. For example, our typical configuration for the CosmoFlow network uses a mini-batch of size 64 and our system has 240 GB/s of PFS bandwidth. Thus, loading each mini-batch requires at least 256 ms, which is prohibitively slow (Figure 4). Handling this workload efficiently requires solving three tasks: maximizing the utilized bandwidth to the parallel file system (PFS), caching the data set efficiently in distributed memory to avoid subsequent access to the PFS, and efficient shuffling of samples from the data cache during each epoch.

However, when training with hybrid-parallelism, even if in-memory caching is enabled, we found that we require that data samples be spatially partitioned and mini-batches are typically small. Traditional sample-parallel I/O approaches have limited parallelism in this regime, and would require data be redistributed to match the spatial parallelism, limiting strong scaling and opportunities for hiding I/O overhead. In Figure 5, we demonstrate that without this spatial-parallel I/O technique training of the CosmoFlow network does not scale at all with any number of GPUs, even if the entire dataset is distributed among the host memory of the computing nodes. This problem is even more acute for the 3D U-Net, where we also spatially distribute the ground-truth segmentation.

To address this, we develop a new parallel I/O pipeline where each process fetches its local *hyperslab*, or contiguous 3D fragment, of a data sample. This incorporates spatial parallelism into the I/O process to enable strong scaling PFS bandwidth and minimize data shuffling, redistribution, and memory footprints. We build on existing infrastructure in LBANN, including its C++ data readers and distributed, in-memory data cache [20] to reduce PFS accesses.

Our data reader uses Conduit [21] as both an in-memory data structure and an interface to an I/O backend, such as HDF5 (Figure 3). Conduit is an open source data exchange library that provides efficient ways of exchanging scientific data. In prior work, LBANN has been optimized to provide parallel I/O using both MPI and multi-threading, but was limited to a single MPI rank per sample. To overcome this performance bottleneck and support spatially parallel I/O, we rearchitected the data ingestion pipeline to use parallel HDF5 with MPI-IO. This allows multiple ranks which each require one hyperslab of a sample to coordinate their activity when ingesting large samples. Using this, data loading can now track the strong-scaling of our hybrid-parallelism while minimizing data redistribution, as each rank reads only the data it needs.



(a) Epoch 0 (b) Epoch 1+

Fig. 3: Data movement during both the initial epoch 0 (left) and steady state epoch 1+ (right). During epoch 0, HDF5 ingests hyperslabs in parallel into the data store. During epoch 1+, the data store distributes the hyperslabs for each sample in the mini-batch that is about to be trained on.

However, as GPU performance continues to outstrip I/O bandwidth, it is also necessary to minimize PFS accesses. To do this, when samples are loaded (Figure 3a), they are placed into Conduit nodes and then into LBANN's distributed, in-memory data store to cache the samples for the duration of training. We extended the data store to hold a sample as a collection of hyperslabs. This aligns the spatially parallel I/O, training, and data caching for best performance with hybrid-parallel convolution.

After the first epoch is complete, the data store has cached the entire data set, which it will distribute on subsequent epochs. Before each epoch, the data store computes a global owner map and a schedule mapping samples to SGD iterations. This allows the data store to redistribute hyperslabs of samples as needed for the upcoming mini-batch (see Figure 3b).

As we strong scale, the capacity of the data store increases in proportion to the compute resources, allowing increasingly large datasets to be cached. This is also well-positioned to take advantage of node-local storage and non-volatile memories on future systems.

C. Performance Modeling

We use a performance model to predict the time to perform one iteration of training with given a configuration, such as the mini-batch size and the number of nodes, to understand the quantitative behavior of the framework and validate performance. We first collect the time to perform (de)convolution, pooling, and batch normalization kernels with various input sizes on a single GPU using cuDNN [22], and then we combine the benchmark results with a communication model to predict layer-wise runtime on multiple GPUs.

The time to perform forward-computation of convolutional or pooling layer l is

$$FP_l = \max\left\{Comp_l\left(D_l^{main}\right), \sum_{d=0}^{2} 2SR\left(D_{l,d}^{halo}\right)\right\} + Comp_l\left(D_l^{halo}\right)$$

where $Comp_l(D)$ is time to compute layer l on a given domain D, and $SR(D)$ is time to perform peer-to-peer send-receive communication between two GPUs (via NVLink or

inter-node InfiniBand depending on the location of the two processes). The shape of D_l^{main}, the domain which can be computed without halo communication, and $D_{l,d}^{halo}$, the domain which requires halo region to be computed, are defined by the partitioning of the layer. We define BD_l and BF_l, the time to perform the backward-data and backward-filter passes on layer l, respectively, in a similar manner.

To estimate $Comp_l(D)$, we benchmark each layer type on a single GPU. Unless noted, we use the largest cuDNN workspace possible, and autotune to find the fastest convolution algorithms. We use the median of three trials after warmup. To estimate $SR(D)$, we use Aluminum's ping-pong benchmark and apply linear regression to estimate the time for arbitrary message sizes.

The time for a batch-normalization layer is the sum of the computational time and time to perform allreduce of the local sum and squared-sum of each channel.

Finally, the total time of the network is

$$Cost = \sum_l FP_l + \max\left\{\sum_l \left(BD_l + BF_l\right), \sum_l AR_l(\theta_l)\right\},$$

where AR_l is time to perform allreduce among all of the GPUs and θ_l is the number of parameters of layer l. To estimate AL_l, measure the performance on one node (4 GPUs) to 128 nodes (512 GPUs), with float vectors of 1 to 16 M elements, and apply linear regression [23], [24] with logarithmic transformations to predict the time for a given message size and the number of GPUs.

We ignore the cost of non-3D part of the 3D CNNs (e.g., fully-connected and loss layers), since their costs are negligible compared to other costs, such as allreduces or convolution. We also ignore the cost of I/O for loading data samples from the PFS or between processes, as our optimized pipeline mitigates I/O costs drastically for the two networks we use in this paper.

IV. Extended CosmoFlow Model

We now discuss extensions we make to the CosmoFlow network, as this is the first attempt to train the network with $64\times$ larger input data than before. We use the CosmoFlow model presented in the previous work as our baseline model and extend it to improve its prediction accuracy by exploiting our new hybrid-parallel training capabilities.

Table I summarizes three models, corresponding to the 128^3, 256^3, and 512^3 voxel training datasets, respectively. For each of the models, we have applied several extensions to the original baseline model. First, we add a batch normalization layer [25] after every convolutional layer. Ravanbakhsh et al. reported that batch normalization was critical in training a similar model [26]. However, in the original CosmoFlow model, it was dropped due to the computational cost of batch normalization, especially in a distributed training setting. We present training results in both configurations (Section V) and observe that while batch normalization increases memory requirements, it improves final prediction accuracy. Second, in order to simplify comparison of the three models, we insert additional pooling layers in the 256^3 and 512^3 models (the pool6 layer in both models and the pool7 layer in the 512^3 model). Finally, we experimentally identified several minor parametric changes that improve prediction accuracy or simplify the implementation of distributed convolution, including removal of biases and use of padding in convolutional layers. We removed biases as we observed significant performance overheads for them in practice.

TABLE I: CosmoFlow network architecture. W_i is the input spatial width. cN→pN are convolution followed by pooling and fc$N$ are fully connected layers. We use stride 1 convolution and stride 2 pooling unless noted. All layers use "same" padding.

Layer(s)		Output width		
Name(s)	**Filter**	$W_i = 128$	$W_i = 256$	$W_i = 512$
c1→p1	16×3^3	128^3→64^3	256^3→128^3	512^3→256^3
c2→p2	32×3^3	64^3→32^3	128^3→64^3	256^3→128^3
c3→p3	64×3^3	32^3→16^3	64^3→32^3	128^3→64^3
c4→p4	128×3^3 (stride of 2)	8^3→4^3	16^3→8^3	32^3→16^3
c5→p5	256×3^3	4^3→2^3	8^3→4^3	16^3→8^3
c6→p6	256×3^3	2^3→N/A	4^3→2^3	8^3→4^3
c7→p7	256×3^3	2^3→N/A	2^3→N/A	4^3→2^3
fc1	2048	2048	2048	2048
fc2	256	256	256	256
fc3	4	4	4	4
# conv. ops. [GFlops/sample]		55.55	443.8	3550
(Forward) [GFlops/sample]		18.52	147.9	1183
Memory [GiB/sample]		0.824	6.59	52.7
# parameters $[10^6]$		9.44	9.44	9.44

The remaining details follow the original model: We use leaky ReLU [27] activations (except for the last layer), dropout with a keep probability of 0.8 after every fully-connected layer, and adopt the mean squared error as the loss function. We use the Adam [28] optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ and a linear learning-rate decay schedule which leads to 0.01x of the initial rate in 100 epochs. We perform grid search to tune initial learning rate $\eta^{(0)}$ for each network.

To compare with prior work and study the impact of data volume, we synthesize two datasets with data volumes of size 128^3 and 256^3, by splitting each 512^3 cube into 64 and 8 sub-volumes, respectively. This is analogous to the partitioning method used in the original work. The intuition behind this was that the data volumes should be sufficiently large to contain galaxy clusters, which are sensitive cosmological probes.

Training the largest network needs 4 GPUs to store the 52.7 GiB of memory required (Table I). When batch normalization layers are introduced, memory requirements double, necessitating at least 8 GPUs (2 nodes) per sample.

V. Evaluation

In this section, we first evaluate the computational performance of our hybrid-parallel implementation for CosmoFlow

and the 3D U-Net in both strong scaling (fixed global mini-batch size) and weak scaling (fixed mini-batch size on each GPU) regimes. Then, we demonstrate the importance of increasing the input data resolution of the CosmoFlow network to improve its prediction accuracy. To our knowledge, this work is the first attempt to train the CosmoFlow network with the full-resolution universe data instead of partitioning them into small sub-volumes.

A. Evaluation environment

We use Lassen, a GPU supercomputer at Lawrence Livermore National Laboratory composed of 792 nodes. Each node has two IBM POWER9 CPUs with 256 GB memory and four NVIDIA V100 GPUs with 16 GB memory and NVLink2. Each CPU has two GPUs directly connected to it via NVLink, and the two GPUs on each socket are also directly connected via NVLink. The network is dual-rail EDR InfiniBand.

We use GCC 7.3.1, CUDA 10.1, cuDNN 7.6.4, NCCL 2.4.2 and IBM Spectrum MPI 10.2.0.11rtm2. We use auto-tuning to select cuDNN convolution algorithms. We use FP32 for computation throughout the experiments. We do not use FP16 mixed-precision training (or Tensor Cores), as the impact of applying low-precision training to CosmoFlow has not yet been evaluated.

For the 3D U-Net, we use the original network architecture proposed in the paper [2], but increase the input/output size to 256^3. As mentioned in Section II-C, the network consumes much more memory than the CosmoFlow network for the same input data size, so we use a smaller size to keep the number of GPUs per sample the same as the CosmoFlow experiments.

B. Strong scaling

Training neural networks with strong scaling increases the number of compute resources brought to bear without perturbing the learning behavior of the model. In conjunction with hybrid-parallelism this technique allows us to use an unprecedented number of GPUs per data sample. Figure 4 shows the strong scaling performance of the CosmoFlow network with the 512^3 dataset. We use global mini-batch sizes (N) of 1, 2, 4, 16 and 64, and split the network in the depth dimension. We run the framework for 4 epochs with a 128-sample subset of the dataset (if the mini-batch size is smaller than 128), or the full dataset, and show the median iteration time except for the first epoch. We also show predicted times by our performance model, which largely match with the actual measured times, confirming our implementation performed as expected.

As shown in the figure, when the mini-batch size, N, is 16 and 64, we achieve speedups of 1.98x with 512 GPUs (128 nodes) compared to 128 GPUs (32 nodes), and 1.77x with 2048 GPUs (512 nodes) compared to 512 GPUs (128 nodes), respectively. We note that for 16 samples the performance gain for going to 1024 GPUs falls off, as the problem becomes over-decomposed. However, the computational performance in terms of throughput can still be scaled further by increasing the batch size to 64. As shown in Section V-D, this mini-batch

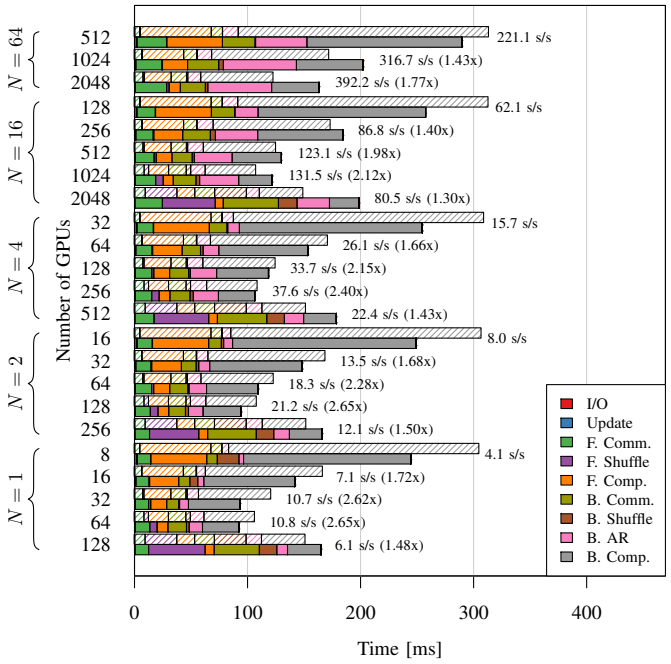


Fig. 4: Strong scaling of CosmoFlow. Shaded bars show time predicted by the performance model. "F." and "B." are forward and backward passes, resp. N is the mini-batch size. Bars are annotated with throughput (samples/s) and speedup relative to the minimum setting with the same N.

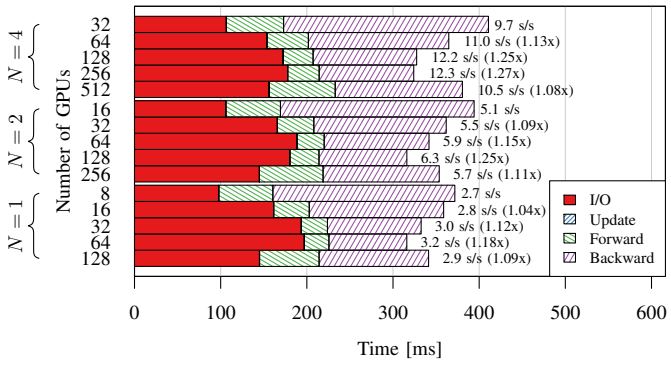
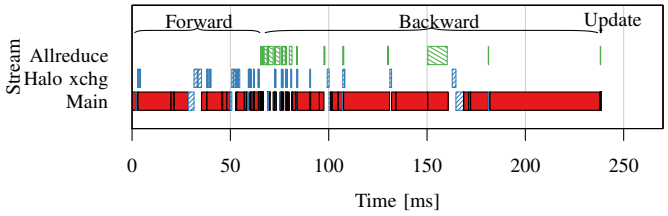
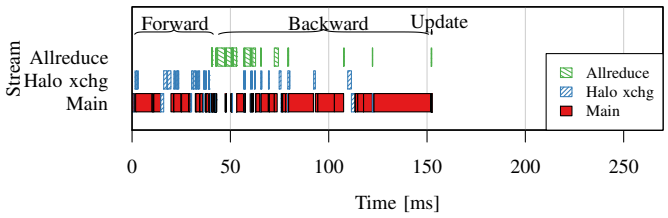


Fig. 5: Strong scaling of CosmoFlow without spatial-parallel I/O, using only distributed caching with Conduit.

size is a reasonable choice for actual training, and thus we prove that we successfully scale the training of the CosmoFlow network to thousands of GPUs. Furthermore, the I/O time is almost invisible in the figure since it is almost completely overlapped with computations in our optimized I/O pipeline. This makes a significant contrast to conventional I/O methods in terms of strong scaling performance, as their I/O parallelism is limited by the mini-batch size. In fact, without our spatially-parallel I/O approach, the iteration time does not scale due to the I/O overhead, as indicated in Figure 5, which visualizes



(a) $N = 4$, 512^3, 8-way, 32 GPUs



(b) $N = 4$, 512^3, 16-way, 64 GPUs

Fig. 6: Single-GPU execution timelines for training the 512^3 dataset with mini-batch size 4. Top: 8 GPUs/sample (32 total). Bottom: 16 GPUs/sample (64 total). We show one iteration of the root process's GPU of each run.

the impact of the I/O overhead when the spatial-parallel I/O is disabled. This demonstrates the necessity of strong-scaling I/O along with compute in order to efficiently parallelize training.

To understand the parallel efficiency of the implementation and identify potential bottlenecks, Figure 6 shows the GPU execution timeline of a mini-batch iteration when 32 and 64 GPUs are used to train the 512^3 model with a mini-batch size of 4. A speedup of approximately 1.66x is achieved using $2\times$ the number of GPUs. The "Main" row corresponds to the CUDA stream where compute kernels are launched; the "Halo xchg" row is an asynchronous stream to perform on-device halo exchanges; and the "Allreduce" row corresponds to a stream used by the asynchronous allreduce operations by NCCL. From the beginning of back propagation, NCCL starts to communicate computed parameter gradients among processes asynchronously to the main computation stream. Since the communication of gradient updates is done asynchronously, this does not block the compute kernels. In both cases, the main streams are nearly fully packed, indicating the GPU compute units are fully occupied. Similarly, the timelines indicate that the cost of our optimized halo exchanges is almost negligible in these scenarios. Overall, we see that the speedup from the 8-way to 16-way parallelization is mostly determined by the speedups of the individual convolution kernels in the cuDNN library. In this work, we have exclusively relied on cuDNN for optimized convolution kernels. These results indicate that they may not be well-tuned for non-cube domains, as we only achieved 1.66x speedup going from 8-way to 16-way parallelization. Identifying the local compute kernels as the bottleneck to better scaling is also corroborated by our performance model, shown in Figure 4, which was generated by profiling cuDNN.

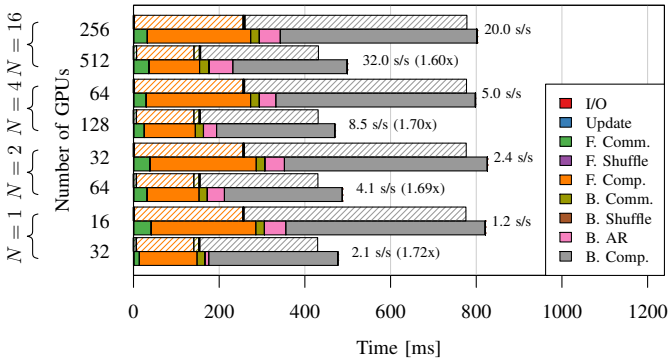


Fig. 7: Strong scaling of the 3D U-Net. Shaded bars show iteration time predicted by the performance model.

TABLE II: Achieved performance of CosmoFlow convolution layers compared to peak performance of cuDNN.

Depth	N	Layer	Time [ms]	Perf. [TFlop/s]	Peak	Rel. [%]
8-way	64	All	142.9	22.6	23.6	95.6
32-way	64	All	48.8	89.9	109.1	82.4
8-way	64	conv1	73.9	12.2	13.0	93.8
32-way	64	conv1	23.5	34.6	53.4	64.7

The ability to strong scale the performance of the 3D U-Net with an input size of 256^3 is shown in Figure 7. With this network, we have to use at least 16 GPUs per sample due to the memory requirements. We achieve good strong scaling performance between 16-way and 32-way partitioning, such as 1.42x on 512 GPUs over 256 GPUs with a mini-batch size of 16. As shown in Figure 7, similarly to the CosmoFlow network, most of the iteration time is spent in computation, implying that we achieve near-peak performance, despite the communication overheads of hybrid-parallelism compared to data-parallelism.

To better characterize our performance and scaling efficiency, we compare the performance of our distributed convolution layers in the 512^3 CosmoFlow network to the peak achievable performance of cuDNN in Table II. The Time and Perf columns give the measured performance of our code (including halo communication, etc.), measured with nvprof. In the Peak column, we report the TFlop/s achieved by running *only the local cuDNN kernel* for that configuration. This gives an effective upper bound on the performance we can achieve using cuDNN in our configuration. Finally, we report the achieved percent of this peak in the Rel column.

We observe that for CosmoFlow, we achieve 95.6% and 82.4% of this peak performance for 8- and 32-way partitioning, respectively. This indicates that the overhead of our distributed convolution is relatively small. We also observe another benefit of strong scaling: the potential peak performances exhibit super-linear scaling, albeit fairly slightly.

This is due to a larger, aggregated memory space available with a larger number of GPUs, allowing cuDNN to use more efficient convolution algorithms.

We also examined which layer dominates runtime, and for CosmoFlow we find that the conv1 layer accounts for almost half of the entire network runtime. This is due to the layer processing the largest spatial dimensions. For 8-way partitioning, we achieve excellent scaling efficiency; for 32-way partitioning, communication overheads limit gains, but still enable overall performance improvements.

While these results show good scaling efficiency with respect to the achievable peak with cuDNN, the TFlops/s achieved is relatively low compared to the theoretical peak of the hardware. This indicates that there is significant potential for further optimizing 3D convolution kernels.

C. Weak scaling

The second component of our hybrid-parallelism approach is to exploit data parallelism in conjunction with spatial parallelism. This allows us to explore the impact of weak scaling by increasing the global mini-batch size while tuning the learning rate. Figure 8 shows the weak scaling performance of the two 3D CNNs with different input sizes. For the CosmoFlow network with 128^3 cubes, the data size used in the previous work, we use per-GPU batch sizes of 8 and increase the global mini-batch size as we increase the number of GPUs. We evaluate the performance using 4-way and 8-way partitioning for reference. In the 512^3 case, we only evaluate hybrid parallelization where each data sample is partitioned among 8, 16 or 32 GPUs as nearly 53 GB of memory is required per sample as shown in Table I. While it is smaller than the aggregate capacity of 4 GPUs, we found that it results in an out-of-memory error as additional auxiliary data need to be allocated. We measure performance as in Section V-B.

In the case of CosmoFlow with 128^3 cubes, our implementation achieves nearly linear speedup up to 512 GPUs (128 compute nodes), in part because of the asynchronous overlapped communication engine of Aluminum and also because of the relatively high compute-to-communication ratio in 3D CNNs. We achieve a 65.4x speedup on 512 GPUs compared to 4 GPUs with the 128^3 cubes. In this case, the highest efficiency is achieved with the data-parallel scheme since the hybrid parallelization involves additional communications due to halo exchanges.

With 512^3, however, hybrid parallelization is required as the model is too large to fit into the device memory of a single GPU. Thus, we evaluate three configurations, 8-way, 16-way and 32-way, and the global mini-batch size is linearly increased as the number of GPUs is increased, resulting in 147.31x, 71.32x, and 37.2x of speedup on 2048 GPUs over 8, 16, 32 GPUs (where the mini-batch size is 1) respectively. With the 3D U-Net, we achieve good weak scalability (28.4x on 1024 GPUs over 32 GPUs with 32-way partitioning) as well.

In all cases, increasing the spatial parallelism results in lower throughput due to the additional communication overhead as well as the decreased compute efficiency of the cuDNN kernel library. However, we note that the hybrid parallelization enables further speedups for a given fixed mini-batch size as it is also shown in Section V-B.

D. CosmoFlow model accuracy improvement with 512^3 universe cubes

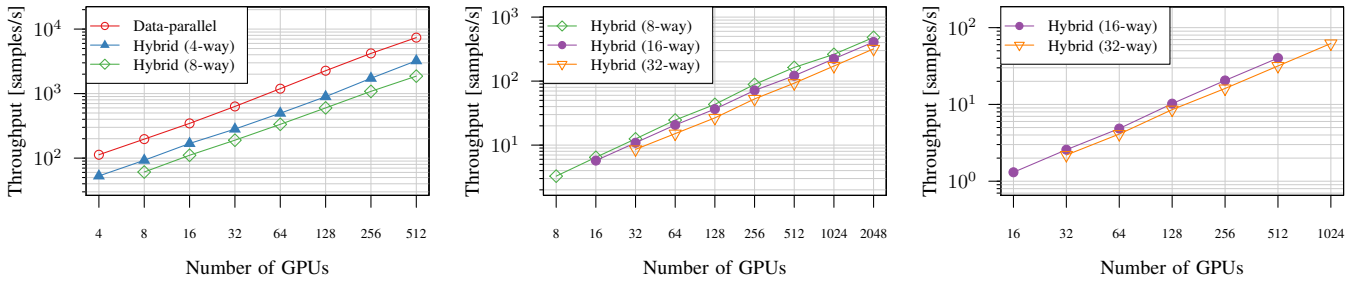
This experiment set out to test if training on entire data samples would improve the quality of the model learned by the network. Figure 9 shows training results of the CosmoFlow network with the full-resolution dataset (512^3) and split versions (128^3 and 256^3). We swept the initial learning rate from 10^{-4} to 10^{-2} logarithmically and show the results with the best. We train for 130 epochs with a mini-batch size of 64 in every configuration, and use the 4-way partitioning (256 GPUs in total) for the networks without batch normalization layers, or 8-way (512 GPUs in total) for networks with batch normalization, due to the increased memory requirements. To account for training variance, we show the median result of five trials with different initial random seeds.

We observe that the test loss decreases significantly as we increase the dataset size to 0.0169 MSE with 256^3 and 0.00727 MSE with 512^3 data. Adding batch normalization improves this result further, to 0.00445 MSE, achieving an order-of-magnitude improvement compared to the baseline 128^3 data. At the same time, we get 2.79x of speedup from 128^3 to 512^3 with the same number of GPUs and the same mini-batch size. This result implies that the CNN can be trained with the same computing resources and dataset size, but with a smaller mini-batch and small overheads (see Section V-C). This brings an opportunity to keep mini-batch sizes fixed and strong-scale onto more GPUs for speedup.

Figure 10 shows the correlation between the predicted and actual cosmological parameters and the associated residuals for our networks on each dataset. We clearly demonstrate improvements in the quality of predictions with increasing data volume; and the benefit of batch normalization. In particular, we observe that prediction of H_0 (the Hubble constant) shows the most improvement in accuracy with increasing data volume. This makes intuitive sense, as it is related to the large-scale expansion of the universe. As cosmological simulations move to sub-percent measurements, being able to test the quality of the surrogates via a greatly improved Cosmoflow network, with an order of magnitude improvement in the measurement of the cosmological parameters, is the only way to quickly validate the quality and precision of the models.

VI. Related work

Scalable training and model-parallelism has a long history in deep learning; Ben-Nun & Hoefler [29] provide a comprehensive overview. We discuss the most relevant. Early work on models such as AlexNet incorporated model-parallelism using grouped convolution or partitioning fully-connected layers [5], [30]. Coates et al. [31] applied spatial partitioning to locally-connected layers. Gholami et al. [32] consider spatial parallelism, but provide only simulated



(a) CosmoFlow, 128^3 (b) CosmoFlow, 512^3, BN layers (c) U-Net, 256^3

Fig. 8: Weak scaling of the two different 3D CNNs. We increase the global mini-batch size as we increase the number of GPUs. In the hybrid results, we partition a single sample by multiple GPUs in its spatial domain.

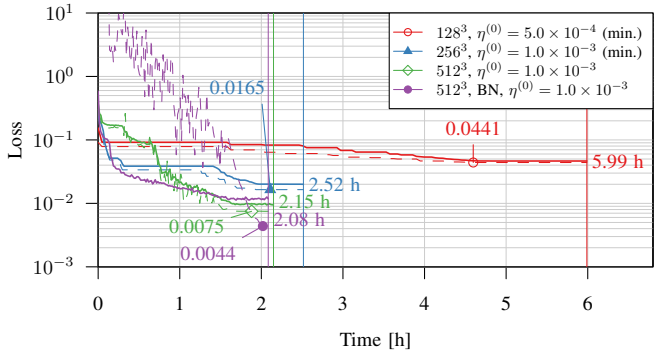


Fig. 9: Training/validation losses (solid/dashed lines respectively) and the smallest validation losses (points) of the CosmoFlow network with four different configurations. For 128^3 and 256^3, we show the minimum loss values at each point in time for visibility.

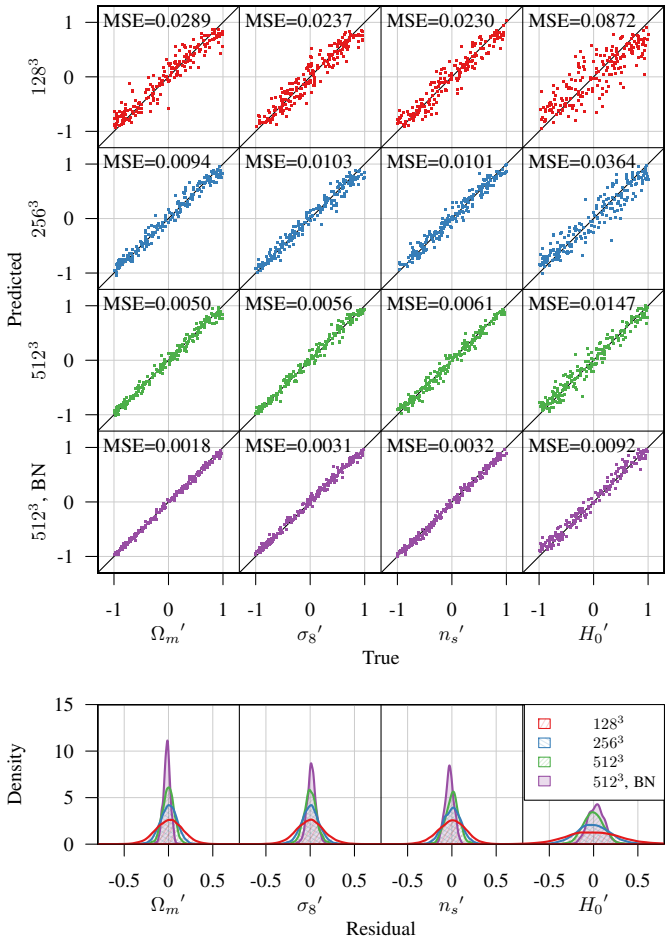


Fig. 10: True/predicted cosmological parameters (normalized to $[-1, 1]$) from four different configurations (top) and the distribution of the residuals (bottom). In the top figure, we show 200 randomly chosen data points for visibility.

results. FlexFlow [33] presents an automated system for identifying model- and hybrid-parallelism. Frameworks such as DistBelief [34], Project Adam [35], Mesh-TensorFlow [36], and TF-Replicator [37] also support limited forms of model-parallelism, but do not do spatial partitioning. The Distconv library [18] provides the basis for our 3D hybrid parallelism, but is limited to 2D CNNs. Extending this to efficiently support 3D CNNs requires significant novel work (Section III-A). Further, this work does not consider I/O, which is a key bottleneck for scaling 3D CNNs, nor does it demonstrate improved learning. Similarly, channel and filter parallelism [19] provides another method for partitioning data, primarily targeting *wide* CNNs. It also only considers only 2D data and neglects I/O. Further, channel/filter parallelism requires communication of entire activation tensors, instead of only halo regions, so the communication overheads will be significantly greater for the large, 3D data we consider. In general, these prior approaches have not considered the extreme strong scaling regime required for training 3D CNNs. Further, with CosmoFlow, we have demonstrated that training on full-resolution input data actually produces better results.

Many approaches, such as pipelining [38]–[41] and micro-batching [42] are orthogonal to our 3D spatial partitioning. Others directly target memory pressure during training,

but perform additional computation, including gradient accumulation [43], out-of-core algorithms [44]–[46], and recomputation [47], [48].

I/O performance has also been a recent focus, particularly for data sets with many or large samples. This has typically involved optimizing I/O pipelines with data staging and asynchronous I/O [1], [3], [20], [49]–[52]. We build upon these to handle the case where I/O for even a single sample is a bottleneck by partitioning and scaling I/O spatially, adapting collective I/O techniques developed for large-scale parallel scientific workloads [53]–[55].

3D CNNs have been widely used for 3D volume datasets, including medical imagery [2], [16], [56] and video action recognition [57], [58]. These typically extend a 2D CNN architecture by replacing the 2D convolutions with 3D operations [2]. While 3D convolutional layers enjoy similar learning properties to 2D ones and are more parameter-efficient than fully-connected layers for extracting spatial features, the memory requirements have made them challenging to use [56]–[58]. In particular, this has limited many works to small mini-batches, low-resolution data, or CNN architecture tradeoffs [56], [58]. We demonstrate that tackling these problems enables further improvements in prediction accuracy.

VII. Conclusions

Parallel training of DNNs is now considered a common practice rather than an art thanks to the wide availability of parallelized software stacks for deep learning. However, as shown in this paper, addressing the computational requirements in applying deep learning to scientific problems on 3D data sets necessitates finer-grained scalable parallel algorithms both in compute and I/O. We demonstrated the ability to spatially partition the training over many GPU-accelerated HPC nodes, enabling the traditional strong scaling that other HPC applications enjoy: accelerated time to solution without a compromise in the quality of the learned model. Further, we have demonstrated this with two networks that differ significantly in task, architecture, and performance characteristics, so we expect our work to be broadly applicable. As a result, we have created a scalable framework that can tackle the 3D CNNs that are rapidly emerging at the forefront of scientific machine learning.

Another hypothesis of this work was that learning on full-resolution data would allow models to learn better representations of long-range features present in the data. Our work with CosmoFlow demonstrated the strength of this hypothesis, where the extended CosmoFlow model achieved an order-of-magnitude improvement in prediction quality while significantly reducing training time by exploiting a larger-scale system.

Acknowledgments

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-JRNL-812691). This research was supported by JSPS KAKENHI Grant Number JP18J22858, Japan and by the Exascale Computing Project (17-SC-20-SC). Experiments were performed at the Livermore Computing facility. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

[1] A. Mathuriya *et al.*, "CosmoFlow: Using deep learning to learn the universe at scale," in *SC*, 2018.
[2] Ö. Çiçek *et al.*, "3D U-Net: Learning dense volumetric segmentation from sparse annotation," in *MICCAI*, 2016.
[3] T. Kurth *et al.*, "Exascale deep learning for climate analytics," in *SC*, 2018.
[4] K. Duraisamy, Z. J. Zhang, and A. P. Singh, "New approaches in turbulence and transition modeling using data-driven techniques," in *53rd AIAA Aerospace Sciences Meeting*, 2015.
[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *NeurIPS*, 2012.
[6] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *ICML*, 2016.
[7] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, 2016.
[8] J. Devlin *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *NAACL*, 2019.
[9] N. S. Keskar *et al.*, "On large-batch training for deep learning: Generalization gap and sharp minima," in *ICLR*, 2017.
[10] B. Van Essen *et al.*, "LBANN: Livermore big artificial neural network HPC toolkit," in *MLHPC*, 2015.
[11] P. Goyal *et al.*, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *arXiv preprint arXiv:1706.02677*, 2017.
[12] T. Akiba *et al.*, "PFDet: 2nd Place Solution to Open Images Challenge 2018 Object Detection Track," *arXiv preprint arXiv:1809.00778*, 2018.
[13] M. Yamazaki *et al.*, "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds," *arXiv preprint arXiv:1903.12650*, 2019.
[14] National Energy Research Scientific Computing Center, "CosmoFlow datasets," https://portal.nersc.gov/project/m3363, 2019.
[15] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *MICCAI*, 2015.
[16] L. Hou *et al.*, "High resolution medical image analysis with spatial partitioning," 2019.
[17] P. Bilic *et al.*, "The liver tumor segmentation benchmark (LiTS)," *CoRR*, vol. abs/1901.04056, 2019.
[18] N. Dryden *et al.*, "Improving strong-scaling of CNN training by exploiting finer-grained parallelism," in *IPDPS*, 2019.
[19] ——, "Channel and filter parallelism for large-scale CNN training," in *SC*, 2019.
[20] S. A. Jacobs *et al.*, "Parallelizing training of deep generative models on massive scientific datasets," in *CLUSTER*, 2019.
[21] Lawrence Livermore National Laboratory, "Conduit," https://github.com/LLNL/conduit, 2019.
[22] S. Chetlur *et al.*, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759*, 2014.
[23] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *IJHPCA*, Feb. 2005.
[24] Y. Oyama *et al.*, "Predicting Statistics of Asynchronous SGD Parameters for a Large-Scale Distributed Deep Learning System on GPU Supercomputers," in *BigData*, 2016.
[25] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015.
[26] S. Ravanbakhsh *et al.*, "Estimating cosmological parameters from the dark matter distribution," in *ICML*, 2016.
[27] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.
[28] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.
[29] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CSUR*, vol. 52, no. 4, 2019.
[30] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[31] A. Coates *et al.*, "Deep learning with COTS HPC systems," in *ICML*, 2013.
[32] A. Gholami *et al.*, "Integrated model, batch and domain parallelism in training neural networks," in *SPAA*, 2018.
[33] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *MLSys*, 2019.
[34] J. Dean *et al.*, "Large scale distributed deep networks," in *NeurIPS*, 2012.
[35] T. M. Chilimbi *et al.*, "Project Adam: Building an efficient and scalable deep learning training system." in *OSDI*, vol. 14, 2014.
[36] N. Shazeer *et al.*, "Mesh-TensorFlow: Deep learning for supercomputers," in *NeurIPS*, 2018.
[37] P. Buchlovsky *et al.*, "TF-Replicator: Distributed machine learning for researchers," *arXiv preprint arXiv:1902.00465*, 2019.
[38] X. Chen *et al.*, "Pipelined back-propagation for context-dependent deep neural networks," in *INTERSPEECH*, 2012.
[39] Y. Li *et al.*, "Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training," in *NeurIPS*, 2018.
[40] Y. Huang *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *NeurIPS*, 2019.
[41] D. Narayanan *et al.*, "PipeDream: generalized pipeline parallelism for dnn training," in *SOSP*, 2019.
[42] Y. Oyama *et al.*, "Accelerating deep learning frameworks with micro-batches," in *CLUSTER*, 2018.
[43] Y. Ito, R. Matsumiya, and T. Endo, "ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity," in *Big Data*, 2017.
[44] M. Rhu *et al.*, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.
[45] C. Meng *et al.*, "Training deeper models by GPU memory optimization on TensorFlow," in *ML Systems Workshop @ NeurIPS*, 2017.
[46] L. Wang *et al.*, "Superneurons: dynamic GPU memory management for training deep neural networks," in *PPoPP*, 2018.
[47] T. Chen *et al.*, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.
[48] P. Jain *et al.*, "Checkmate: Breaking the memory wall with optimal tensor rematerialization," in *MLSys*, 2020.
[49] S. Pumma *et al.*, "Towards scalable deep learning via I/O analysis and optimization," in *HPCC*, 2017.
[50] S. W. Chien *et al.*, "Characterizing deep-learning I/O workloads in TensorFlow," in *PDSW-DISCS*, 2018.
[51] Y. Zhu *et al.*, "Entropy-aware I/O pipelining for large-scale deep learning on HPC systems," in *MASCOTS*, 2018.
[52] F. Chowdhury *et al.*, "I/O characterization and performance evaluation of BeeGFS for deep learning," in *ICPP*, 2019.
[53] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999.
[54] Jianwei Li *et al.*, "Parallel netCDF: A high-performance scientific I/O interface," in *SC*, 2003.
[55] M. Howison *et al.*, "Tuning hdf5 for lustre file systems," in *IASDS10*, 2010.
[56] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-Net: Fully convolutional neural networks for volumetric medical image segmentation," in *3DV*, 2016.
[57] D. Tran *et al.*, "Learning spatiotemporal features with 3D convolutional networks," in *ICCV*, 2015.
[58] J. Carreira and A. Zisserman, "Quo vadis, action recognition? A new model and the kinetics dataset," in *CVPR*, 2017.