Hy-Fi: Hybrid Five-Dimensional Parallel DNN Training on High-Performance GPU Clusters

Arpan Jain(✉), Aamir Shafi, Quentin Anthony, Pouya Kousha, Hari Subramoni, and Dhableswar K. Panda

The Ohio State University, Columbus, OH 43210, USA
{jain.575,shafi.16,anthony.301,kousha.2}@osu.edu,
{subramon,panda}@cse.ohio-state.edu

Abstract. Recent advances in High Performance Computing (HPC) enable Deep Learning (DL) models to achieve state-of-the-art performance by exploiting multiple processors. Data parallelism is a strategy that replicates the DL model on each processor, which is impossible for models like AmoebaNet on NVIDIA GPUs. Layer parallelism avoids this limitation by placing one or more layers on each GPU, but still cannot train models like AmoebaNet on high-resolution images. We propose Hy-Fi: Hybrid Five-Dimensional Parallelism; a system that takes advantage of five parallelism dimensions—data, model, spatial, pipeline, and bi-directional parallelism—which enables efficient distributed training of out-of-core models and layers. Hy-Fi also proposes communication-level optimizations to integrate these dimensions. We report up to 2.67× and 1.68× speedups over layer and pipeline parallelism, respectively. We demonstrate Hy-Fi on up to 2,048 GPUs on AmoebaNet and ResNet models. Further, we use Hy-Fi to enable DNN training on high-resolution images, including 8,192 × 8,192 and 16,384 × 16,384.

Keywords: DNN · Model-parallelism · Distributed training · Hybrid parallelism · MPI · GPU

1 Introduction

In the last decade, Deep Learning (DL) has emerged as a viable approach to practice Artificial Intelligence (AI) in emerging disciplines including object recognition/detection, speech recognition, language translation, and emotion recognition. A typical DL model is capable of "learning" non-linear mathematical relationships between the input data and the corresponding output during training on sufficiently large datasets—this knowledge is later used to make predictions with new and unseen data. One of the main driving forces behind the success of complex and large-scale Deep Neural Networks (DNNs) is the availability of compute resources offered by modern High Performance Computing (HPC) hardware. Current state-of-the-art models like AmoebaNet [22] and

A.-L. Varbanescu et al. (Eds.): ISC High Performance 2022, LNCS 13289, pp. 109–130, 2022.
https://doi.org/10.1007/978-3-031-07312-0_6

GPT3 have become complex and computationally expensive—due to a large number of parameters—and cannot be trained on a single processing element (for e.g. a GPU). This fundamental limitation on training state-of-the-art DNNs is resolved by exploiting parallel and distributed training on HPC hardware. One popular and easy-to-use parallelization strategy for distributed DNN training is *data parallelism* [1]. This technique accelerates training by creating model replicas on multiple processing elements—including GPUs—and performs DNN training by dividing the input data into multiple *batches*. Each processing element is assigned a unique set of data, called a *batch*, which is used to perform parallel training steps, this assignment is followed by a synchronization step using an allreduce operation. This step incurs communication overhead in order to accumulate the gradients across all processing elements and ensure weights are synchronized after each training step.

While data parallelism offers near-linear scaling [3,17] for distributed DNN training, it incurs significant memory overhead since the entire model is replicated on each GPU. It therefore requires the entire model to fit inside the memory of a single GPU, which is not always possible especially for emerging large models. Even at the finest granularity of a training sample, most of the current state-of-the-art models like AmoebaNet [22] and GPT3 models cannot fit inside the memory of a single GPU; these models are hence known as *out-of-core DNNs or models*. The memory requirement of a DNN depends on its size (number of parameters) and the size of the input image. For this reason, a DL model that is trainable on a single GPU for a small-sized images may not be trainable on a single GPU for high-resolution (large-sized) images. This means that due to the inability to store a DL model replica on a single GPU, the data parallelism approach is only limited to the training of modestly-sized DNNs on low-resolution images like Cifar-100 (32×32 pixels) and ImageNet (244×244 pixels).

However, in modern scientific applications, image sizes can range from 512×512 pixels to $2{,}048 \times 2{,}048$ pixels [6]. For example, the 2D mesh-tangling problem represents hydraulic simulation and can be formulated as semantic image segmentation. The input data in mesh-tangling can range from $1{,}024 \times 1{,}024$ pixels to $2{,}048 \times 2{,}048$ pixels. In digital pathology, the advent of high-resolution scanners has led to the adoption of digital whole slide imaging (WSI) for diagnostic purposes. A typical application of WSI is measuring the degree of a tumor grade for diseases such as cancer. The detection problem [19] can be formulated as a classification task in which the input is a WSI and output is the presence or absence of cancer. Normally, WSIs are very high-resolution images.

To address this fundamental limitation of data parallelism, *layer-parallelism*—also known as inter-layer *model-parallelism*—is proposed in the literature [5] to enable training of the out-of-core DNNs. Here, the DNN is divided into smaller partitions, each consisting of one or more layers, that can fit inside a single GPU's memory. This approach—referred to as *basic model-parallelism*—has inherent scaling issues [10,11]. The reason is as follows: A DNN essentially is a directed acyclic graph where each node corresponds to a layer. As part of the forward pass, each layer takes inputs/activations from the previous layer and gives output to the next layer making this an inherently sequential process. This data dependency

serializes parallel processing of layers in a DNN since only one GPU does the computation at any given time. In addition to basic model-parallelism, *pipeline-parallelism* [12] and *sub-graph parallelism* [13] are also instances of the inter-layer model-parallelism approach. A variant of model-parallelism is to exploit parallelism within layers. This approach is sometimes called intra-layer model-parallelism. Here, a single layer is divided across multiple GPUs. An instance of this technique is *spatial parallelism*, which partitions the images across multiple GPUs thereby distributing the layer. *Hybrid parallelism* combines data- and model-parallelism but also suffers from data dependency limitations.

1.1 Motivation

Several approaches have been proposed in the literature to address some of the limitations of model parallelism. However, most studies are performed for low-resolution images that exhibit different characteristics [12]. Compared to low resolution images, high-resolution images results in higher activation memory and larger tensors, which results in a larger communication overhead (Fig. 1).

These approaches include pipeline, spatial, and bi-directional parallelism. Pipeline parallelism such as the schemes proposed in [10,12,21] exploits parallelism within training samples and accelerates the performance of model parallelism. However, pipeline parallelism is only possible when the model is trainable with batch size >1, which is typically impossible with high-resolution images due to memory constraints. There have been efforts to exploit pipeline parallelism for large-sized images [14] but they still require a single layer to fit inside the GPU—such layers are called *out-of-core layers*. Spatial parallelism, on the other hand, has performance issues due to high communication overhead and the inability to accelerate low-resolution images that are common in the latter half of DNNs. Bi-directional parallelism exploits the memory and compute available between the backward and forward passes of the first and second training iterations. It trains the model from both directions in order to use these potentially-wasted resources [14]. Therefore, existing solutions [12,14] limit the ability to train DNNs on very high-resolution images, which affects the DNN's accuracy and prohibits the training of complex DNNs.

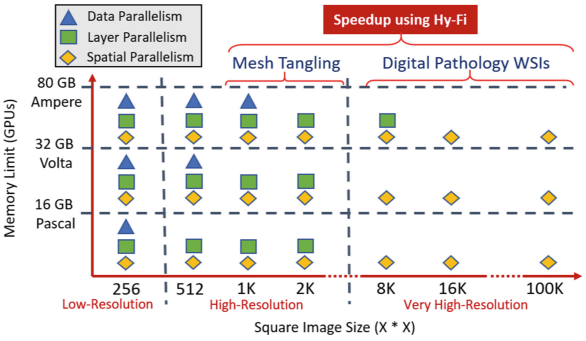


Fig. 1. The emerging need for integrated spatial and model parallelism solution as suggested in [14]

To summarize, spatial parallelism distributes images across multiple GPUs, layer parallelism distributes the model, pipeline parallelism parallelizes the

samples in a batch, and bi-directional parallelism employs memory-aware solutions to enhance the performance.

This paper focuses on efficiently utilizing distributed training for very high-resolution images that appear in real-world applications. These have unique requirements from the underlying DL training framework. Digital pathology, for example, uses a tiling mechanism to train Tall Cell Variant (TCV) classifiers on very high-resolution images, limiting the structural information and local/global context. Based on this, we seek to solve the following application-level requirements in this paper: 1) Enabling training on very high-resolution images, 2) Ability to train the model with any batch size on the same number of resources, and 3) provide speedup and support new emerging applications like TCV classifier. To address these requirements for training large-sized images, several design and system-level challenges need to be solved:

- How to efficiently integrate spatial, layer, pipeline, bi-directional, and data parallelism?
- How to reduce the communication overhead in an integrated distributed DNN training system?
- Can different distribution layouts improve the performance for spatial parallelism and reduce the communication overhead?
- How to enable the training of out-of-core models and out-of-core layers?
- Can the integration of spatial, layer, pipeline, data, and bi-directional parallelism achieve scalability similar to data parallelism?

Table 1. Features offered by Hy-Fi compared to existing frameworks

Studies	Features										
	Out-of-core model training (max batch size = 1)	Out-of-core layer training	Out-of-core batch size	Memory -aware solution	Pipelining support	Bidirectional training	Optimized communication for bi-directional training	Hybrid parallelism	Multi-node support	Speedup for out-of-core BS training	Speedup for CNN
Basic model parallelism (layer parallelism)	✔	×	×	×	×	×	N/A	×	✔	×	✔
Pipeline parallelism (Gpipe [12])	×	×	×	×	✔	×	N/A	✔	✔	×	✔
GEMS [14]	✔	×	✔	✔	✔	✔	×	✔	✔	✔	✔
TorchGipe [16]	✔	×	✔	✔	✔	×	N/A	×	×	✔	✔
PipeDream [10]	✔	×	×	×	✔	×	N/A	✔	✔	×	✔
LBANN (spatial [6]/domain parallelism [8])	✔	✔	×	×	×	×	N/A	✔	✔	×	✔
FlexFlow [15]	✔	✔	×	×	×	×	N/A	✔	✔	×	✔
Mesh-TensorFlow [23]	✔	✔	×	×	×	×	N/A	✔	✔	×	✔
Megatron [24]	✔	×	×	×	×	×	N/A	✔	✔	×	×
SUPER [13]	×	×	×	×	×	×	N/A	✔	✔	×	×
Proposed Hy-Fi	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

1.2 Contributions

In this section, we highlight the major contributions of this study. To the best of our knowledge, no state-of-the-art distributed DNN training system supports out-of-core models, layers, and batch size with memory-efficient designs. Table 1 compares related data, model, pipeline, and spatial parallelism studies against the proposed Hy-Fi system. Section 6 provides an in-depth comparison of related studies. Major contributions of this study are as follows:

- We propose, design, and evaluate Hy-Fi: an integrated memory-efficient system that uses five dimensions of parallelism and provides scalable training.
- We overcome the limitations of individual parallelization techniques—spatial, layer, and bi-directional parallelism—by proposing communication optimizations and efficiently integrating all five dimensions of parallelism (spatial, layer, pipeline, bi-directional, and data) to use in tandem.
- Hy-Fi offers up to 2.02× speedup over pure layer parallelism and 1.44× speedup over pure pipeline parallelism for the spatial parallelism dimension. Using memory-efficient bi-directional parallelism, we increase speedup to 2.67× over pure layer parallelism and 1.68× over pure pipeline parallelism.
- We show near-linear scaling (94.5%) for distributed DNN training using Hy-Fi on 2,048 Volta V100 GPUs.
- We enable the training of next-generation deep learning models on very high-resolution images ($8{,}192 \times 8{,}192$ and $16{,}384 \times 16{,}384$ pixels) and show up to 1.47× speedup over spatial parallelism.

2 Challenges in Exploiting Different Parallelism Dimensions in Distributed DNN Training

We highlight the challenges in implementing a multi-dimensional DNN training framework like Hy-Fi which has several communication optimizations and enables training on very high-resolution images.

Challenge-1: Halo Exchange in PyTorch
Spatial parallelism requires the halo exchange to implement distributed convolution and pooling layers. A halo exchange involves communication in different directions and differs in message size (Fig. 3). The message size and communication pattern depends on several parameters such as the kernel size, spatial parallelism size, partition position in the distributed image, and the number of neighbors, which exacerbates the challenge of implementing a halo exchange. A halo exchange can be implemented using non-blocking point-to-point communication provided by CUDA-aware MPI libraries, but they need to be synchronized with asynchronous execution in PyTorch [20] to ensure data validation.

Challenge-2: Exploitation of Different Parallelism Dimension
The proposed framework must be designed in a modular and user-transparent fashion to exploit different parallelism dimensions in tandem. Further, the system should be robust enough to take advantage of all parallelism dimensions

whenever possible. Spatial, layer, pipeline, bi-directional, and hybrid data parallelism offer compute parallelization in different dimensions and a potential to accelerate training of CNN. Hence, every parallelization dimension should be efficiently implemented and integrated with others in order to fully exploit the benefits of all the strategies in tandem.

Challenge-3: Scaling Integrated Hybrid Training Solutions
Training CNNs on high-resolution images is a compute-intensive task and requires a large numbers of GPUs to make it feasible. Hence, the proposed solution must be scalable to thousands of GPUs, which requires hybrid data parallelism. Integrating data parallelism into a multi-dimension parallelization framework like Hy-Fi is a non-trivial task since each parallelization dimension combines data parallelism differently. For example, spatial parallelism uses allreduce operations to synchronize weights across the distributed input, layer parallelism distributes the model and needs sub-communicators to implement hybrid parallelism, and bi-directional parallelism introduces extra replicas for data parallelism. Hence, designing a scalable solution exploiting multiple parallelism dimensions is a challenging task.

3 Limitations in Existing Approaches for Model Parallelism

We provide an overview of various existing model-parallelism approaches and discuss their limitations.

3.1 Layer Parallelism

The simplest model-parallelism scheme consists of placing DNN partitions (consisting of one or more *layers*) on separate GPUs before applying distributed forward and backward passes. These distributed forward and backward passes are implemented with simple Send and Recv operations. Layer parallelism has two primary drawbacks: 1) Under-utilization of resources and 2) A complex implementation compared to data parallelism. Given that only a single GPU can operate at once, layer parallelism suffers from poor scalability. Since DL frameworks do not provide distributed back-propagation implementations, layer parallelism is often challenging to implement. Manually partitioning a DNN is challenging in itself because not all layer connections preserve a simple ordering (e.g. skip or residual connections).

3.2 Pipeline Parallelism

Pipelining divides the input batch into smaller batches called micro-batches, the number of which we call *parts*. The goal of pipeline parallelism is to reduce under-utilization by overlapping micro-batches, which allows multiple GPUs to proceed with compute within the forward and backward passes. Pipeline parallelism has

two primary drawbacks; 1) batch size limits the length of the pipeline, and 2) performance is poor compared to data or hybrid Parallelism. Pipelining also wastes GPU resources when the pipeline is not full. The only case with a full pipeline occurs when the number of parts equals the number of DNN splits *and* the batch size equals the pipeline length. These issues worsen at scale due to the pipelining *bubble* [12]. Further, it is not possible to use pipelining when the largest batch size is 1. Due to the above limitations, there is a need to further optimize both layer and pipeline parallelism.

3.3 Memory-Aware Synchronized Training (Bi-directional Parallelism)

Both basic and pipelining model parallelism suffer from under-utilization of resources. After completing the forward and backward passes for a given model partition, each GPU has free memory and compute resources available, which can be utilized to perform the forward and backward passes of a new model. GEMS-MAST [14] uses this free memory and compute resources by training a replica of the same DNN in an *inverted* manner. This design is called GEMS-Master. We call this bi-directional parallelism in the rest of the paper.

4 Proposed Hybrid Five-Dimensional Parallelism System

4.1 Spatial Parallelism

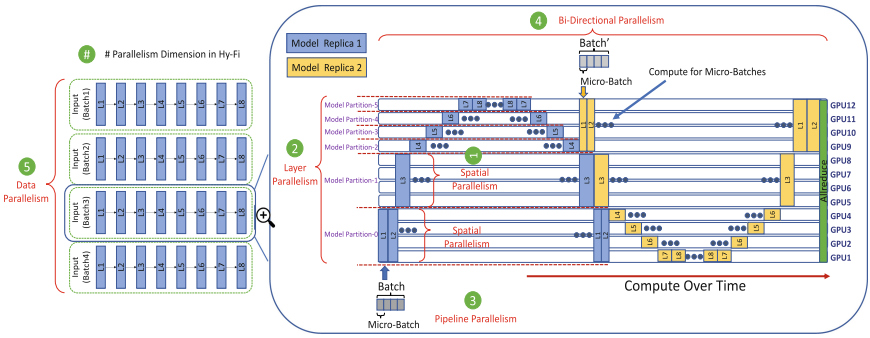


Fig. 2. High-level overview of proposed Hybrid Five-Dimensional Parallelism (Hy-Fi) where L# represents layer number. It shows the integration of different parallelism dimensions in Hy-Fi.

In spatial parallelism, the convolution layer is replicated across multiple GPUs, and image parts are partitioned across replicas. Specifically, the level of granularity in layer parallelism is a layer, while in spatial parallelism it is neurons. Convolution and Pooling layers can be distributed across multiple GPUs to work on the different regions of the image. Hence, unlike layer parallelism, this approach enables simultaneous computation on multiple GPUs while facilitating the training of the out-of-core convolution layer. There are two significant issues with the spatial parallelism approach; 1) Extra Communication is necessary and 2) Complex implementation. Spatial parallelism requires a halo exchange (shown in Fig. 3) at every convolution and pooling layer to compute the result for the pixels present on the boundary of image parts [6]. Parameters like stride, filter size, and padding affect the size of the halo exchange, which increases any spatial parallelism implementation's complexity compared to layer parallelism. In the backward pass, `allreduce` is required to synchronize the weights of the convolution layer for every process performing spatial parallelism (Fig. 2).

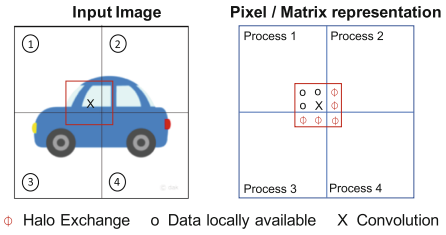
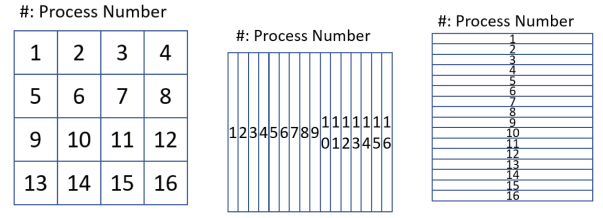


Fig. 3. Halo exchange in spatial parallelism. The input image is partitioned into four regions, and each region is given to the different processes. To calculate the convolution operation at X location, the value of nearby pixels is required.

To tackle communication overhead in spatial parallelism, we propose two optimization strategies.

Layout Optimization. Distribution layout plays an important role in the number of send/recv operations in a halo exchange. There are many ways to partition an image among processes. A common approach is to partition the image into square patches as shown in Fig. 4(a). This approach is known as a square layout. We investigate vertical and horizontal layouts for spatial parallelism. In a vertical layout, the image is partitioned along the width dimension. Similarly, in a horizontal layout, the image is partitioned along the height dimension. In a



(a) Square Layout (b) Vertical Layout (c) Horizontal Layout

Fig. 4. Image distribution strategies

square layout, the maximum number of send/recv operations is 8. For example, process P6 will send/recv data from P1, P2, P3, P5, P7, P9, P10, and P11. However in horizontal and vertical layouts, the maximum number of send/recv operations is limited to 2 (can be inferred from Fig. 4(b) and 4(c)). Peculiar process placement in vertical and horizontal layout helps in reducing the inter-node communication by placing adjacent processes on the same node, which is not possible in a square layout. These factors help in reducing the communication.

Halo-D2: Reduced Communication Operations. A halo exchange is required at each layer in spatial parallelism in order to apply convolution/pooling operation in basic spatial parallelism. The main objective of the convolution operation is to produce an output of the same width and height. Normally in CNNs, several convolution operations of kernel size 3 are stacked together to efficiently implement a large kernel size [25]. This approach leads to several halo exchanges since it's required at every layer. We reduced the number of blocking communication operations by exchanging more pixels around the border. Spatial parallelism avoids the repeated computation on the border by exchanging data at every layer. However, in our evaluation, we found that the convolution operation takes the same time for images with a few more pixels due to the massively parallel computation provided by GPUs. For example, the computation time for a 256×256 image was the same as a 260×260 image. Therefore, by exchanging more data at one layer, we can avoid more halo exchanges in subsequent layers. Figure 5 shows an example of spatial parallelism with Halo-D2.

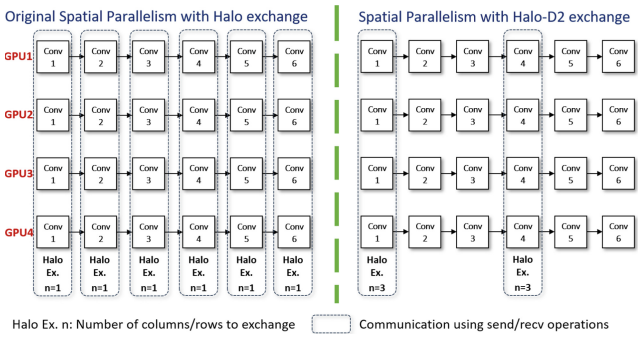


Fig. 5. Motivation for Halo-D2. Instead of exchanging only required data at every layer, additional data is exchanged to eliminate the need of exchanging data for subsequent layers.

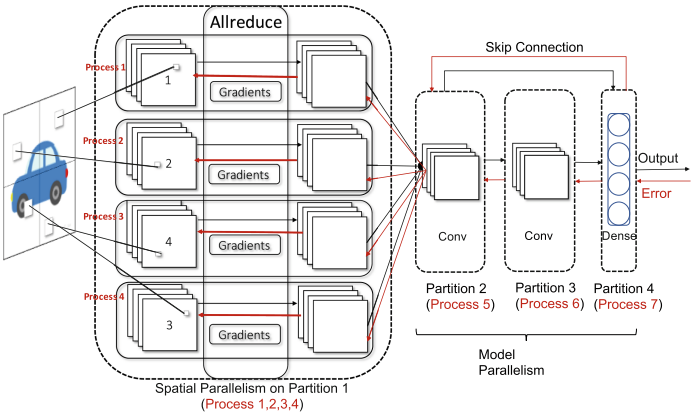


Fig. 6. Proposed spatial parallelism + layer parallelism design. CNN is sliced into four partitions. Spatial parallelism is applied to the 1st partition, and layer parallelism is used for the rest of the partitions.

4.2 Spatial Parallelism + Layer Parallelism

Due to the increased communication overhead, spatial parallelism is more suitable for large images, which makes this approach inappropriate for the latter half of CNNs where the image input size is usually few pixels. Layer parallelism can be used to compute this latter half. Figure 6 shows a combination of spatial parallelism and layer parallelism for a CNN partitioned into four partitions on layer granularity. Spatial parallelism is applied to the first model partition, and layer parallelism is applied to the other three model partitions.

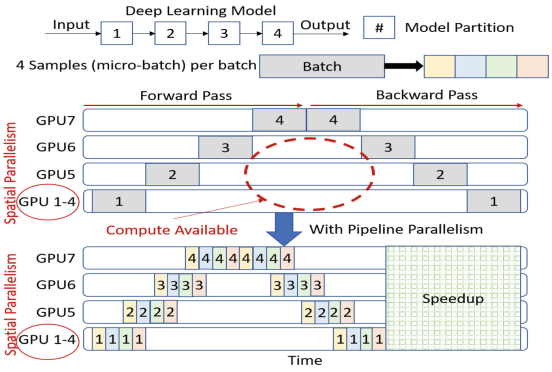


Fig. 7. Spatial and layer parallelism combined with pipeline parallelism. The combination of spatial and layer parallelism fail to exploit the parallelism within batches that can be used by pipeline parallelism to utilize more than one GPU at any given time.

4.3 Pipeline Parallelism

Spatial and Layer parallelism exploits parallelism within a layer and model. However, they fail to exploit parallelism within batches. Figure 7 shows the computational view of spatial and layer parallelism for the model shown in Fig. 6. As shown in the figure, compute is available between forward and backward pass. However, previous strategies fail to exploit this compute when the batch size is greater than 1. We use pipeline parallelism to exploit a third dimension of parallelism using micro-batches. Figure 7 shows the integration of pipeline parallelism with Spatial and Layer parallelism to exploit parallelism within batches, which improves the overall performance.

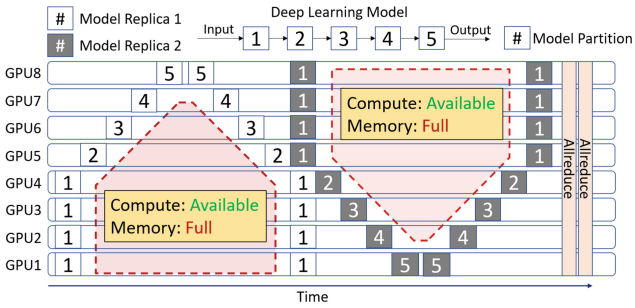


Fig. 8. Bi-directional with spatial and layer parallelism. A naive integration limits the performance because of blocking allreduce operations at the end. The available compute can be used to eliminate allreduce operation.

4.4 Spatial + Bidirectional Parallelism

To further improve the performance of spatial, layer, and pipeline parallelism, we explore a fourth dimension of parallelism i.e. the direction of forward and backward pass. By using bi-directional parallelism, we are able to overlap computation with different batches and therefore improve performance. This dimension is suitable when a DL researcher wants to train their model with larger batch size than the maximum feasible batch size (the maximum batch size is limited by GPU memory). Bi-directional parallelism increases the performance when the batch size is not possible under traditional parallelization strategies. In this section, we integrate first three dimension of parallelism with a fourth dimension (Bidirectional parallelism). Figure 8 shows the need for communication optimizations in the integration of spatial, layer, and pipeline parallelism with bi-directional parallelism.

Communication Optimization for Integration with Spatial Parallelism. To remove the necessary allreduce operation at the end, we use send/recv operations to communicate parameters and gradients of replica1 during the dotted

bubble in Fig. 8. We divide our design into two steps: 1) Parameters exchange and 2) Gradient exchange.

Parameters Exchange: In this step, we assume that the first model replica has the latest DL model parameters and the second model replica does not have the latest parameters since we are not using an allreduce operation at the end to synchronize the training. Therefore, we will send the latest parameters from model replica 1 to model replica 2 during the first bubble.

Gradients Exchange: In an allreduce operation, gradients are averaged across all model replicas to synchronize the training and make sure parameters remain the same for all replicas. Since we are updating only the last replica, we need the gradients of the previous replica in order to synchronize the training and update replica 2 to the latest parameters. The second bubble in Fig. 8 can be used to exchange these gradients from replica 1 to replica 2.

After the first iteration of a parameter and gradient exchange, model replica 2 becomes the replica with the latest parameters. Therefore, the next forward and backward iteration will complete on model replica 2 first. Figure 9 shows two iterations of our proposed communication optimization.

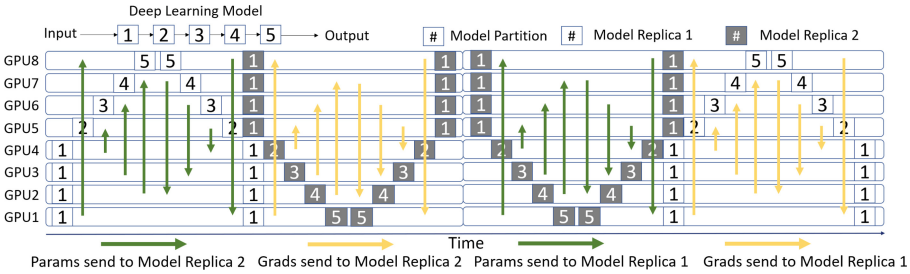


Fig. 9. Two iterations of communication optimized Hy-Fi master (spatial, layer, pipeline, and bi-directional)

4.5 Hybrid Data Parallelism

To scale proposed designs to a large number of GPUs, we exploit a fifth dimension of parallelism: data parallelism. We create clusters of GPUs, where each cluster implements the first four dimensions of parallelism in Hy-Fi and synchronizes each replica's parameters via allreduce operations. The integration with data parallelism allows our proposed design to scale to a large number of GPUs and provides better scaling efficiency.

Implementation Details: Our implementation of Hy-Fi is inspired by the pipeline parallelism fundamentals and APIs presented in the HyParFlow system [2]. For communication, we have used PyTorch's distributed module and created a wrapper communication class to create required communicators in proposed hybrid data parallelism (Sect. 4.5) and spatial parallelism (Sect. 4.1).

A model generator class is created to divide the model into partitions. *Trainer* class is created for every parallelism dimension to implement distributed forward and backward pass. For spatial parallelism, a wrapper class around *Conv2D* class is implemented to realize proposed designs for Halo communication.

5 Performance Evaluation

5.1 Evaluation Platform

All the experiments were conducted on LLNL/Lassen, which is an OpenPOWER system equipped with POWER9 processors and 4 NVIDIA Volta V100 GPUs. Each node of the cluster is a dual-socket machine, and each socket is equipped with 22-core IBM POWER9 processors and 2 NVIDIA Volta V100 GPUs with 16 GB HBM2. NVLink is used to connect GPU-GPU and GPU-Processor.

Softwares: Pytorch v1.7 and MVAPICH2-GDR 2.3.5.

Deep Neural Networks: We defined ResNet variants from Keras examples/applications in PyTorch and used the AmoebaNet model from TorchGpipe [16].

5.2 Evaluation Setup and Performance Metrics

We use images per sec as the main performance metric in this paper. Other terms and legends used in this performance evaluation are explained below.

- **Images per sec:** Number of images processed in training per sec.
- **BS:** Batch Size
- **LP:** Layer Parallelism (or Model-Parallelism Basic)
- **Pipeline:** Pipeline Parallelism.
- **SP and SP-Opt:** Hy-Fi's Spatial Parallelism and its optimized version (Layout Optimization and Halo-D2).
- **SP-#:** Hy-Fi's Spatial Parallelism with # Layout (Sq: Square, Hor: Horizontal, and Ver: Vertical)
- **SP-#-D2:** Hy-Fi's Spatial Parallelism with # Layout and Halo-D2 optimization.
- **Master-#:** Hy-Fi with four parallelism dimensions (Spatial, Layer, Pipeline, and Bi-Directional). # is the number of replications in Bi-Directional's Master design.
- **Master-#-Opt:** Master-# with communication optimization.

5.3 Evaluation Methodology

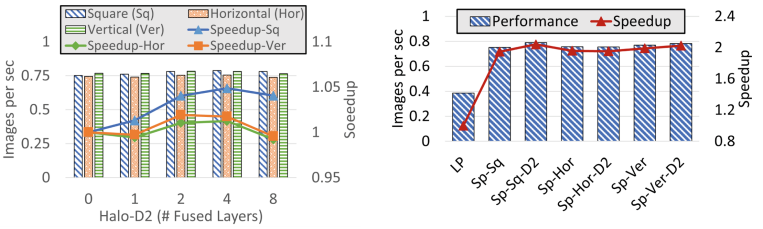
In this section, we describe the evaluation methodology used to conduct experiments and show Hy-Fi's benefits. Broadly, our experiments can be divided into four categories; 1) Performance analysis of different dimensions of parallelism in

Hy-Fi and their optimizations (Sect. 5.4 and 5.5) 2) Scaling Hy-Fi on a large number of GPUs (Sect. 5.6), 3) Comparison against existing frameworks (Sect. 5.7), and 4) Enabling training of very high-resolution images and speedup using Hy-Fi (Sect. 5.8). We use two variants of AmoebaNet and ResNet-218 v2 model. AmoebaNet-f214 and AmoebaNet-f416 have 18 cells and the number of initial filters is 214 and 416, respectively. AmoebaNet model variants are evaluated on $2{,}048 \times 2{,}048$ images. AmoebaNet-f214 is used since it can be trained on 8 GPUs with BS 2, making pipeline parallelism possible. AmoebatNet-f416 on $2{,}048 \times 2{,}048$ and ResNet-218-v2 on $1{,}024 \times 1{,}024$ can only be trained with BS $= 1$ on 8 GPUs, which makes pipeline parallelism impossible.

5.4 Performance Benefits of Spatial Parallelism

We start by demonstrating the benefits of Layout and Halo-D2 optimizations for Hy-Fi spatial parallelism and compare Hy-Fi's spatial parallelism with layer parallelism and pipeline-parallelism in the literature. Figure 10(a) shows the effect of number of fused layers in Halo-D2 (Sect. 4.1). The number of fused layers determines both the size of a halo exchange and how many layers can be skipped for a halo exchange. For the ResNet-218v2 model, we found that Halo-D2 gives the best performance for 4 fused layers. Proposed Halo-D2 optimization increases the training performance by up to 4.8%. Figure 10(b) shows the performance comparison of different proposed optimizations on spatial parallelism and compares them to LP. Hy-Fi's spatial parallelism is 1.94× faster than LP without optimizations for spatial parallelism.

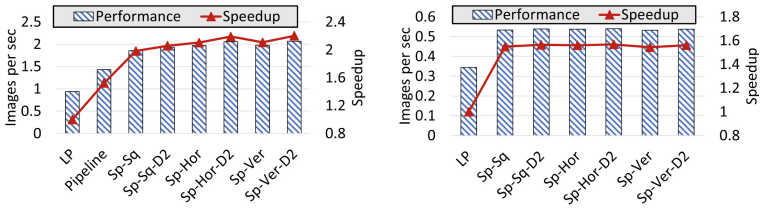
By combining layout and Halo-D2 optimizations, we are able to improve the performance to 2.04×. Figure 11(a) and Fig. 11(b) show the performance comparison of spatial parallelism optimizations, LP, and pipeline parallelism (when possible). For AmoebaNet-f214, we use the first three dimensions of parallelism in Hy-Fi (spatial, model, and pipeline) when a batch size greater than 1 is possible. Hy-Fi is 2.2× faster than LP and 1.44× faster than existing pipeline parallelism. The proposed optimizations to spatial parallelism increases the performance improvement from 1.98× to 2.2×.



(a) Effect of number of fused layers in Halo-D2 for different layouts

(b) Performance comparison of LP and spatial parallelism optimizations

Fig. 10. ResNet-218v2 on 8 GPUs using $1{,}024 \times 1{,}024$ images

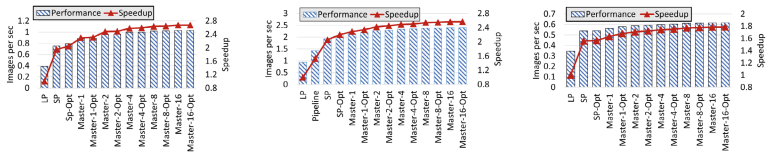


(a) AmoebaNet-f214 (b) AmoebaNet-f416

Fig. 11. Performance comparison of LP and different spatial parallelism optimizations for AmoebaNet on 8 GPUs using 2,048 × 2,048 images

5.5 Improving Performance Using Bi-directional Parallelism

The first three dimensions of parallelism do not exploit the free memory and compute resources available between training steps. Therefore we integrate bi-directional parallelism and increases its performance by removing the blocking allreduce operation at the end (Sect. 4.4). This enables the training of larger batch sizes on the same number of resources and improves the throughput, which was impossible earlier because of memory requirements. Figure 12 demonstrates the benefits of a fourth dimension of parallelism (GEMS-MASTER) in Hy-Fi. We compare our designs against existing layer and pipleine parallelism. Bi-directional parallelism uses a number of replications to stack more batches before the weight update. Therefore, we show performance improvements for up to 16 replications. The improvement in performance was negligible after 16 replications. In Fig. 12(a), we improve the performance from 2.04× to 2.67× using bi-directional parallelism. For AmoebaNet-f214 (Fig. 12(b)) and AmoebaNet-f416 (Fig. 12(c)), we show speedup improvement from 2.05× to 2.56× and from 1.56× to 1.78×. Using our proposed communication optimization (Sect. 4.4), we are able to improve speedup for replications = 1 from 2.28× to 2.34× and from 1.63× to 1.68× for AmoebaNet-f214 and AmobaNet-f416. For the ResNet-218v2 model, we observed that the improvement in speedup is 1.01× because of a small number of parameters compared to the AmoebaNet model, which translated into negligible allreduce time. As we tack more and more compute in MASTER by increasing the number of replications and batch size, the percentage of allreduce time decreases. Therefore, we see smaller and smaller speedup improvement for the communication optimization approach. However, we found that the communication optimized design always gave better performance than basic integration and proposed communication optimization improves the overall training performance by up to 7%. Therefore, Hy-Fi improves the performance for smaller batch sizes and enables researchers to use it without compromising on throughput.

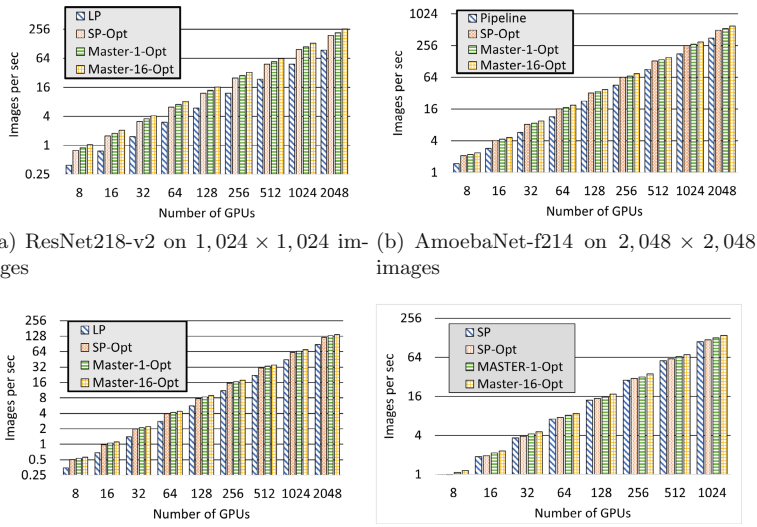


(a) ResNet218-v2 on $1,024 \times 1,024$ images (b) AmoebaNet-f214 on $2,048 \times 2,048$ images (c) AmoebaNet-f416 on $2,048 \times 2,048$ images

Fig. 12. Performance comparison of Hy-Fi's 4th dimension of parallelism (bi-directional) with and without communication optimization

5.6 Hybrid Parallelism

We demonstrate the proposed Hy-Fi system's scalability (Fig. 17) by scaling four CNNs. This experiment uses all five dimensions of parallelism and respective optimizations to scale Hy-Fi to 2,048 GPUs. All four evaluated Hy-Fi designs (Sp-Opt, Master-1-Opt, and Master-16-Opt) achieve near-linear speedup. For Hy-Fi's Master-1-Opt, we achieve 246× speedup for ResNet, 244× speedup for AmoebaNet-f214, and 242× speedup for AmoebaNet-f416 on 2,048 GPUs. The ideal speedup is 128× for 1,024 GPUs and 256× for 2,048 GPUs since models are partitioned across 8 GPUs. VGG16 achieves 199× speedup on 1,024 GPUs. The near-linear scaling of proposed designs can be attributed to the proposed communication optimization in Hy-Fi and its efficient implementation. Instead of doing allreduce operation twice in bi-directional parallelism, we do allreduce once in our proposed communication optimization (Fig. 13).



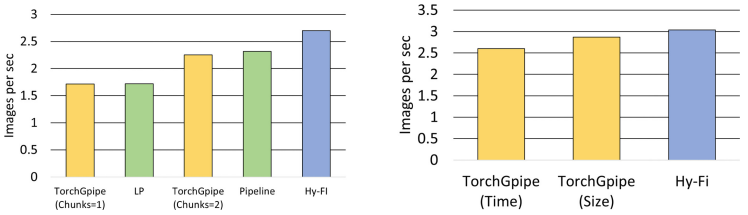
(a) ResNet218-v2 on $1,024 \times 1,024$ images (b) AmoebaNet-f214 on $2,048 \times 2,048$ images

(c) AmoebaNet-f416 on $2,048 \times 2,048$ images (d) VGG16 on $4,096 \times 4,096$ images

Fig. 13. Scaling Hy-Fi's optimized designs with all 5 parallelism dimensions on 2,048 GPUs

5.7 Hy-Fi vs Existing Frameworks

Comparison with TorchGPipe. We compare Hy-Fi against TorchGPipe for two primary reasons; 1) TorchGPipe has an efficient implementation of pipeline parallelism in PyTorch and 2) TorchGpipe has memory-level optimizations to enable the training of out-of-core batch sizes. Since TorchGpipe does not have multi-node support, we reduced the number of cells and initial filters in the AmoebaNet-f214 model to enable training on 4 GPUs. Figure 14(a) compares TorchGpipe's layer and pipeline parallelism implementations with ours and shows the benefits of Hy-Fi (1.2×) on the same batch size. It further validates the efficiency of our baseline implementation for layer and pipeline parallelism. Figure 14(b) compares the maximum performance attainable by both frameworks for any batch size and shows up to 1.06× speedup for Hy-Fi.



(a) Performance for Batch Size = 2 (b) Maximum Performance with any Batch Size

Fig. 14. Performance comparison of Hy-Fi and TorchGpipe for AmoebaNet on 4 GPUs using 2,048 × 2,048 images

Comparison with Mesh-TensorFlow and GEMS. To the best of our knowledge, there is no distributed training framework in PyTorch that implements spatial parallelism. Therefore, we use Mesh-TensorFlow since it is implemented in TensorFlow (TensorFlow and PyTorch are the two most popular DL frameworks). Since GEMS [14] conducted experiments on the same system, we use their Mesh-TensorFlow and GEMS numbers to compare our proposed designs.

Figure 15 compares Hy-Fi against state-of-the-art Mesh-TensorFlow and GEMS frameworks. We show 1.13× and 1.4× speedup for Hy-Fi over GEMS and Mesh-TensorFlow, respectively. We attempted to compare results with the FlexFlow framework, but encountered a number of issues with their PyTorch plugin. First, at the time of writing, many of the advanced operators/-modules in the Amoebanet PyTorch model are not interpretable by the base FlexFlow model transformation function. Further, we

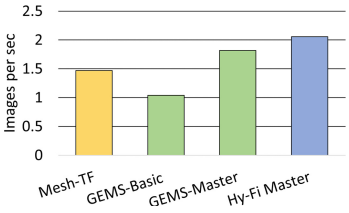
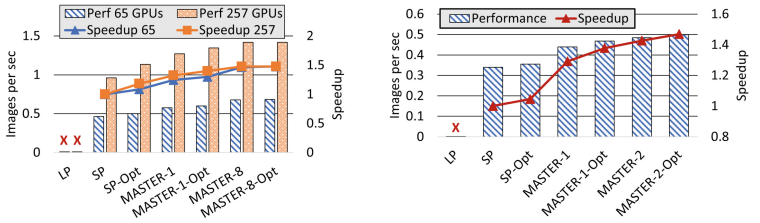


Fig. 15. Comparison with Mesh-TensorFlow and GEMS for ResNet-110 on 4 GPUs using 1,024 × 1,024 images

were unable to train on out-of-core batch sizes due to a conflict with the memory managers of FlexFlow and Legion [4] (which FlexFlow uses for intra-node communication).

5.8 Next-Generation DNN Designs on Very High-Resolution Images Using Hy-Fi

Today, Deep learning researchers develop models restricted by the number of layers for high-resolution images such as $8{,}192 \times 8{,}192$ and $16{,}384 \times 16{,}384$. Layer parallelism can be used to train out-of-core models, yet requires a single layer to fit inside a GPU's memory, which is a limitation for very high-resolution images. For example, a single channel $16{,}384 \times 16{,}384$ image consumes around 1GB of memory with FP32 representation. This makes the training impossible for CNNs using very high-resolution images. To illustrate the possibility of training models on very high-resolution images, we stress-test the proposed Hy-Fi system by training the AmoebaNet-f214 model on $8{,}192 \times 8{,}192$ and $16{,}384 \times 16{,}384$ very high-resolution images. Figure 16(a) and Fig. 16(b) demonstrate the benefits of Hy-Fi for both enabling the training entirely, and further accelerating it.



(a) AmoebaNet-f214 on $8,192 \times 8,192$ images

(b) AmoebaNet-f214 on $16,384 \times 16,384$ images

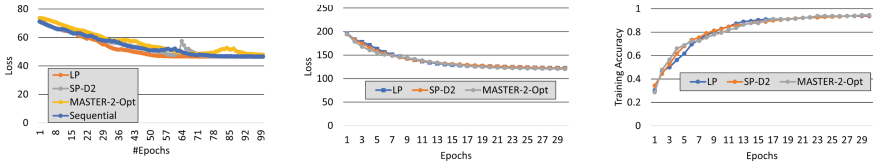
Fig. 16. Enabling and accelerating training on very high-resolution images

At least 16 GPUs are needed to train the AmoebaNet-f214 model on $8{,}192 \times 8{,}192$ images (Fig. 16(a)); therefore, we use spatial parallelism on 16 GPUs for convolution and pooling layers and layer parallelism on 1 GPU for the classification module in the AmoebaNet model. By using the optimizations in Hy-Fi's spatial parallelism (Sect. 4.1) and bi-directional parallelism (Sect. 4.4), we are able to further accelerate the training and achieve up to 1.476× speedup compared to the basic spatial parallelism approach. Further, we accelerate the training using strong scaling by increasing the number of GPUs to train the model with the same batch size. We are able to achieve a 2.26× speedup using strong scaling. In Fig. 16(b), we enable and accelerate the training for $16{,}384 \times 16{,}384$ images and achieve up to 1.47× speedup compared to basic spatial parallelism.

5.9 Verifying the Correctness of Hy-Fi

We have extended the PyTorch and implemented distributed training from scratch to support proposed designs. Therefore, it is important to show that Hy-Fi trains the model in the same number of epochs using proposed designs to give confidence to DL researchers. We trained ResNet-218 v2 CNN for a subset of Cifar-10 and Places-365 datasets. First, we provide results for the Cifar-10 dataset as it can be trained on a single GPU without distributed DNN training. Figure 17(a) shows trend of loss function for 100 epochs The objective of this experiment is to showcase the correctness of Hy-Fi's proposed designs with respect to sequential out-of-the-box training provided by PyTorch.

Figure 17(b) and Fig. 17(c) show trend of loss and accuracy functions for 30 epochs when training ResNet-218 v2 model on a dataset with larger image sizes (512×512). It cannot be trained on a single GPU as the model becomes out-of-core for 512×512 image size. We note that DNN training is a stochastic process; therefore, there can be variations in few epochs whether we use sequential training or distributed DNN training. However, the overall trend should remain the same. We ran these experiments multiple times to ensure that the loss function trend presented here is reproducible.



(a) On a subset of Cifar-10 dataset (b) On a subset of Places-365 dataset (Loss) (c) On a subset of Places-365 dataset (Acc)

Fig. 17. Verifying the correctness of proposed designs in Hy-Fi by training ResNet-218 v2 on multiple datasets

6 Related Work

The growth of scientific and medical applications requiring massive data sample sizes [7] has led deep learning researchers to explore new parallelism techniques that train on such images without high accuracy and efficiency. Krizhevsky's work pioneered basic model parallelism techniques in [18]. GPipe [12] employs pipeline parallelism to enable the training of extremely large models like AmoebaNet [22]. Further, PipeDream [10] expands upon GPipe's pipelining idea by introducing pipeline parallelism, which combines inter-batch and intra-batch parallelism to increase overlap among GPUs. Torchgpipe [16] combined the overall design of GPipe (pipeline parallelism) with some of the eager execution and memory-aware enhancements of HyPar-Flow into a distributed PyTorch DL

framework. GEMS [14] introduced memory-aware partition overlap for out-of-core models on GPUs, but does not support spatial parallelism. Spatial parallelism, however, is a more recent addition to model parallel techniques [9]. LBANN introduced spatial convolutions split across nodes in [6]. However, spatial parallelism support in LBANN doesn't include pipelining nor bidirectional training as in the GEMS design. FlexFlow [15] searches through all parallelization strategies with simulation algorithms and highlights different DNN parallelism dimensions. We attempted to compare our work with FlexFlow but ran into issues with the framework when handling large images on our system. Mesh-TensorFlow (MTF) [23] is a framework for distributed DNN training which partitions tensors across a processor mesh. We summarize these related studies and their features in Table 1.

7 Conclusion

Convolutional Neural Networks (CNNs) are making breakthroughs in the computer vision area, but are hard to train on very high-resolution images due to memory and compute constraints. In this paper, we present Hy-Fi - an integrated hybrid five-dimensional distributed DNN training system that uses different parallelism dimensions in tandem and accelerates training for very high-resolution images. Hy-Fi uses novel communication and compute optimizations for different parallelism dimensions and efficiently integrates these dimensions to speed up training. The proposed design is evaluated with state-of-the-art deep learning models like AmoebaNet and ResNet. We report up to 2.02× speedup over layer parallelism and 1.44× speedup over pipeline parallelism using our optimized spatial, layer, and pipeline parallelism. Further, we improve speedup using optimized memory-aware designs to 2.67× over layer parallelism and 1.68× over pipeline parallelism. We scale our designs to 2,048 GPUs and show up to 94.5% scaling efficiency. In the end, we demonstrate training on very high-resolution images and report up to 1.47× speedup over basic spatial parallelism. We believe that Hy-Fi will pave a way forward for solving complex and compute-intensive problems in scientific, digital pathology, and artificial intelligence areas.

Acknowledgement. This research is supported in part by NSF grants 1818253, 1854828, 1931537, 2007991, 2018627, 2112606, and XRAC grant NCR-130002.

References

1. Awan, A.A., Hamidouche, K., Hashmi, J.M., Panda, D.K.: S-Caffe: co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 193–205. ACM, New York (2017)
2. Awan, A.A., Jain, A., Anthony, Q., Subramoni, H., Panda, D.K.: HyPar-Flow: exploiting MPI and Keras for scalable hybrid-parallel DNN training using TensorFlow (2019)

3. Awan, A.A., Subramoni, H., Panda, D.K.: An in-depth performance characterization of CPU- and GPU-based DNN training on modern architectures. In: Proceedings of the Machine Learning on HPC Environments, MLHPC 2017, pp. 8:1–8:8. ACM, New York (2017)
4. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012. IEEE Computer Society Press (2012)
5. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. CoRR abs/1802.09941 (2018)
6. Dryden, N., Maruyama, N., Benson, T., Moon, T., Snir, M., Essen, B.V.: Improving strong-scaling of CNN training by exploiting finer-grained parallelism. CoRR abs/1903.06681 (2019). http://arxiv.org/abs/1903.06681
7. Farrell, S., et al.: Novel deep learning methods for track reconstruction (2018)
8. Gholami, A., Azad, A., Jin, P., Keutzer, K., Buluc, A.: Integrated model, batch, and domain parallelism in training neural networks. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, pp. 77–86. ACM, New York (2018). https://doi.org/10.1145/3210377.3210394
9. Gholami, A., Azad, A., Jin, P., Keutzer, K., Buluc, A.: Integrated model, batch, and domain parallelism in training neural networks. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, pp. 77–86 (2018)
10. Harlap, A., et al.: PipeDream: fast and efficient pipeline parallel DNN training. CoRR abs/1806.03377 (2018). http://arxiv.org/abs/1806.03377
11. Huang, Y., et al.: GPipe: efficient training of giant neural networks using pipeline parallelism. CoRR abs/1811.06965 (2018). http://arxiv.org/abs/1811.06965
12. Huang, Y., et al.: GPipe: efficient training of giant neural networks using pipeline parallelism. In: NeurIPS (2019)
13. Jain, A., et al.: SUPER: SUb-graph parallelism for transformers. In: 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2021
14. Jain, A., et al.: GEMS: GPU-enabled memory-aware model-parallelism system for distributed DNN training. In: 2020 SC 2020: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 621–635. IEEE Computer Society (2020)
15. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. CoRR abs/1807.05358 (2018). http://arxiv.org/abs/1807.05358
16. Kim, C., et al.: torchgpipe: on-the-fly pipeline parallelism for training giant models (2020)
17. Kousha, P., et al.: Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance GPU clusters. In: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), pp. 93–102 (2019). https://doi.org/10.1109/HiPC.2019.00022
18. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. CoRR abs/1404.5997 (2014). http://arxiv.org/abs/1404.5997
19. Lee, S., et al.: Interactive classification of whole-slide imaging data for cancer researchers. Cancer Res. **81**(4), 1171–1177 (2021). https://doi.org/10.1158/0008-5472.CAN-20-0668. https://cancerres.aacrjournals.org/content/81/4/1171
20. Paszke, A., et al.: Automatic differentiation in PyTorch (2017)
21. Petrowski, A., Dreyfus, G., Girault, C.: Performance analysis of a pipelined backpropagation parallel algorithm. IEEE Trans. Neural Netw. **4**(6), 970–981 (1993). https://doi.org/10.1109/72.286892

22. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. CoRR abs/1802.01548 (2018)
23. Shazeer, N., et al.: Mesh-TensorFlow: deep learning for supercomputers. In: Advances in Neural Information Processing Systems, vol. 31. Curran Associates, Inc. (2018)
24. Shoeybi, M., Patwary, M.A., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: training multi-billion parameter language models using model parallelism. ArXiv abs/1909.08053 (2019)
25. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)