

Efficient Data-parallel Distributed DNN Training for Big Dataset under Heterogeneous GPU Cluster

Shinyoung Ahn, Sookwang Lee*
Supercomputing System Research Section
ETRI

Daejeon, Republic of Korea
{syahn, 2sookwang}@etri.re.kr

Hyeonseong Choi
Software Team
MangoBoost

Seoul, Republic of Korea
chyon794@gmail.com

Jaehyun Lee
Research Institute
Puzzle AI Inc.

Daejeon, Republic of Korea
jay@puzzle-ai.com

Abstract—Training large-scale deep neural networks (DNNs) using a large number of parameters requires significant computational resources. Despite the rapid advancements in GPU technology, limited budgets have forced many institutions to gradually build GPU servers, leading to growing challenges with resource heterogeneity. However, most open-source distributed deep-learning libraries use synchronous training algorithms that perform better on homogeneous GPUs than on heterogeneous GPUs. Therefore, many researchers have struggled to efficiently conduct distributed training on heterogeneous GPU clusters owing to the straggler problem. In this study, we introduce Efficient Distributed Deep learning library based on SoftMemoryBox(EDDIS), a novel data-parallel distributed deep learning library. EDDIS overcomes the scalability limitations caused by heterogeneity, enabling the efficient utilization of heterogeneous GPU resources. EDDIS trains DNNs synchronously, asynchronously, and in a hybrid manner and supports TensorFlow and PyTorch. In a heterogeneous GPU environment, EDDIS's three training modes—synchronous, asynchronous, and hybrid synchronous—accelerate distributed DNN training by approximately 8.2x, 19x, and 18.7x on 16 nodes, respectively, compared to a single node. In particular, the EDDIS hybrid synchronous training mode achieves training speeds that are 2.8 times faster than PyTorch DDP and 2.3 times faster than Horovod when training the Yolov5m model.

Index Terms—Data-parallel Training, Deep Neural Network, Distributed Training, DNN, EDDIS, Heterogeneous GPU, Hybrid-EASGD, SoftMemoryBox, SSGD,

I. INTRODUCTION

As deep-learning continues to advance, there is an increasing demand for models with more capabilities (multimodal) and higher performance (accuracy). Deep neural network models that generally offer a higher performance and more functionalities have more parameters [1]. Larger models must be trained with more data to achieve better performance. Consequently, training DNN models with superior capabilities and accuracy requires an exponential increase in computational power [2]–[4]. To address the rapidly increasing computational requirements, high-performance computing (HPC) systems must be used to train DNNs in a distributed manner. The distributed training of DNN models using HPC systems requires a very high communication bandwidth among

the distributed processes. Insufficient bandwidth can lead to communication bottlenecks and prevent efficient distributed DNN training. Furthermore, as the number of distributed training processes (each process typically occupies one GPU) increases, the communication bottlenecks become more pronounced. Thus, the overall training time does not decrease linearly, and as the scale of distributed training exceeds a certain level, the training time increases. The rapid advancement and high cost of GPUs have led institutions to build fragmented, small-scale, heterogeneous GPU servers incrementally. Except for TensorFlow, the majority of open-source deep-learning libraries, such as Horovod and PyTorch DDP, support only synchronous training methods that perform better in homogeneous GPU environments. Therefore, they cannot effectively perform distributed training in environments that utilize heterogeneous GPUs owing to straggler problems.

In this study, we propose EDDIS, which reduces the scalability limitations and inefficiencies in distributed DNN training on heterogeneous GPU clusters, enabling faster and more cost-effective deep-learning model development. EDDIS is the successor version of ShmCaffe [5], which is a distributed deep learning technique that extends Caffe using SoftMemoryBox [6], [7] and TensorFlow based on SoftMemoryBox(TFSM) [8], a distributed deep-learning library based on TensorFlow v1.x. EDDIS supports data-parallel distributed training of DNN models written in TensorFlow v2.x. and PyTorch v1.x. It can significantly reduce the training time of computationally intensive DNNs with large amounts of training data by efficiently performing distributed training on heterogeneous GPU servers.

EDDIS is also more convenient than the conventional parameter server method for developing asynchronously distributed training codes. In general, asynchronously distributed training codes using the parameter server approach require separate implementation of code logic for the parameter servers and training workers with manual connection configurations between the parameter servers and workers in the source code, and they need to be executed separately individually. However, EDDIS enables all training workers to run simultaneously via the Message Passing Interface (MPI) library, whereas SoftMemoryBox (SMB) servers run as background daemons, allowing for easy initiation and termination

This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government[24ZS1100, Research on High-Performance Computing to overcome Limitations of AI]

* Corresponding author: 2sookwang@etri.re.kr

of the training processes. Additionally, during distributed training, stopping a single mpirun process terminates all the training worker processes in the batch. Unlike the conventional parameter server model, the SMB servers in EDDIS operate as daemons and support distributed training for other DNNs.

The contributions of this study are as follows:

- 1) EDDIS can support data-parallel distributed training in various hardware environments, including homogeneous GPU and heterogeneous GPU environments, by providing three training modes that enable customized utilization of computing resources.
- 2) EDDIS offers high-speed asynchronous parameter sharing through global parameter sharding across multiple SoftMemoryBox servers and particularly achieves efficient GPU utilization capabilities by controlling the number of trained mini-batches per GPU differently based on the computing capacity.
- 3) EDDIS provides a fully parallelized technique between the threads responsible for performing training and updating global weights, which helps alleviate communication bottlenecks when training in a heterogeneous GPU environment using asynchronous/hybrid approaches. This technique allows the communication time to be masked as much as possible by computational time.
- 4) We propose Hybrid-EASGD (HEASGD), a new distributed training algorithm that improves asynchronous training performance over the Shared Memory-based EASGD (SEASGD) algorithm [5].

The remainder of this paper is organized as follows. Section II describes the background of the key concepts in this study. Section III introduces EDDIS. Section IV presents a comprehensive guide for training DNNs using EDDIS in a distributed manner. Section V presents the performance evaluation results of EDDIS. Finally, Section VI summarizes our study and discusses future research.

II. BACKGROUND

Distributed deep learning is performed using data parallel, model parallel, or both. Data parallelism is an approach in which multiple workers train replicas of DNNs using a subset of the training dataset. Model parallelism is an approach in which multiple workers train a portion of the DNNs with the entire dataset [2], [9]–[12]. This paper discusses distributed deep learning with a focus on data parallelism, which is one of the two main methods.

In data-parallel training, distributed workers must periodically synchronize newly explored parameters with other workers, typically after each mini-batch training (i.e., step in PyTorch). Global parameter synchronization algorithms can be broadly classified into two categories: Synchronous Stochastic Gradient Descent (SSGD) and Asynchronous Stochastic Gradient Descent (ASGD). In the SSGD method, gradients are exchanged using all-reduce, and local weights are updated, which can lead to synchronization overhead [13], [14]. By contrast, the ASGD method uses a parameter server to receive gradients from workers, update global weights, and send the

updated weights back to the workers. Although ASGD allows faster training with less synchronization overhead, it may take longer to achieve effective learning owing to gradient staleness [15]. EASGD is a model-averaging method in which each deep-learning worker trains a copy of the DNN model and exchanges weights with a parameter server to update the local weights [16], [17]. Stale Synchronous Parallelism (SSP) blends SSGD and ASGD by synchronizing gradients within a set delay, reducing the impact of outdated information [18], [19]. Chen’s research showed that ASGD converges more slowly and yields lower accuracy than SSGD, making SSGD the preferred method for large-scale distributed deep learning [14]. Recent SSGD studies have improved accuracy and reduced communication overhead [20]. SSGD is most effective in homogeneous GPU clusters, particularly with Tesla GPUs that offer superior communication capabilities such as GPUDirect RDMA (Remote Direct Memory Access). However, in heterogeneous GPU clusters, the straggler problem can lower GPU utilization. HetSeq [21] approached this problem by using different batch sizes for heterogeneous GPUs and averaging losses based on the batch size to calculate gradients. However, HetSeq does not fully solve the straggler problem, and its need for tight integration with the model code significantly reduces its usability. Therefore, the training modes should be dynamically adjusted based on GPU resources: SSGD for homogeneous clusters, ASGD for heterogeneous clusters, and a combination of both, when different GPU types are used at the node level. ShmCaffe [5] combines the parameter server approach of DistBelief [10] with the shared memory access method of Hogwild [22]. SoftMemoryBox [6], [7] provides a remote shared memory buffer to store global weights among distributed workers, thereby creating the illusion of lock-free data access. However, unlike Hogwild, because the actual data reside in remote memory, updating weights without locks can lead to data races, potentially causing a loss of learned data. To address this issue, SoftMemoryBox ensures data consistency on the server side. Unlike traditional parameter servers, SoftMemoryBox does not depend on specific model network information and operates as a daemon that supports various models. Therefore, distributed workers must request updates directly from the global weights in SoftMemoryBox at a low level (e.g., tensor). EDDIS was developed based on this need and built upon SoftMemoryBox. The zero-redundancy optimizer (ZeRO) [23] is a technique that overcomes GPU memory limitations, requiring sufficient communication bandwidth between the GPU and the host memory or SSD. ZeRO is only effective for Tesla GPUs. Although open-source libraries such as DeepSpeed [24], Megatron-LM [25], and AxoNN [26] utilize ZeRO, this study does not include a direct comparison with ZeRO-based libraries because it targets heterogeneous GPUs such as GTX, RTX, and Tesla GPUs.

III. EDDIS

A. Software architecture

EDDIS is a high-speed data-parallel deep learning library based on SoftMemoryBox [7] designed for efficient distributed

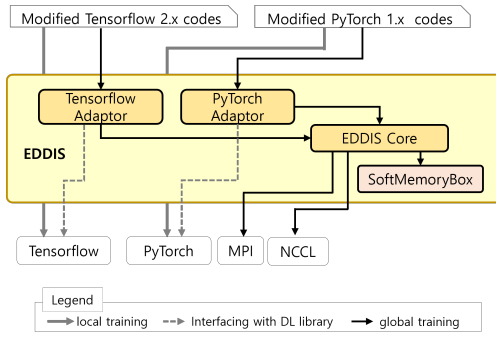


Fig. 1. Software architecture of EDDIS

DNN training in heterogeneous GPU environments. Fig. 1 shows the high-level software architecture of EDDIS. It utilizes SoftMemoryBox servers for parameter exchange and operates in conjunction with TensorFlow or PyTorch. To train DNNs in a distributed manner using EDDIS, some code must be modified using the EDDIS API. TensorFlow/PyTorch handles local parameter training, while EDDIS manages global parameter training. EDDIS comprises TensorFlow and PyTorch adaptors, EDDIS Core module, and SoftMemoryBox. The adaptors provide APIs to modify the original code for distributed training, whereas the EDDIS Core module handles parameter exchanges and updates, working with SoftMemoryBox. EDDIS Core also adjusts the training load based on the processor's computational power. EDDIS uses the SoftMemoryBox framework [6], [7] to exchange and update global parameters, and employs multiple SMB servers and RDMA to reduce communication bottlenecks in InfiniBand networks.

EDDIS utilizes SMB for asynchronous and hybrid parameter updates. The SMB framework creates an environment similar to a virtual shared memory buffer by pooling the available memory segments from remote nodes for distributed processes. Fig. 2 illustrates the parameter update method using SoftMemoryBox for distributed training in EDDIS. EDDIS consolidates all DNN parameters into a single large tensor, with the SMB framework allocating a virtual shared buffer of matching size. By pooling equally sized memory segments from remote servers, this buffer enables seamless global parameter sharing across distributed training processes. EDDIS avoids communication bottlenecks and optimizes bandwidth utilization by evenly distributing memory segments across a scalable set of SMB servers, thereby dynamically increasing the number of servers required to satisfy the bandwidth requirements for training.

B. Data-parallel training modes

EDDIS supports three data-parallel training modes for distributed training. The manner in which the distributed processes update their local and global weights varies, depending on the training mode. First, EDDIS supports a synchronous training mode using the Synchronous Stochastic Gradient Descent (SSGD) algorithm such as Horovod, PyTorch Distributed Data Parallel(DDP), or TensorFlow distribution. Fig. 3 a) illustrates the operation of the synchronous training

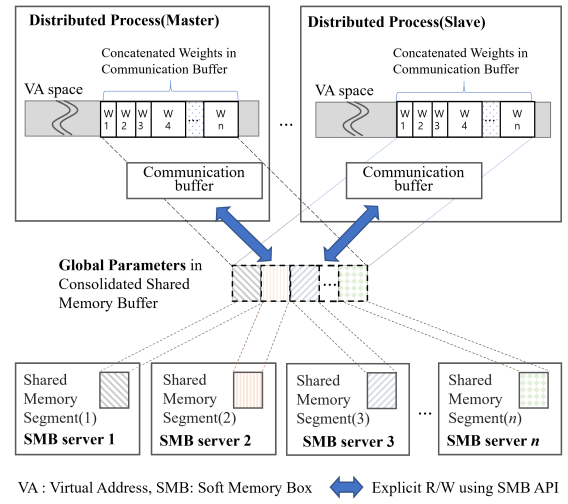


Fig. 2. The consolidated shared memory buffers are used for communicating global parameters among distributed processes.

mode(SYNC). Each distributed deep-learning process, which occupies one GPU in a computing node, trains a mini-batch with all the parameters. After calculating the gradients from the minibatch, the processes average the gradients using the AllReduce communication API of NVIDIA Collective Communications Library(NCCL) or MPI. The local weights are then updated using the averaged gradients. All processes start with the same initial weights and update the weights with the same averaged gradients; therefore, the weights of all the processes have the same values. Second, EDDIS supports an asynchronous training mode(ASYNC), as shown in Fig. 3). The asynchronous mode is particularly suitable for distributed training when GPUs are heterogeneous. It is also an algorithm that can be used in homogeneous GPU environments. Finally, EDDIS supports the hybrid training mode(HSYNC), as shown in Fig. 3 c). The HSYNC model combines the benefits of both SYNC and ASYNC training methods. In this mode, each distributed-training worker belongs to a specific subgroup. For example, in Fig. 3 c), the distributed processes within a single computing node belong to the same subgroup. The key feature of this approach is that training is performed synchronously within each subgroup, whereas training between subgroups is performed asynchronously. Fig. 3 c) illustrates the operation of the hybrid algorithm. After completing mini-batch training, the distributed processes within each subgroup exchange and average gradients using NCCL AllReduce, and then update their local weights based on the averaged gradients. The representative processes of each subgroup read the global weights from the SoftMemoryBox server, update the local weights, and then update the global weights using the differences between them.

C. Distributed optimizer algorithms

EDDIS supports a few parameter-update algorithms, such as SSGD, ASGD, SEASGD, and HEASGD. The SSGD algorithm is the only parameter update algorithm for the SYNC training mode. The others are used in the ASYNC and HSYNC

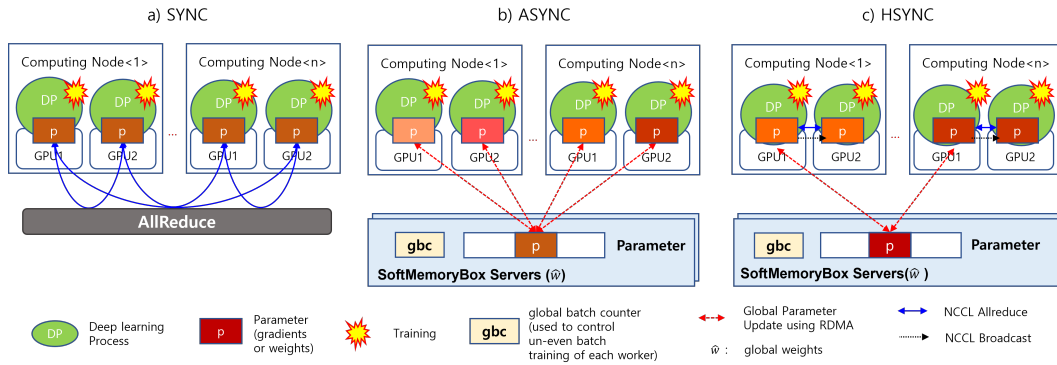


Fig. 3. Comparison of the three data-parallel training modes of EDDIS

training modes. Equation (1) is the local and global weight update formula for SSGD (w' : updated weights, w : weights, η is the learning rate, $\hat{g}$ is the averaged gradient).

$$w' = w - \eta \hat{g} \quad (1)$$

Equation (2) describes the manner in which the global ($\hat{w}$) and local weights (w_x) are updated in the ASYNC training mode (g_x : gradients of x workers).

$$\begin{aligned} \hat{w}' &= \hat{w} - \eta g_x \\ w'_x &= w' \end{aligned} \quad (2)$$

Equation (3) expresses the SEASGD algorithm [5]. Conceptually, this algorithm updates local and global weights using the EASGD algorithm. After computing the local gradients from each minibatch training, each distributed process updates its local weights ($w_x - \eta g_x$) using a locally computed gradient (g_x). Second, it updates the local weights using the weight difference ($\Delta w = w_x - \hat{w}$) between the global weights ($\hat{w}$) and local weights (w_x). After the local weights are updated, the weight difference (Δw) is uploaded to the SoftMemoryBox servers and used to update the global weights ($\hat{w}'$). The moving rate (α) is a hyperparameter that represents the extent to which the local and global weights are updated immediately from the weight differences (Δw). The SEASGD algorithm generally uses a tuned value for the moving rate (α) starting from the reciprocal of the number of workers (N), which is $1/N$.

$$\begin{aligned} w'_x &= w_x - \eta g_x - \alpha (w_x - \hat{w}) \\ \hat{w}' &= \hat{w} + \alpha (w_x - \hat{w}) \end{aligned} \quad (3)$$

Equation (4) illustrates the newly proposed HEASGD algorithm, which is a variant of SEASGD. The local weight update equation for HEASGD is the same as SEASGD. However, the value of the moving rate (α) is applied different from that used in SEASGD. The moving rate (α) is applied with the value of $(N-1)/N$. Conceptually, the global weight update of HEASGD in Equation (4) combines the global weight update methods of ASGD and SSGD. The difference from the global weight update method of ASGD in Equation (2) is that the gradients are divided by the number of workers (N) before updating the global weights.

$$\begin{aligned} w'_x &= w_x - \eta g_x - \alpha (w_x - \hat{w}) \\ \hat{w}' &= \hat{w} - \frac{\eta g_x}{N} \end{aligned} \quad (4)$$

Equation (5) represents the formula for sequential global weight updates by N workers along the time axis, starting from time t .

$$\begin{aligned} \hat{w}^{t+1} &= \hat{w}^t - \frac{\eta g_x^t}{N} \\ &\dots \\ \hat{w}^{t+N} &= \hat{w}^{t+N-1} - \frac{\eta g_x^{t+N-1}}{N} \end{aligned} \quad (5)$$

Equation (6) is the integrated formula derived from Equation (5). When the formulas for the global weight updates by N workers are combined in Equation (6), it is similar to the SSGD algorithm in Equation (1), which updates the global weights using the average weights of N workers. However, in the SSGD algorithm, each worker averaged the gradients learned from the same weights, whereas in the HEASGD algorithm, each worker averaged the gradients learned from slightly different weights. We estimated that the averaged gradients ($\hat{g}$) of SSGD and the averaged gradients ($\tilde{g}$) of HEASGD are very similar ($\hat{g} \approx \tilde{g}$). Through extensive experimentation, we aim to prove that the aforementioned estimation enables the scaling of asynchronous distributed training.

$$\hat{w}^{t+N} = \hat{w}^t - \frac{\eta (g_x^t + g_x^{t+1} + \dots + g_x^{t+N-1})}{N} = \hat{w}^t - \eta \tilde{g} \quad (6)$$

D. Full-parallel and uneven-batches training

EDDIS supports SoftMemoryBox-based, full-parallel, and uneven-batch trainable asynchronous parameter updates. EDDIS improves upon ShmCaffe [5], where computation and communication are performed in a half-parallel manner by enabling full-parallel global weight updates. In addition, it introduces the capability of distributed workers running on heterogeneous GPUs to train an uneven number of batches during each epoch. In Fig. 3 b) and c), it can be observed that a global batch counter (gbc) is allocated in the SoftMemoryBox servers and shared among the workers. This counter is atomically shared among workers via InfiniBand RDMA atomic operations, and is used to control uneven batch training.

Fig. 4 illustrates the procedure between the main thread responsible for local training and the GWU thread that handles global weight updates in SEASGD and

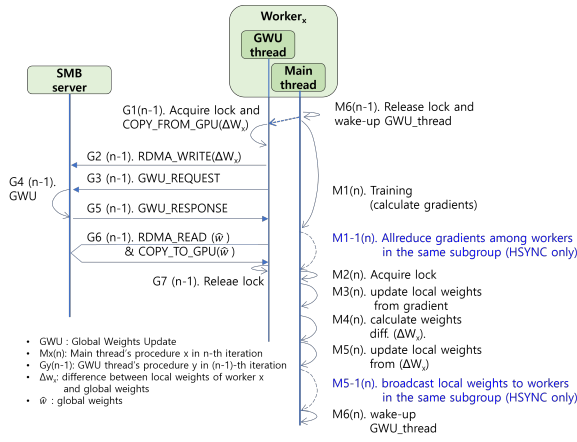


Fig. 4. The procedure of the full-parallel training and parameters updating for SEASGD and HEASGD algorithms

HEASGD algorithms. Specifically, during step G1(n-1), the COPY_FROM_GPU(Δw_x) phase involves reading Δw_x from the GPU memory of worker x into host memory. In step G6(n-1), the most recent global weights are read in advance and copied to the GPU's global weights buffer. Computation and communication were parallelized by having the GWU thread performing RDMA_READ($\hat{w}$) & COPY_TO_GPU($\hat{w}$), the computation and communication are fully parallelized. In general, the training time of the M1(n) phase is longer than that of the entire G1-7 phases. However, in cases where the time of G1-7 phases is longer than the training time of M1(n) owing to large parameters or limited network bandwidth, the main thread may need to wait until the completion of the G1-7 phases in the GWU thread. Both threads employ control locks to control the flow between them.

The procedure in Fig. 4 was applied in HSYNC training mode. However, gradient averaging among the same subgroups of workers was included, such as M1-1(n). The calculation of weight differences (Δw_x), second update of local weights, and communication with the SoftMemoryBox server for updating global weights($\hat{w}$) were performed by a representative worker of the subgroup. The representative worker broadcasts the secondarily updated local weights to the other distributed workers within the same subgroup using NCCL/MPI Broadcast in phase M5-1(n), enabling them to proceed with the training of the next mini-batch. The ideal hardware environment for applying this hybrid algorithm is when the GPU servers internally have the same type of GPUs but there are different types of GPUs across servers.

E. EddisOptimizer

The Optimizer class in TensorFlow or PyTorch is crucial for computing gradients and updating weights. EddisOptimizer is a wrapper function that takes an instance of the original optimizer class, creates a new class that extends it based on the training mode, and returns an instance of this new class. As shown in Fig. 5, EddisOptimizer outperforms the methods based on the training mode.

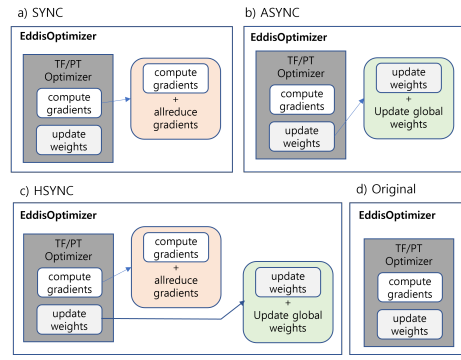


Fig. 5. The EddisOptimizer based on training modes and environment.

- SYNC mode (Fig. 5 a): Overrides *compute gradients* method to perform an all-reduce operation that averages gradients among all distributed processes.
- ASYNC mode (Fig. 5 b): Extends *update weights* method to perform a second update based on the difference between local and global weights, and triggers an update of the global weights.
- HSYNC mode (Fig. 5 c): Overrides both *compute gradients* and *update weights* methods, with the all-reduce operation averaging gradients only among distributed processes within the same subgroup.

If training is not distributed, EddisOptimizer returns the original optimizer instance without modifications (Fig. 5 d)).

IV. DISTRIBUTED TRAINING USING EDDIS

To perform data-parallel distributed training using EDDIS, researchers must modify the code for training a DNN model according to the manual modification procedures of EDDIS. The basic modification procedures for TensorFlow and PyTorch training codes are as follows.

A. Initializing and preparing distributed training

During the initialization of EDDIS, different parameter communication threads are launched depending on the training mode, with SMB initialization handled accordingly. In the SYNC mode, each process runs a main thread for local parameter training and a gradient communication thread that exchanges gradients during backpropagation to overlap communication with computation. SMB is not initialized in SYNC mode. The SMB should be initialized in the ASYNC or HSYNC modes. Each process starts with a global weight update (GWU) thread (Fig. 4) to establish reliable InfiniBand connections to the SMB servers using the SMB API [7]. Additionally, a global batch counter (gbc) is allocated and initialized to zero.

1) *Initializing codes:* First, the *eddis.tensorflow* or *eddis.torch* packages initialize the EDDIS framework by calling *eddis.init()*. Users can specify the data-parallel training mode and moving rate for ASYNC/HSYNC modes as arguments of *eddis.init()*.

```
eddis.init(training_mode=eddis.HSYNC, sgd_type=eddis.HEASGD, moving_rate=0.75).
```

2) *Adjusting learning rate*: The initial learning rate in the existing training code was optimized for training on a single GPU. Generally, in the SSGD training mode, as the number of distributed training processes increases from 1 to n , the batch size increases proportionally. Consequently, the learning rate should also be scaled by the same factors. If the training mode is ASYNC, it is not necessary to adjust the learning rate. If the training mode is HSYNC, rather than multiplying the learning rate by the total number of distributed processes n , users scale it by the number of processes within each subgroup, denoted as m . For instance, a subgroup may consist of all GPUs within a single node.

3) *Wrapping Optimizer*: Developers or researchers need to wrap the existing TensorFlow or PyTorch optimizer into the EddisOptimizer, allowing it to support the three training modes in EDDIS while preserving the default behavior of the original optimizer. Thus, the functions for the *compute gradients* and *update weights* of the original optimizer are selectively expanded based on the specified training mode in the initialization phase.

```
optimizer = optimizers.SGD(learning_rate=0.01*n, momentum
                           =0.9) # adjusted learning rate
optimizer = eddis.EddisOptimizer(optimizer) #Wrap optimizer
```

B. Controlling training loop

TensorFlow uses *tf.data.Dataset* or *tf.data.Iterator*, whereas PyTorch uses *torch.utils.data.DataLoader* to feed training data. If these classes are not used as repeat options, they stop providing mini-batches after the total number of steps required for an epoch. This study proposes a training progress control technique to enhance training efficiency in heterogeneous GPU environments. In the SSGD mode, when scaling to n training processes, each process trains the same number of mini-batches ($total_steps_per_epoch/n$). However, this method is inefficient when GPUs have different computational powers. To address this, we propose a technique that adjusts the training steps such that faster GPUs train more mini-batches, whereas slower GPUs train fewer mini-batches. This allows each distributed process to train a different number of mini-batches, thereby efficiently utilizing heterogeneous GPUs. For training codes that utilize a Data Iterator or *DataLoader*, the *DataGenerator* class is used to provide a variable number of minibatches during an epoch instead of a fixed number. This allows the distributed processes to train different numbers of minibatches. The EDDIS *DataGenerator* class is instantiated with arguments, such as the optimizer, data source, and *steps_per_epoch*, enabling it to efficiently supply varying mini-batches to the distributed processes. The *DataGenerator* instance shares the total number of minibatches trained by all distributed processes through a global batch counter on the SoftMemoryBox server. It calculates the number of averaged mini-batches trained, and continues feeding mini-batches until the average count reaches the *steps_per_epoch* specified as an argument.

1) *Instantiating DataGenerator*: To effectively utilize heterogeneous GPUs in ASYNC or HSYNC modes, users can optionally create *DataGenerator* instances as follows:

```
data_generator = eddis.DataGenerator(optimizer, data_source
                                     = train_data_iterator, steps_per_epoch =
                                     iterations_per_epoch // eddis.size())
```

2) *Modifying training loop*: To enable heterogeneous GPU-based distributed training, as previously mentioned, users must modify the training loop using an instance of *DataGenerator*.

```
for i (images and labels) in data_generator:
```

C. Other modification procedures

1) *Modifying validation codes*: Typically, training codes include codes for evaluating the trained model on the validation data at specific intervals, such as after each training epoch. Because it is not necessary for all distributed processes to execute these validation codes, users can restrict them to only the master process. In this case, although the training time is reduced to $1/n$ using distributed training, the validation time remains unchanged. However, it is possible to reduce the training time further by dividing the validation computation to distributed processes.

2) *Restricting codes for master process*: The training activities that should be performed only by the master (or root) process include saving or loading the checkpoint, standard output and logging, and validation.

D. Starting and stopping distributed training

EDDIS runs distributed DNN training processes concurrently using MPI, unlike PyTorch DDP and TensorFlow, where each distributed process must be launched separately. The *mpirun* command in MPI allows running a large number of distributed processes simultaneously, with the added advantage of terminating them all at once if necessary during distributed training, such as when DNN training is not progressing as expected because of incorrect hyperparameters. Furthermore, in the event of abnormal termination of any distributed process, the remaining processes can be terminated simultaneously, preventing a scenario in which the DNN training stalls while still occupying GPU resources.

V. EXPERIMENTAL STUDY

A. Environments

1) *Hardware environments*: We used nine Supermicro 4028GR-TRT2 servers, four 4124GS-TNR servers, two 7048GR-TR servers, and an ASUS ESC4000A server. The 4028GR-TRT2 servers are equipped with two Intel Xeon E5-2690 v4 CPUs (14 cores, 2.3 GHz), 128 GB DDR4-2400 MHz memory, and four Nvidia GPUs. The 4124GS-TNR servers featured two AMD EPYC™ 7343 CPUs (16 cores, 3.2 GHz) and 256 GB DDR4-3200 MHz memory. The 7048GR-TR servers had two Intel Xeon E5-2620 v4 CPUs (8 cores, 2.1 GHz) and 128 GB DDR4-2400 MHz memory. The ESC4000A server has an AMD EPYC 7313P CPU (16 cores, 3.0 GHz) and 128 GB DDR4-3200 MHz memory. The GPUs used were

NVIDIA GTX Titan Xp, GTX Titan V, RTX 2080TI, RTX Titan, RTX 3090, Tesla P100, and Tesla A100. Each server was connected via an HDR InfiniBand HCA (200 Gbps) to a Mellanox QM8700 InfiniBand switch.

2) *Software environments*: The operating system was Ubuntu 20.04, and the Linux kernel version was 5.4.0-135-generic. We used CUDA 11.4 (including cudnn 8.2.4), TensorFlow 2.5, PyTorch 1.10.1, EDDIS v1.5 and USMB v2.0.

3) *DNN & dataset used in the experiments*: For this experiment, we selected two convolutional neural networks (CNNs): ResNet50 [27] for image classification and YOLOv5 [28]–[31] for object detection. ResNet50 is one of the most widely used DNNs for evaluating the parallel training of distributed data [32]–[35]. Many researchers have demonstrated their work by showcasing the results of the distributed training of ResNet50 using the ILSVRC 2012 ImageNet dataset [36]. In this study, YOLO, which has gained the widest user base as an object detection model, was evaluated by conducting distributed training on the YOLOv5 medium [31] using the COCO dataset [37].

B. Comparison of training time among three training modes

Fig. 6 compare the training times of ResNet50 implemented in TensorFlow using the three EDDIS training modes: SYNC, ASYNC, and HSYNC. The experiment began with a single node and was scaled up to 16 nodes, doubling the number of nodes at each step. The graph shows the training times for 100 epochs for the Imagenet1K dataset.

Fig. 7 compares the training times for YOLOv5m implemented in PyTorch for the three EDDIS training modes. This graph shows the training times for 300 epochs on the COCO dataset, using the same hardware environment as that in Fig. 6. Uniform batch sizes were used in all the experiments (ResNet50: 64, YOLOv5m: 16), and SGD was used as the local optimizer. There were five for ResNet50 and three for YOLOv5m, with a weight decay of $5e-4$ for YOLOv5m. The learning rates, weight decay (for ResNet50), and moving rates (only for ASYNC and HSYNC) were tailored to the node count for each experiment, as listed in Tables I and II.

TABLE I
HYPER-PARAMETERS FOR RESNET50 TRAINING

Training mode	weight decay	learning rate(moving rate) (#Nodes)				
		1	2	4	8	16
SYNC	1e-5	0.4	0.8	1.6	3.2	6.4
ASYNC	1e-4	0.1 (0.75)	0.2 (0.875)	0.2 (0.938)	0.2 (0.969)	0.2 (0.984)
HSYNC	1e-4	0.2 (-)	0.2 (0.5)	0.2 (0.75)	0.4 (0.875)	0.6 (0.938)

SYNC with SSGD and ASYNC/HSYNC with HEASGD

TABLE II
HYPER-PARAMETERS FOR YOLOV5M TRAINING

Yolov5m	learning rate	Moving rate (#Nodes)				
		1	2	4	8	16
SYNC	$0.01 \times \#N^1$	-	-	-	-	-
ASYNC	0.01	0.25	0.125	0.0625	0.0625	0.035
HSYNC	0.01x4	-	0.5	0.25	0.15	0.1

¹ #N : Number of Nodes(each node has 4 GPUs)

The experiment was conducted using 16 nodes with heterogeneous GPUs. The baseline training time for a single

node was determined by averaging the times across all the nodes, whereas for two, four, and eight nodes, the average training time was calculated by grouping nodes with different heterogeneous GPU combinations into two or more sets. Fig. 6 and Fig. 7 displays the training speedup as a line graph, illustrating the improvement when using 2, 4, 8, and 16 nodes compared to a single node.

1) *Synchronous training*: According to Fig. 6 a), and Fig. 7 a), the training time for ResNet50 using the SYNC mode on one node was approximately 59 h and that for YOLOv5m was 62 h and 53 min. With 16 nodes, the training times for ResNet50 was reduced to approximately 7 h and 1 min and for YOLOv5m, it was 7 hours and 48 min, respectively. times higher than speedups of 8.39 times for ResNet50 and 8.06 times for YOLOv5m. Despite utilizing 16 times the resources, the scaling efficiencies were approximately 52.4% and 50.3%, respectively, owing to GPU heterogeneity.

2) *Asynchronous training*: Fig. 6 b) and Fig. 7 b) shows the results obtained using the ASYNC training mode. For 16 nodes, the training times for ResNet50 was approximately 3 h and 13 min, whereas for YOLOv5m, it was 3 h and 6 min, respectively. This corresponds to speedups of 20.1 times for ResNet50 and 17.8 times for YOLOv5m, with scaling efficiencies of 125.5% and 111%, respectively. Scaling efficiency cannot technically exceed 100%, but in this experiment, it did due to the “fallacy of average”. The baseline was set using the average training speed across all the nodes, which underestimated the potential performance. During scaling, faster GPUs handled more work, resulting in scaling efficiencies that exceed 100%. This outcome was further attributed to EDDIS’s ability of EDDIS to effectively utilize heterogeneous GPU nodes through uneven batch training, leading to even higher scaling efficiencies.

3) *Hybrid training*: Fig. 6 c) and Fig. 7 c) shows the results for the HSYNC mode. On 16 nodes, the training times for ResNet50 was approximately 3 h and 22 min and for YOLOv5m, it was 3 h and 9 min, respectively. This results in speedups of 17.5 times for ResNet50 and 19.9 times for YOLOv5m compared to a single node. The scaling efficiencies were approximately 109.3% and 124.5%, respectively. HSYNC mode is more than twice as fast as SYNC mode, with training times on 16 nodes being 2.08 times and 2.46 times faster for ResNet50 and YOLOv5m, respectively.

C. Comparison of scaling performance between SEASGD and HEASGD algorithms

Through our experiments, we confirmed that the deep-learning performance (e.g., accuracy) of the previously proposed SEASGD algorithm degraded significantly as the number of workers increased. Fig. 8 a) depicts the training performance (Top-1 Accuracy) of ResNet50 when trained using the SEASGD algorithm in ASYNC training mode. As the number of training workers increased from one node(4GPUs) to 16 nodes(64GPUs), the training performance degraded significantly. A previous study mitigated this issue to a certain extent by introducing the HSYNC training mode [5]. This

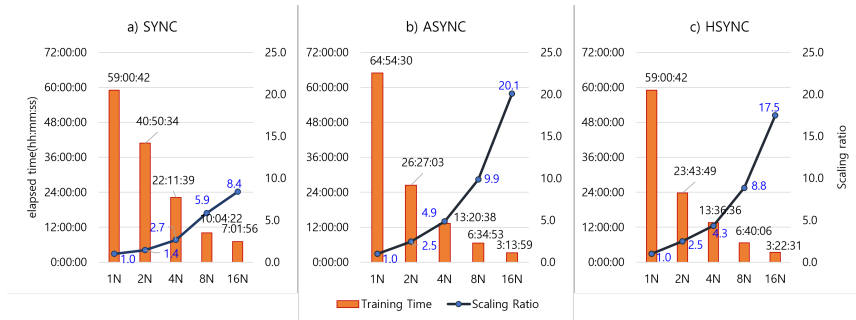


Fig. 6. Comparison of EDDIS (TensorFlow) Training Time of ResNet50

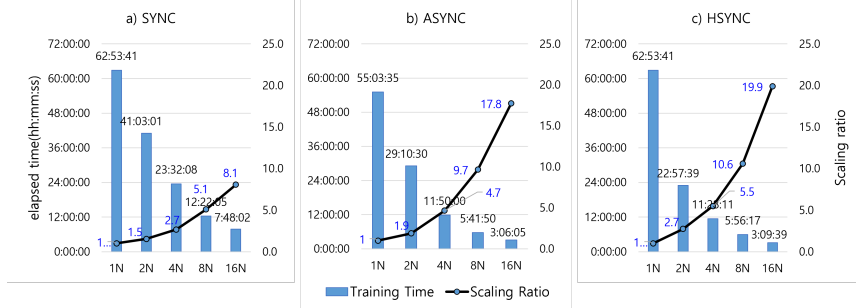


Fig. 7. Comparison of EDDIS (PyTorch) Training Time of Yolov5

is because in HSYNC mode, the training workers are sub-grouped, reducing the number of workers updating the global weights asynchronously from 64 to 16. Fig. 8 b) illustrates the training performance when using the SEASGD algorithm in HSYNC mode. It can be observed that in a 16-node, 64-GPU environment, the performance degradation in ASYNC mode was 14.1%, whereas in HSYNC mode, it decreased by only 0.68% as shown in Fig. 8 b). However, even when training in HSYNC mode, there is an increased performance degradation as the scaling factor increases. This study addresses this issue by using the newly proposed HEASGD algorithm. Fig. 8 c) shows the performance when trained using the HEASGD algorithm in ASYNC training mode. In a 16 nodes 64 GPUs environment, the HEASGD algorithm experienced a 1.08% degradation in training performance. As seen in Fig. 8 d), in HSYNC mode, the performance degradation is limited to only 0.27%. Fig. 9 directly compare the SEASGD and HEASGD algorithms. Looking at Fig. 9 a), it can be seen that the HEASGD algorithm is 12.3% better than the SEASGD algorithm in ASYNC training mode, and Fig. 9 b) shows that HEASGD outperformed SEASGD by 0.41% in HSYNC training mode.

D. Comparison of accuracy among three training modes

Fig. 10 compares the top-1 accuracy of ResNet50 across the three training modes: SYNC, ASYNC, and HSYNC. In the SYNC mode, the SSGD algorithm was used, whereas HEASGD was employed in both ASYNC and HSYNC modes. As the number of workers increased, the training performance tended to degrade across all training modes. According to Fig. 10, in SYNC mode, the training performance decreases by 0.34% as the number of nodes increases from 1 to 16. Although ASYNC mode shows a better performance than the

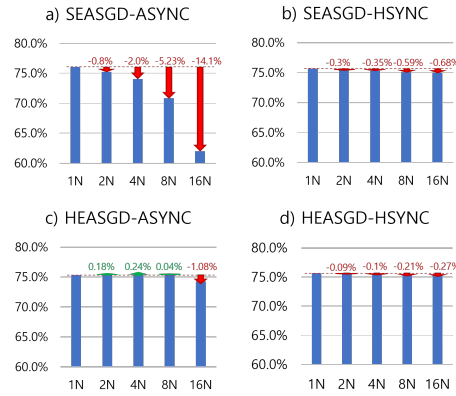


Fig. 8. Comparison of scaling performance of distributed optimizer algorithms (SEASGD and HEASGD) in ASYNC and HSYNC training modes

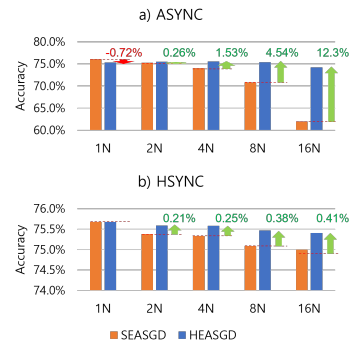


Fig. 9. Direct Comparisons of performance between SEASGD and HEASGD algorithms

SYNC mode with two, four, and eight nodes, its performance drops below that of the SYNC mode when the number of nodes it reaches is 16. By contrast, HSYNC mode consistently

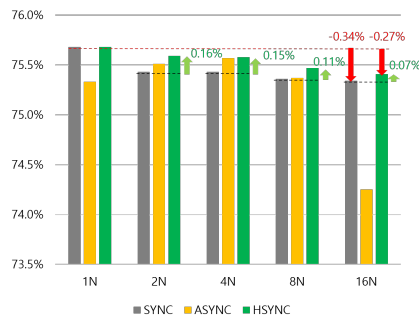


Fig. 10. Comparison of EDDIS Validation Results(Top-1 Accuracy)

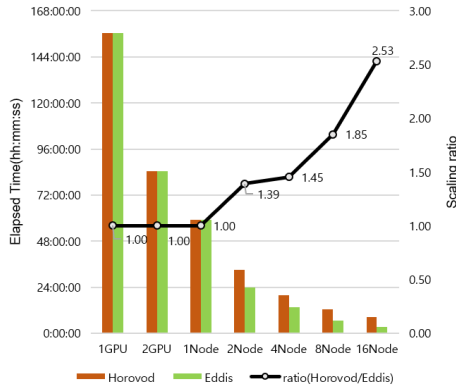


Fig. 11. Comparison of Tensorflow Resnet50 Training Time between Horovod and EDDIS(HSYNC)

achieves better performance than the SYNC mode, even as the number of nodes increases. This experiment was conducted to compare the performances across the three training modes. The lower performance observed compared with the model's best-known performance is due to the lack of optimizations, such as data augmentation, in the training process.

E. Training comparison of EDDIS, Horovod, and DDP

Fig. 11 compares the training times of the Tensorflow ResNet50 model with those of the baseline(1GPU), Horovod (HVD), and EDDIS(HSYNC) models when training the ResNet50 model on the ImageNet1K dataset for 100 epochs. EDDIS completed training 46.36 times faster than the baseline, with a scalability efficiency of 72.43%. EDDIS also achieved 2.53 times faster as Horovod in a 16-node, 64-GPU environment. The final training accuracy was 76.63% for the baseline, 75.42% for EDDIS, and 73.4% for Horovod, with EDDIS showing a 2% better training performance than Horovod.

Fig. 12 a) compares the training times of the YOLOv5m model among the baseline(1GPU), PyTorch Distributed DataParallel(DDP), Horovod(HVD), and EDDIS(only SYNC mode). Fig. 12 b) compares the training times of the YOLOv5m model using the three training modes, DDP, HVD and EDDIS, on 16 nodes with 64 heterogeneous GPUs. The EDDIS SSGD training mode achieved a similar training time to Horovod and was 1 h and 20 min faster than DDP. EDDIS HSYNC training mode is 2.8x and 2.3x faster than DDP and HVD respectively in training the YOLOv5m model with 300 epochs on the COCO dataset. We achieved a similar training

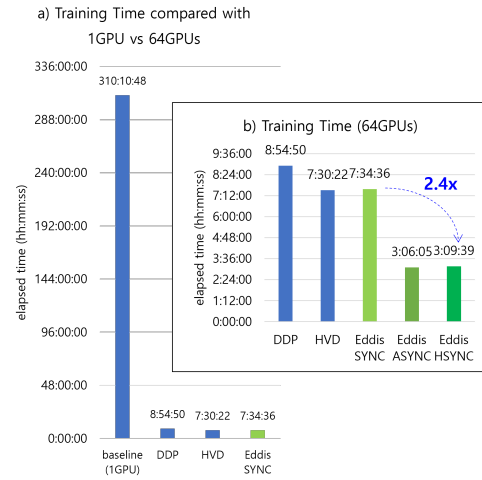


Fig. 12. Comparison of YOLOv5m Training Time among a) PyTorch(1GPU, baseline), EDDIS(SYNC), DDP and Horovod(16 Nodes, 64 Heterogeneous GPUs), b) DDP, Horovod, EDDIS(SYNC, ASYNC, HSYNC)

performance (mAP) using EDDIS. When training the YOLOv5m model, the baseline trained on one GPU achieved 0.441, DDP trained on a 16-node 64-GPU environment achieved 0.429, Horovod achieved 0.426, and EDDIS achieved 0.431.

VI. CONCLUSION

In this study, we propose EDDIS, which addresses the communication bottleneck in distributed deep neural network (DNN) training and efficiently utilizes heterogeneous GPUs to significantly speed up training. EDDIS offers three training modes—SYNC, ASYNC, and HSYNC—allowing the selection of the optimal mode based on the combination of available GPUs. This approach facilitates the distributed training of DNNs written in TensorFlow and PyTorch with minimal modifications to the existing code. EDDIS has strong node scalability and operates by increasing the number of SMB memory servers to secure the required bandwidth as the communication volume increases with the expansion of the training nodes. EDDIS also fully overlaps computation and communication times, significantly reducing the synchronization time between nodes.

EDDIS achieves approximately 8.2 times, 19, and 18.7 times faster training compared to using a single node when employing 16 nodes in the SYNC, ASYNC, and HSYNC modes, respectively. Notably, EDDIS(HSYNC) delivers training speeds that are 2.8x and 2.3x faster than DDP and Horovod, respectively. To address the scaling limitations of the existing SEASGD algorithm, we introduced a new HEASGD algorithm. The HSYNC mode in HEASGD further improves performance over the SYNC mode. EDDIS leverages MPI to enable concurrent deep learning process execution and termination, but requires uniform hyperparameters across processes. This limitation restricts the batch size to the GPU with the smallest memory in heterogeneous setups, potentially underutilizing GPUs with a larger memory. EDDIS partially addresses this by enabling the asynchronous training of more batches on GPUs with larger memory.

In future research, we plan to adapt asynchronous algorithms for heterogeneous GPU environments with varying numbers of GPUs per node and evaluate the performance of distributed training. Additionally, to address the convergence issues in the HSYNC method, we will enhance EDDIS scalability by extending the unit of subgroups from a single node to a multi-node configuration. Furthermore, experiments will be conducted on large language models (LLMs) such as BERT within the EDDIS framework, followed by further optimization based on the results.

REFERENCES

- [1] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *CoRR*, vol. abs/2001.08361, 2020.
- [2] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, (Broomfield, CO, USA), pp. 571–582, 2014.
- [3] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*, (Bellevue, Washington, USA), pp. 265–272, 2011.
- [4] S. Bianco, R. Cadène, L. Celona, and P. Napolitano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64270–64277, 10 2018.
- [5] S. Ahn, J. Kim, E. Lim, W. Choi, A. Mohaisen, and S. Kang, "Shmcaffe: A distributed deep learning platform with shared memory buffer for hpc architecture," in *Proceedings of the 38th IEEE International Conference on Distributed Computing System (ICDCS'18)*, (Vienna, Austria), July 2 - 5, 2018.
- [6] S. Ahn, J. Kim, E. Lim, and S. Kang, "Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing," *IEEE Access*, vol. 6, pp. 26493–26504, 2018.
- [7] S. Ahn and E. Lim, "Softmemorybox ii: A scalable, shared memory buffer framework for accelerating distributed training of large-scale deep neural networks," *IEEE Access*, vol. 8, pp. 207097–207111, 2020.
- [8] E.-J. Lim, S.-Y. Ahn, Y.-M. Park, and W. Choi, "Distributed deep learning framework based on shared memory for fast deep neural network training," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1239–1242, 2018.
- [9] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09*, (New York, NY, USA), p. 873–880, Association for Computing Machinery, 2009.
- [10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of 26th Annual Conference on Neural Information Processing (NIPS'12)*, (Lake Tahoe, NV, USA), pp. 1232–1240, December 3–6, 2012.
- [11] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, 2014.
- [12] W. Wang, G. Chen, H. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K. Tan, S. Wang, and M. Zhang, "Deep learning at scale and at ease," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 12, no. 4s, pp. 69:1–69:25, 2016.
- [13] J. Wang and L. Cheng, "Distdl: A distributed deep learning service schema with GPU accelerating," in *Proceedings of the 17th Asia-Pacific Web Conference on Web Technologies and Applications (APWeb'15)*, (Guangzhou, China), pp. 793–804, September 18–20, 2015.
- [14] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *Proceedings of the International Conference on Learning Representations (ICLR) Workshop Track*, 2016.
- [15] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, (Broomfield, CO, USA), pp. 583–598, October 6–8, 2014.
- [16] B. T. Polyak and A. B. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM J. Control Optim.*, vol. 30, p. 838–855, July 1992.
- [17] S. Zhang, A. E. Choromanska, and Y. LeCun, "Deep learning with elastic averaging sgd," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 685–693, Curran Associates, Inc., 2015.
- [18] S. Gupta, W. Zhang, and F. Wang, "Model accuracy and runtime tradeoff in distributed deep learning: A systematic study," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 171–180, 2016.
- [19] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, Aug. 2019.
- [20] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, and Y. Kageyama, "Imagenet/resnet-50 training in 224 seconds," *CoRR*, vol. abs/1811.05233, 2018.
- [21] Y. Ding, N. Botzer, and T. Weninger, "Hetseq: Distributed gpu training on heterogeneous infrastructure," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 15432–15438, May 2021.
- [22] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the 25th Advances in Neural Information Processing Systems (NIPS'11)*, (Granada, Spain), pp. 693–701, 12–14 December 2011.
- [23] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," *CoRR*, vol. abs/2101.06840, 2021.
- [24] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, (New York, NY, USA), Association for Computing Machinery, 2021.
- [25] D. Narayanan, M. Shoenybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21*, (New York, NY, USA), Association for Computing Machinery, 2021.
- [26] S. Singh and A. Bhatle, "Axonn: An asynchronous, message-driven parallel framework for extreme-scale deep learning," 2023.
- [27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, (Las Vegas, NV, USA), pp. 770–778, June 27–30, 2016.
- [28] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.
- [29] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.
- [30] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," 2022.
- [31] J. Glenn, "Yolov5 release v6.1." <https://github.com/ultralytics/yolov5/releases/tag/v6.1>.
- [32] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.
- [33] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018*, (New York, NY, USA), pp. 1:1–1:10, ACM, 2018.
- [34] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, et al., "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," in *Proceedings of Workshop on Machine Learning Systems in The 32th Annual Conference on Neural Information Processing Systems. IEEE*, 2018.
- [35] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *ArXiv*, vol. abs/1811.06992, 2018.
- [36] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition (CVPR) 2009. IEEE Conference on. IEEE*, 2009.
- [37] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014.