IPSJ SIG Technical Report

Vol.2019-HPC-170 No.8
2019/7/24

Toward Training a Large 3D Cosmological CNN with Hybrid Parallelization

(Unrefereed Workshop Manuscript)

Yosuke Oyama[1,2,a] Naoya Maruyama[2,b] Nikoli Dryden[3,2,c] Peter Harrington[4,d]
Jan Balewski[4,e] Satoshi Matsuoka[5,1,f] Marc Snir[3,g] Peter Nugent[4,h] Brian Van Essen[2,i]

Abstract: We report our preliminary work on large-scale training of a 3D convolutional neural network model for cosmological analyses of dark matter distributions. Previous work showed promising results for predicting cosmological parameters using CNNs trained on a large-scale parallel computing platform. However, due to its weak scaling nature, there exists a trade-off of training performance and prediction accuracy. This paper extends the existing work for better prediction accuracy and performance by exploiting finer-grained parallelism in distributed convolutions. We show significant improvements using the latest complex cosmological dataset with a huge model that was previously unfeasible due to its memory pressure. We achieve 1.42 PFlop/s on a single training task with a mini-batch size of 128 by using 512 Tesla V100 GPUs. Our results imply that the state-of-the-art deep learning case study can be further advanced with HPC-based algorithms.

1. Introduction

Training of Convolutional Neural Networks (CNNs) has been drastically accelerated by exploiting data-parallelism in the last decade. "Data-parallel" training in the context of Deep Learning (DL) means that each processor, such as a CPU or a GPU, 1) holds the same copy of the network parameters and 2) computes their gradients with respect to a specific set of data samples, and 3) performs collective aggregation to update the parameters. Since it only requires coarse-grained inter-processor communication and has near-perfect load balance, data-parallel training has been exploited to train many types of CNNs, such as networks for image recognition [1], [2], [3], [4], [5], action recognition [6], [7] and physical parameter prediction [8].

However, in extreme-scale training, model-parallelism is also required for various reasons. In model-parallel (or "hybrid-parallel" where both strategies are used at the same time) training, a single independent model is split into multiple processors, incurring fine-grained communication to exchange activations and errors in the middle of the network. One of the important advantages of model-parallelism over data-parallelism is that it relives memory requirements for each processor. Hence it enables one to train a bigger model than what can be trained under the data-parallel scheme. We demonstrate that our framework can train a network which cannot fit into the memory of a single GPU even if the batch size is one. Another benefit of model-parallelism is that it can increase the amount of parallelisms under a fixed mini-batch size constraint. It has been reported that a huge mini-batch size causes considerable accuracy drop [1], [8], which implies that the mini-batch size limits the maximum amount of parallelisms in data-parallel training. By using hybrid-parallelism, however, the amount of parallelisms can be larger than the mini-batch size; therefore, it can bring additional speedups.

In this paper, we demonstrate the scaling of the CosmoFlow network [8], originally proposed in previous work [9], a 3D CNN to predict cosmological parameters from 3D mass distribution, using hybrid parallelism in order to utilize 128 GPUs and 64 times larger sample size. We extend our previous work, the Livermore Big Artificial Neural Network Toolkit (LBANN) [10], Distconv [11] and Aluminum [12] to perform hybrid-parallel training of multi-dimensional convolutional neural networks for GPU clusters. Our framework enables users to use the arbitrary model partitioning of the sample and of spatial dimensions for each layer, enabling flexible and efficient load balancing among GPUs.

[1] Tokyo Institute of Technology, Tokyo, Japan
[2] Lawrence Livermore National Laboratory, California, USA
[3] University of Illinois at Urbana-Champaign, Illinois, USA
[4] Lawrence Berkeley National Laboratory, Illinois, USA
[5] RIKEN Center for Computational Science, Hyogo, Japan
[a] oyama.y.aa@m.titech.ac.jp
[b] maruyama3@llnl.gov
[c] dryden2@illinois.edu
[d] PHarrington@lbl.gov
[e] balewski@lbl.gov
[f] matsu@acm.org
[g] snir@illinois.edu
[h] penugent@lbl.gov
[i] vanessen1@llnl.gov

2. Background

2.1 The methodology of data-/model-/hybrid-parallelism

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

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

where $W^{(t)}$ are the network parameters on the step t, $\eta^{(t)}$ is the learning rate on the step t, N is the mini-batch size, E_n is the loss function with respect to the n-th sample of the mini-batch, and x_n the input data of the n-th sample.

2.1.1 Data-parallelism

Since there is no dependency between the computation of ∇E_n with different data samples, a most straightforward way to parallelize mini-batch SGD is to parallelize the computation of ∇E_n with different GPUs, and then aggregate the gradients by performing an all-reduce collective operation. This technique, called "data-parallel" training (**Fig. 1**, left), matches with the fact that N is typically big enough to parallelize on hundreds or thousands of GPUs, and the bandwidth requirement of performing the collective operations (O(100 MB)) is smaller than computation requirement especially for CNNs. Therefore, training of CNNs has been scaled to thousands of GPUs by simply exploiting data-parallelism [1], [2], [3], [8].

One of the limitations of data-parallelism is, however, the number of processes does not scale beyond the mini-batch size N. As it has been reported that too large a mini-batch size degrades inference accuracy of trained networks [1], [8], there is an inevitable parallelizing limit in data-parallel training.

2.1.2 Model-parallelism

On the other hand, in "model-parallel" training (Fig. 1, right), the computation of ∇E_n for each data sample is parallelized across multiple GPUs. One of the advantages of model-parallelism is that it parallelizes the computation of the loss function with one data sample with multiple GPUs; hence the amount of parallelisms can be set to one larger than the mini-batch size.

Another benefit of model-parallelism is that it relieves the GPU memory requirement by increasing the number of GPUs per sample. In data-parallel training, if the memory requirement for one data sample exceeds the memory capacity of a GPU, training cannot be started. On the contrary, in model-parallel training, the memory requirement is nearly in inverse proportion to the number of model-parallelisms. Therefore, training is feasible when a sufficient number of GPUs are available. This advantage is attractive especially for high-dimensional CNNs, which require massive GPU capacity to store high-dimensional tensors relative to conventional 2D CNNs.

At the same time, model-parallel training requires careful design of the underlying deep learning framework, to mitigate the overhead of layer-wise communication. While data-parallel training only requires a single global collective per iteration to aggregate parameter gradients, model-parallel training incurs many numbers of communication operations, and the operations have to be synchronized with the computation in every layer. Therefore, model-parallelism has been less studied than data-parallelism.

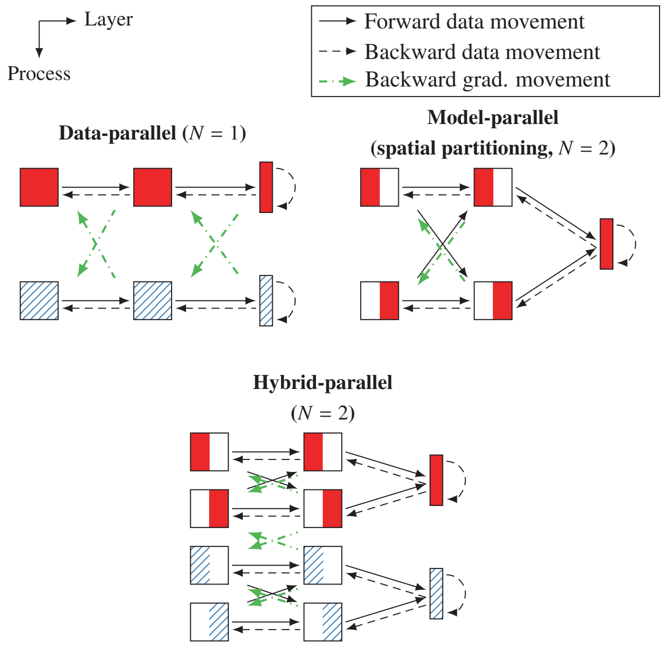

Fig. 1 Three different parallel strategies for deep neural networks. N denotes the mini-batch size. Note that data movement within a single process does not cost in most cases.

For CNNs, there are several strategies to exploit model-parallelism:

- *Spatial partitioning*: Since most of the operations for CNN type layer are layer-wise operations (such as activation functions) or only require neighbor data to compute each output element (such as convolution and pooling), a network can be partitioned in spatial dimension(s) into multiple GPUs without excessive inter-GPU communication.
- *Channel partitioning*: Each layer of a CNN has multiple "channels", disjointed tensors which are involved only in convolution or in the first fully-connected layer. As the typical number of channels in recent CNNs [8], [13] is much larger than the number of available GPUs on a GPU cluster, and complexity of each convolution channels are the same, it is convenient to exploit model-parallelism in the channel dimension. One of the problems is, however, it requires global all-to-all communication to exchange each distinct portion of channels among GPUs.
- *Layer-wise partitioning*: The computation of ∇E_n is composed of hundreds of kernels as well as its derivative, each of which is derived from each layer of the network. Thus, pipelining techniques with multiple GPUs can be applied to the computation.

In this paper, we focus on **spatial partitioning**, as it brings better communication-to-computation ratio than channel partitioning and always brings better load-balance than layer-wise partitioning.

2.1.3 Hybrid-parallelism

"Hybrid-parallelism" is the combination of data-parallelism and model-parallelism. The benefit of exploiting hybrid-parallelism is that it can achieve a high degree of parallelism by exploiting data-parallelism, and at the same time, each data sample is parallelized by using model-parallelism so that larger models can be trained with more computing resources. It requires careful design of the amount of parallelisms for different dimensions to sustain good scalability, as demonstrated in Section 5.3.

2.2 Distconv

Distconv [11] is a hybrid-parallel kernel library designed for CNNs. Distconv is currently supported by LBANN.

The basic concept of Distconv follows parallelized stencil computations. First, convolution is performed to the center part of an input tensor, and at the same time a halo exchange is started in a different asynchronous CUDA stream among GPUs in the same sample group. We repeat the one-dimensional halo exchange three times to perform the three-dimensional halo exchange. Once halo exchange is completed, convolution is performed on the halo region in the stream. We apply the same concept for backward-data and backward-filter kernels since these kernels are conceptually convolution operations.

When hybrid-parallel training is enabled, we simply combine the model-parallel implementation to the data-parallel training mechanism. In forward computation, no additional communication is introduced as data-parallel training only requires all-reduce collectives in backward passes. In backward passes, once a backward-filter kernel is completed, parameter gradients aggregation is started in another asynchronous stream. As convolution channels are not parallelized in this paper, all of the GPUs are involved in each all-reduce collectives.

2.3 CosmoFlow

CosmoFlow [8] is a project to estimate the values of important cosmological parameters from 3-dimensional universe data by using deep learning. In the previous work, the authors first conduct thousands of independent N-body simulations with varied initial cosmological parameters, and then constructed CNN training data set allowing regression of 3D mass distribution to ground truth parameters. Due to this, in the dataset we use in this paper, the distribution of the parameters is normalized to the $[-1, 1]$ uniform distribution in advance. In the paper, the authors demonstrated that a CNN is capable of predicting 3 such parameters with reasonable accuracy.

The original CosmoFlow network is composed of seven 3D convolutional layers, including three average-pooling layers, followed by three fully-connected layers with dropout. It outputs three scalar variables at the end. Its input size is 128^3 in 32-bit floats, and the network contains 7 M free parameters and requires 69.33 GFlops to compute parameter gradients from one sample.

In the original paper, it was reported that training with the mini-batch size of 8,192 did not converge to the same rate as training with the mini-batch size of 2,048, possibly due to a decrease of stochasticity of mean parameter gradients. Hence it is essential to introduce model- and hybrid-parallelism to keep the mini-batch size from such statistical limitations under large-scale training environments.

In this paper, we mainly use the "4parE" dataset [14]. It contains 1,027 universes each of which is composed of four channels, 512^3 voxels in the double-precision floating-point format, along with four cosmological parameters which were used to generate each universe. Each universe is split into four NumPy files in a subdirectory. The dataset size is $1027 \times 8 \times 4 \times 512^3 \sim 4$ TB in total.

We synthesize two datasets of $64 \times 4 \times 128^3$ and $8 \times 4 \times 256^3$ cubes, where 4 is the number of channels (C), by splitting each 4×512^3 cubes from the original dataset into subvolumes. The intuition motivating this technique is that the simulated universes are sufficiently uniform even in their smaller sub-regions to infer their cosmological parameters. In this paper, we distinguish the datasets with the input width W. We split each dataset into 80%, 10%, and 10% as training, validation, and test datasets respectively.

3. Related work

The concept of model-parallelism has been studied since the rise of deep learning. One of the most well-known CNNs, AlexNet [15], was trained on two GeForce GTX580 GPUs, each of which has 3 GB of memory. The main reason to introduce model-parallelism for the network was that the GPU memory size is relatively small for the network, so dividing a single network into multiple GPUs can relieve memory pressure for each GPU.

Thanks to advances in GPU memory technology, the memory size per GPU has been increased to more than 10 GB in recent years. Despite this improvement, however, the memory requirement to store a single DNN has also been enlarged by increasing the number of layers and the number of channels of each layer, to improve the inference performance of the models. For instance, most of the competition-wining CNNs [13], [15], [16] in the ILSVRC [17] image recognition competition use the mini-batch size of hundreds on one or a small number of GPUs. These networks illustrate the fact that the increase of GPU memory capacity does not always satisfy the demand for rapidly-advancing DNNs.

3.1 Deep convolutional neural networks for 3D datasets

3D convolutional neural networks have been proposed for 3D volume datasets such as cosmological [8], medical image [4], [5], and action recognition [6], [7] datasets. The most straightforward way to design a 3D CNN is to replace all of the 2D operations in a known CNN for 2D images with 3D operations [5].

Similar to 2D convolutional layers, it is regarded that 3D convolutional layers are capable of extracting the spatial features of input data with a much smaller number of parameters and floating-point operations than fully-connected layers. On the other hand, the memory size to store a 3D CNN is $O(n^3)$ while a 2D CNN is $O(n^2)$, which makes it challenging to use a big mini-batch size or to increase the resolution of input data in the data-parallel fashion [4], [6], [7]. It is also reported that careful design of size-reducing operations such as pooling layers is needed to reduce the memory footprint of 3D CNNs [4], [7]. We discuss the memory issue and the redesign of the CosmoFlow network in Section 4.3.

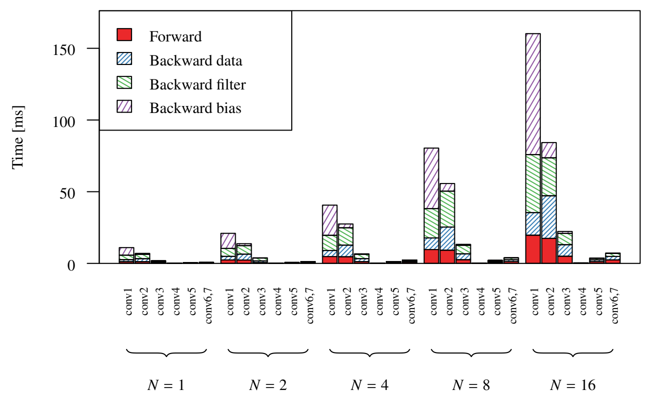

Fig. 2 Time to perform convolution kernels of the CosmoFlow network for the CosmoFlow dataset on a Tesla V100 GPU. We use the mini-batch size (N) from 1 to 16. We use cuDNN 7.5.0.

In this paper, we break the memory limitation problem of 3D CNNs in the existing work by building a flexible hybrid-parallel training software. To best of our knowledge, our work is the first attempt to train a 3D CNN with the CosmoFlow dataset whose input size is 512^3 voxels without partitioning the data samples.

4. Implementation details

In this section, we introduce the details of our implementation as well as some technical difficulties and discuss how to perform large-scale training of 3D CNNs.

4.1 The notation of tensor dimensions

In this paper, we adopt cuDNN's notation of N, C, D, H and W, which means samples, channels, depth, height, and width respectively. Unless it is explicitly mentioned, we assume all of the tensors are "fully-packed", whose width-, height-, depth-, channels-, samples-strides are products of widths of previous dimensions. Specifically, in this paper, we omit C as we do not split the channel dimension among GPUs as explained in Section 1. We also adopt the notation of $\{D_p, H_p\}$ to represent the amount of parallelisms for depth- and height-dimensions. We also omit N when the amount of parallelisms is the quotient of the number of GPUs and the product of other dimensions. For example, if the total number of GPUs is 16, $\{2, 1\}$ means there are $16/(2 \times 1) = 8$ groups each of which split a data sample into two GPUs in the depth dimension.

4.2 Extending Distconv for 3D CNNs

All of the spatial sizes of layers of a network cannot necessarily be divided by the number of processes. For instance, in many CNNs, the spatial size of each layer decreases by applying pooling layers, because this technique is considered to efficiently summarize spatial features into smaller domains. In Distconv, if a spatial size is not a multiple of the number of processes, we gather valid data to a part of the process. Even though this technique makes some GPUs idle during the computation of the latter part of a CNN, this does not spoil the performance of GPUs for 3D CNNs, as the former part of a 3D CNN has much greater complexity than others (**Fig. 2**, **Table 1**).

Table 1 The number of operations (in GFlops) of each convolutional layer for the CosmoFlow dataset We change the padding width to 1.

Name	Forward	Backward data	Backward filter
conv1	1.81	1.81	1.81
conv2	7.25	7.25	7.25
conv3	3.62	3.62	3.62
conv4	0.03	0.03	0.03
conv5	0.91	0.91	0.91
conv6,7	1.81	1.81	1.81

4.3 Network

Table 2 shows the CosmoFlow network architecture we use. We have applied the following changes to the original network [8]:

- **Adding pooling layer(s)**: We add up to two additional pooling layers, "pool6" and "pool7", to keep the output size of the last convolutional layer, "conv7". Since a large portion of the number of parameters of a CNN comes from its first fully-connected layer, this change fixes the number of parameters regardless of the input size.
- **Applying "same" padding with odd-size filters**: We add padding width of 1 for all convolutional and pooling layers which have 3^3 filters. This "same" padding policy prevents the output sizes from being reduced more than their corresponding input sizes, hence each layer size is always a power of two. Therefore, this change enables ideal model partitioning for any number of layers and any number of processes. For the same reason, we change the filter sizes to 3^3.
- **Removing biases from convolutional layers**: The backward bias kernel of convolutional layers computes the sum of elements of each feature map across all of the

Table 2 CosmoFlow network architecture. Unless it is explicitly mentioned, we use stride width of 1 for convolutional layers and 2 for pooling layers. We use padding width of 1 for all the layers.

Layer		Output size		
Name	**Filters / Weights**	$W = 128$	$W = 256$	$W = 512$
conv1	16×3^3	128^3	256^3	512^3
pool1	16×3^3	64^3	128^3	256^3
conv2	32×3^3	64^3	128^3	256^3
pool2	32×3^3	32^3	64^3	128^3
conv3	64×3^3	32^3	64^3	128^3
pool3	64×3^3	16^3	32^3	64^3
conv4	128×3^3 (stride of 2)	8^3	16^3	32^3
pool4	128×3^3	4^3	8^3	16^3
conv5	256×3^3	4^3	8^3	16^3
pool5	256×3^3	2^3	4^3	8^3
conv6	256×3^3	2^3	4^3	8^3
pool6	256×3^3	N/A	2^3	4^3
conv7	256×3^3	2^3	2^3	4^3
pool7	256×3^3	N/A	N/A	2^3
fc1	2048×2048	2048	2048	2048
fc2	2048×256	256	256	256
fc3	256×4	4	4	4
# conv. ops. [GFlops/sample]		55.55	443.8	3550
(Forward)		18.52	147.9	1183
Memory [GiB/sample]		0.824	6.59	52.7
# parameters $[10^6]$		9.44	9.44	9.44

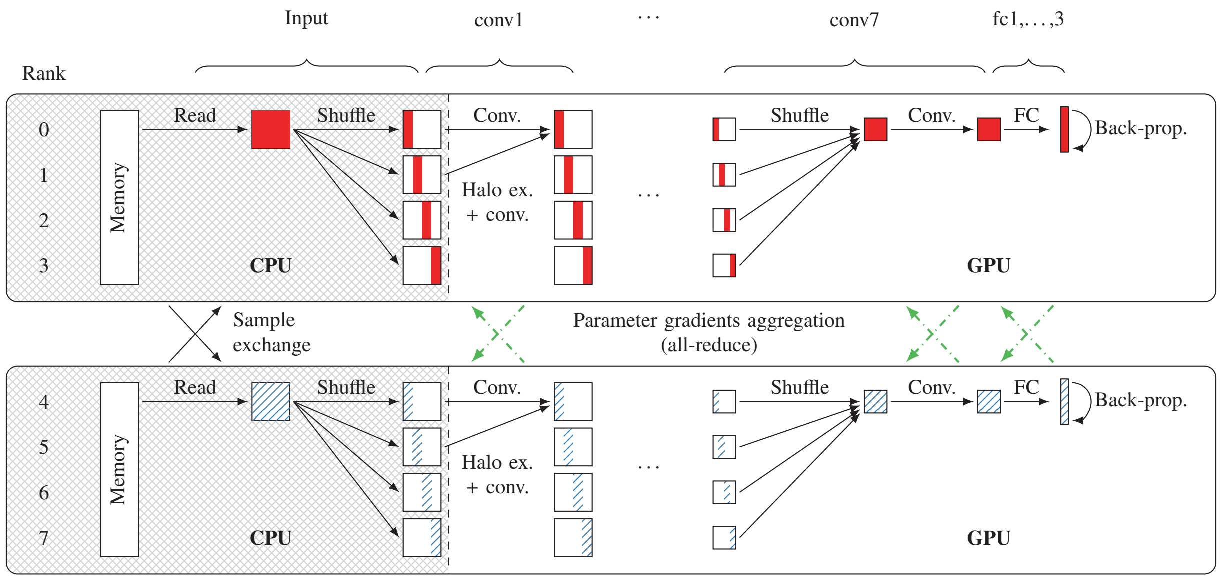

Fig. 3 Overview of hybrid-parallel LBANN training. Each node contains four processes which partition a single data sample.

samples of the input tensor. In cuDNN [18] 7.5.0, however, we found that the kernel takes considerably more time than other backward kernels, even though its complexity is much smaller than others (Fig. 2). Therefore, we remove biases from convolutional layers to eliminate this inefficiency. As seen in **Table 3**, we do not observe any accuracy degradation by removing the biases.

The rest of the details of the network follow the original model: We apply leaky ReLU after every convolutional and fully-connected layer except for the last layer, apply dropout before every fully-connect layer with the keep probability of 0.8, and adopt the mean squared error as the loss function. We use the Adam [19] optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. We perform exhaustive search for the initial learning rate α for each network.

Table 3 summarizes preliminary training results of our CosmoFlow network variant using the "2parB" dataset [14]. In Table 3, our network achieves better accuracy than the baseline model, and hence we use this network throughout this paper. In addition, the test loss of the original network is degraded by increasing the mini-batch size, even if the learning rate is increased in the same proportion to the mini-batch size to keep the stochasticity of parameter gradients [1], [8], as shown in the previous work [8]. This result implies that there is a hidden scaling limit for this network in the data-parallel scheme around hundreds of GPUs. Our hybrid-parallel scheme can accelerate training by using hundreds of GPUs while keeping the same mini-batch size in Section 5.2.

Table 3 Training results of our CosmoFlow network variants. We train each network for 130 epochs.

Network	Number of nodes	N	α	Minimum loss	
				Training	Test
Original	8	512	0.0001	0.0041	0.0051
Original	32	2,048	0.0002	0.0110	0.0077
Original	128	8,192	0.0005	0.0241	0.0184
Proposal	32	1,024	0.0002	0.0062	0.0041

As shown in Table 2, the per-sample memory requirements for the $W = 128$ and $W = 256$ are under the memory size of the latest GPUs: For instance, an NVIDIA Tesla V100 GPU has 16 GB memory which is capable of holding one or more samples of the datasets. Thus, data-parallel training is the most efficient way to train the networks as it only requires a global collective communication in an iteration. For $W = 512$, however, the memory requirement exceeds the memory size. Hence it is not feasible to perform data-parallel training. Our framework resolves this problem by introducing hybrid-parallel scheme: LBANN successfully train the network by splitting it among 4 or more distinct GPUs in the "depth" (D) dimension or the combination of D and "height" (H) dimensions. This ability has a significant advantage for high-dimensional CNNs as their memory requirements are cubic of dimension size.

4.4 I/O performance optimization

We study the performance of two different data sample readers: The "Direct" data reader loads data samples from the file system directly, or from a node-local SSD where the entire dataset is preloaded in advance of training. The "Conduit" data reader uses Conduit [20] as an I/O backend. Conduit is an open source data exchange library that provides efficient ways of exchanging scientific data between applications, exchanging data between different processes within a single MPI-based application, and managing in-memory data movement within a single process. Our Conduit data reader preloads the entire dataset from the file system into CPU memory before training starts. The process that performs the read thereafter "owns" the data. Subsequently, prior to each minibatch, we employ an MPI-based data exchange to shuffle the data to the process that requires it. After a mini-batch is loaded, the MPI processes perform a data shuffle operation to exchange the desired spatial parts of the mini-batch.

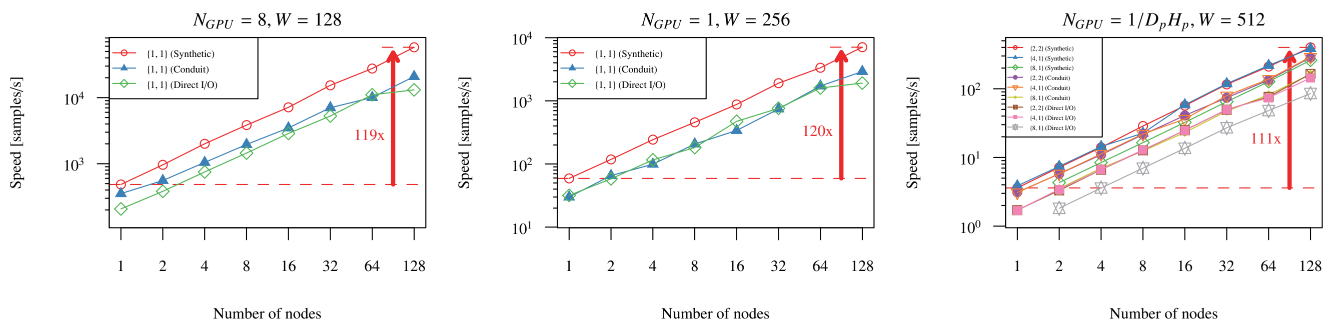

Fig. 4 Weak scaling of the CosmoFlow network. We use GPU batch sizes of 8 and 1 for the $W = 128$ and $W = 256$ datasets, and 1 for each GPU group for the $W = 512$ dataset respectively. We perform several epochs of training and use the minimum iteration time of the last epoch.

5. Evaluation

5.1 Evaluation environment

We use Lassen, a GPU cluster at Lawrence Livermore National Laboratory which is composed of 666 computing nodes. Each node has two IBM Power9 CPU chips with 256 GB memory and four NVIDIA Tesla V100 GPUs with 16 GB memory and NVLink links. GPUs on each node achieve 56 single-precision TFlop/s. Lassen adopts 6 NVLink links between GPU-GPU and GPU-CPU with 300 GB/s total bandwidth, and 100 Gb/s EDR InfiniBand among computing nodes. Each node is equipped with a 1.6 TB NVMe SSD.

We use GCC 7.3.1, CUDA 9.2, cuDNN [18] 7.5.0, NCCL 2.4.2 and IBM Spectrum MPI 10.2.0.11rtm2. All experiments are performed on Red Hat Enterprise Linux Server 7.5. We use the single-precision format for computation and data storing throughout the experiments.

5.2 Weak scaling

Fig. 4 shows the weak scaling of our implementation with three different datasets.

For the $W = 128$ and $W = 256$ configurations, we use GPU batch sizes of 8 and 1 respectively. For $W = 512$, we use a per-node batch size of 1, and split each data sample evenly in each node. We run the framework for few epochs with a subset (8192, 1024, and 128 samples from $W = 128$, $W = 256$, and $W = 512$ respectively) of each dataset, and show the minimum iteration time of the last epoch. We use the full dataset on 128 nodes for better stability. In this experiment, we also measure the performance with a "Synthetic" dataset configuration, where the I/O process is skipped so that the pure computation and communication performance is measured.

In all of the cases, our implementation achieves nearly linear speedup up to 128 compute nodes. We achieve 119x and 120x of speedup on 128 nodes compared to 1 node with the $W = 128$ and $W = 256$ datasets respectively. With these datasets, this result is convincing as the network has a high computation (Flops) to communication (the number of parameters) ratio compared to other conventional 2D CNNs which are used in recent large-scaling precedents. In fact, our variant of the CosmoFlow network has 9.44 M parameters and requires 18.5 GFlops for forward computation, while ResNet-152 [13] has O(10 M) parameters and 1.13 GFlops for computation. This characteristic makes it easy to scale in data-parallel training.

With $W = 512$, however, it is infeasible to perform data-parallel training since the model is too huge to fit into GPU memory as shown in Table 2. With our implementation, however, we achieve 111x of speedup over 1 node by exploiting hybrid-parallelism (even if layer-wise communication is introduced). In the experiment with the $W = 512$ dataset, we use the minimum number of nodes for the batch size of one, and then increase the number of nodes in weak scaling fashion. Thus, on 128 nodes, we use a mini-batch size of 128 for {4, 1} and

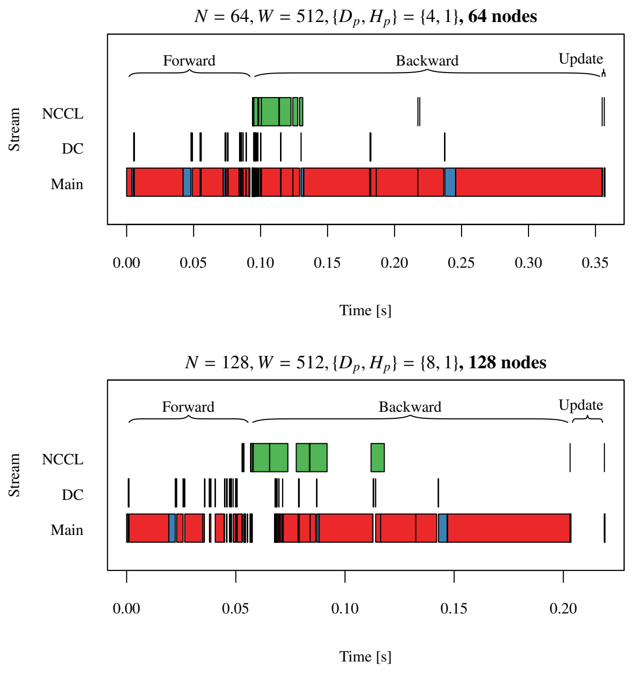

Fig. 5 Visualization of the GPU kernel timeline of a single iteration of training on two different weak scale configurations for the $W = 512$ dataset on 64 nodes. The "Main" stream orchestrates the main computation kernels the "DC" (Distconv) and "NCCL" streams are used for asynchronous communication. We merge NCCL's multiple CUDA streams to a single stream for simplicity as each GPU kernel do not overlap to each other.

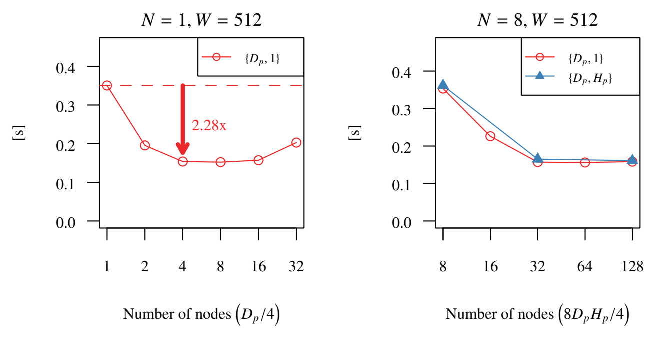

Fig. 6 Strong scaling of the CosmoFlow network. We use global mini-batch sizes (N) of 1 or 8 for the $W = 512$ dataset.

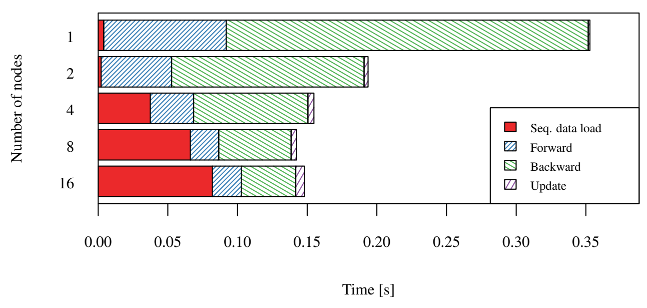

Fig. 7 Breakdown of the strong scaling experiment with $N = 1$. "seq. I/O" represents time to perform asynchronous data scatter from the root process to other processes that cannot be hidden by computation time.

{2, 2}, but 64 for {8, 1}. Even though this configuration degrades per-sample computation efficiency for {8, 1}, it also introduces the possibility of parallelizing the computation on each data sample among more GPUs. Indeed, when the mini-batch size is 64, the computation speed with {4, 1} (on 64 nodes) is 218.3 samples/s, while it is 260.0 samples/s with {8, 1} (1.19x of {4, 1}), even if inter-node layer-wise communication is required.

It achieves 1.42 PFlop/s on 128 nodes with the {2, 2} configuration and the synthetic data reader, and 289 TFlop/s without the synthetic data reader. The former achieves 20% of the peak computational performance.

With the $W = 256$ dataset, the slowdown of the CosmoFlow data reader and the Conduit data reader are 2.43x more than the synthetic data reader respectively on 128 nodes. In this configuration, the data reader loads $256^3 \times 4 \times 2 \times 4 = 512$ MiB of data from each SSD in each iteration, but in both data readers, data samples have to go through a PCIe bus or InfiniBand, whose one-side bandwidth is 32 GB/s or 2×12.5 GB/s respectively. In addition, it is required to perform a memory copy multiple times in or between CPUs and GPUs to perform the data shuffle and type conversion. Even if Conduit is introduced to eliminate I/O, the data reader still needs to perform inter-node communication as the entire dataset does not fit into the CPU memory of a single node. Therefore, the data load takes O(10 ms) while the iteration time of the synthetic data reader is around 70 ms.

Fig. 5 shows the GPU kernel timeline of a training iteration for each configuration. Distconv invokes packing/unpacking and P2P communication kernels on the main stream as well as on its dedicated stream asynchronously, as mentioned in Section 2.2. From the beginning of back-propagation, NCCL starts to communicate computed parameter gradients among processes asynchronously to the main (computation) stream. Note that there are explicit barriers to preserve the correctness of the computation, which are not shown in the figure. For instance, the reason why the main stream is idle at the end of the last two timelines for around 20 ms is that it has to wait for NCCL to complete gradient aggregation to update the weights.

As shown in Fig. 4 and Fig. 5, there are two clear reasons why {8, 1} on 64 nodes achieves less computation speed than {4, 1}:

- Since the batch size per GPU is halved, the computation efficiency per sample is degraded. The former takes about 200 ms in the main stream while the latter takes about 350 ms.
- Due to this computational inefficiency, part of the all-reduce cannot be hidden in the main stream, especially at communication for the fully-connected layers whose communication intensity per operations is much higher than convolutional layers, and at the end of back-propagation where all communication have to be completed before updating the weights.

This communication inefficiency is, however, not the main bottleneck of scaling from 64 nodes to 128 nodes, as seen in the middle and the bottom of Fig. 5. It is also observed that the overhead introduced by Distconv (colored in blue) is nearly negligible compared to the computational kernels (red) in any configurations in Fig. 5.

5.3 Strong scaling

Fig. 6 shows the strong scaling performance of our implementation with the $W = 512$ dataset. We use global mini-batch sizes of 1 or 8. We use the same methodology in Section 5.2 to measure the performance. $\{D_p, 1\}$ in the figure represents that the entire network is divided on the number of processes in the "depth" dimension, and if possible, the samples are also parallelized among processes following the data-parallel fashion. $\{D_p, H_p\}$ means that we use the same amount of parallelisms ($D_p = H_p$) for both "depth" and "height" dimensions.

In Fig. 6, even when the mini-batch size is one, it achieves 2.28x of speed up on 4 nodes (16 GPUs) compared to one node. In this experiment, the scalability limit is 8 GPUs, and the main bottleneck is input data loading, as shown in **Fig. 7**. As we do not adopt splitting each data sample in advance, this is unavoidable overhead in the current implementation. Similarly, it achieves 2.25x of speedup on 32 nodes (128 GPUs) when the mini-batch size is 8.

6. Conclusions

In this paper, we demonstrated that our framework successfully accelerates training of the CosmoFlow network by introducing hybrid-parallelism, achieving 171 TFlop/s on 128 Tesla V100 GPUs. Our experimental results showed the performance possibility of hybrid-parallelism for high-dimensional convolutional neural networks.

This case study implies the possibility that training of high-dimensional CNNs can be accelerated by exploiting model-parallelism on HPC infrastructure while the mini-batch size (i.e., the computation semantics) is unchanged. We also demonstrated that one of the main bottlenecks of such training is I/O, as the data size is O(n^3). To relieve this overhead, I/O has to be also parallelized among multiple nodes, so that each process loads the part of a single sample that is needed by its GPU.

Acknowledgment

This research was supported by JSPS KAKENHI Grant Number JP18J22858, Japan. This research was supported by the Exascale Computing Project (17-SC-20-SC), 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. LLNL-CONF-778764.

References

[1] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y. and He, K.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, *arXiv e-prints*, p. arXiv:1706.02677 (2017).

[2] Akiba, T., Kerola, T., Niitani, Y., Ogawa, T., Sano, S. and Suzuki, S.: PFDet: 2nd Place Solution to Open Images Challenge 2018 Object Detection Track, *arXiv e-prints*, p. arXiv:1809.00778 (2018).

[3] Yamazaki, M., Kasagi, A., Tabuchi, A., Honda, T., Miwa, M., Fukumoto, N., Tabaru, T., Ike, A. and Nakashima, K.: Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds, *arXiv e-prints*, p. arXiv:1903.12650 (2019).

[4] Milletari, F., Navab, N. and Ahmadi, S.-A.: V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation, *arXiv e-prints*, p. arXiv:1606.04797 (2016).

[5] Çiçek, Ö., Abdulkadir, A., Lienkamp, S., Brox, T. and Ronneberger, O.: 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation, *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, LNCS, Vol. 9901, Springer, pp. 424–432 (2016).

[6] Tran, D., Bourdev, L., Fergus, R., Torresani, L. and Paluri, M.: Learning Spatiotemporal Features with 3D Convolutional Networks, *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, Washington, DC, USA, IEEE Computer Society, pp. 4489–4497 (2015).

[7] Carreira, J. and Zisserman, A.: Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset, *arXiv e-prints*, p. arXiv:1705.07750 (2017).

[8] Mathuriya, A., Bard, D., Mendygral, P., Meadows, L., Arnemann, J., Shao, L., He, S., Kärnä, T., Moise, D., Pennycook, S. J., Maschhoff, K., Sewall, J., Kumar, N., Ho, S., Ringenburg, M. F., Prabhat and Lee, V.: CosmoFlow: Using Deep Learning to Learn the Universe at Scale, *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, Piscataway, NJ, USA, IEEE Press, pp. 65:1–65:11 (2018).

[9] Ravanbakhsh, S., Oliva, J., Fromenteau, S., Price, L. C., Ho, S., Schneider, J. and Póczos, B.: Estimating Cosmological Parameters from the Dark Matter Distribution, *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, JMLR.org, pp. 2407–2416 (2016).

[10] Van Essen, B., Kim, H., Pearce, R., Boakye, K. and Chen, B.: LBANN: Livermore Big Artificial Neural Network HPC Toolkit, *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, MLHPC '15, New York, NY, USA, ACM, pp. 5:1–5:6 (2015).

[11] Dryden, N., Maruyama, N., Benson, T., Moon, T., Snir, M. and Van Essen, B.: Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism, *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS '19 (2019).

[12] Dryden, N., Maruyama, N., Moon, T., Benson, T., Yoo, A., Snir, M. and Van Essen, B.: Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems, *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pp. 1–13 (2018).

[13] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778 (2016).

[14] National Energy Research Scientific Computing Center: CosmoFlow Datasets, https://portal.nersc.gov/project/m3363 (2019).

[15] Krizhevsky, A., Sutskever, I. and Geoffrey E., H.: ImageNet Classification with Deep Convolutional Neural Networks, *Advances in Neural Information Processing Systems 25 (NIPS2012)*, Vol. 86, No. 11, pp. 1–9 (2012).

[16] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *CoRR*, Vol. abs/1409.1, pp. 1–14 (2014).

[17] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C. and Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge, *International Journal of Computer Vision*, Vol. 115, No. 3, pp. 211–252 (2015).

[18] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B. and Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning, *arXiv e-prints*, p. arXiv:1410.0759 (2014).

[19] Kingma, D. P. and Ba, J.: Adam: A Method for Stochastic Optimization, *arXiv e-prints*, p. arXiv:1412.6980 (2014).

[20] Lawrence Livermore National Laboratory: Conduit, https://github.com/LLNL/conduit (2019).