AIACC-Training: Optimizing Distributed Deep Learning Training through Multi-streamed and Concurrent Gradient Communications

Lixiang Lin†, Shenghao Qiu*, Ziqi Yu†, Liang You†, Long Xin†, Xiaoyang Sun*, Jie Xu*, Zheng Wang*
†Alibaba Group, *University of Leeds

***Abstract*—There is a growing interest in training deep neural networks (DNNs) in a GPU cloud environment. This is typically achieved by running parallel training workers on multiple GPUs across computing nodes. Under such a setup, the communication overhead is often responsible for long training time and poor scalability. This paper presents AIACC-Training, a unified communication framework designed for the distributed training of DNNs in a GPU cloud environment. AIACC-Training permits a training worker to participate in multiple gradient communication operations simultaneously to improve network bandwidth utilization and reduce communication latency. It employs auto-tuning techniques to dynamically determine the right communication parameters based on the input DNN workloads and the underlying network infrastructure. AIACC-Training has been deployed to production at Alibaba GPU Cloud with 3000+ GPUs executing AIACC-Training optimized code at *any time*. Experiments performed on representative DNN workloads show that AIACC-Training outperforms existing solutions, improving the training throughput and scalability by a large margin.**

***Index Terms*—Distributed deep learning, Model training, Communication optimization**

I. Introduction

Deep neural network (DNN) training is an important application workload on GPU clouds. Because a DNN model is often trained over a large number of samples, distributed deep learning (DDL) is widely used to reduce the training time by parallelizing the training workload across multiple GPUs.

Data parallelism[1] is a common parallelization strategy for DDL [1], [2]. This is achieved by partitioning the training samples across parallel training workers, where each worker processes a subset of the training data. During each training iteration, all workers combine the results of their computation (i.e., the *local gradient*) to produce an aggregated gradient to update the model parameters stored on distributed computing devices before the next training iteration. Aggregation of gradients requires training workers to communicate and exchange their local gradients via the communication network.

The volume of gradients to be exchanged among training workers is proportional to the number of model parameters and the tensor size of individual parameters. Because the size of new DNN models is increasing at a much faster pace than the increased hardware performance [1], [3], gradient communication has become a major performance bottleneck in DDL [4], [5]. Recently, efforts have been made to optimize distributed gradient communications by exploiting heterogeneous communication links [6] or additional CPU servers [2]. Other works apply gradient compression methods to reduce data transfer size by using a lower precision representation of gradients [7], [8]. Major deep learning frameworks like PyTorch and DDL libraries like Horovod [9] also support the decoupling of gradient communication from computation to overlap the communication with computation.

While promising, exciting solutions all fail to capitalize on the large network bandwidth in a modern cloud environment. As we will show later in the paper, in real-life scenarios, the state-of-the-art distributed communication framework for DDL may utilize up to 30% of the available bandwidth of a standard TCP/IP network in the GPU cloud. The poor network bandwidth utilization, in turn, leads to increasingly poorer scalability when more GPUs are used.

After running Alibaba's public GPU cloud and internal servers for several years, we have observed many examples where DDL performance suffers from poor gradient communication efficiency. While reducing the training time is crucial for many users, most data scientists and GPU cloud users are not expert programmers and are unfamiliar with distributed communication optimization. This motivates us to design AIACC-Training, a unified distributed communication library to support efficient gradient communications. AIACC-Training supports mainstream deep learning frameworks like Tensorflow [10], PyTorch [11], and MXNet [12] with Horovod-like API. It lowers the programming barrier by automatically converting a sequential DNN code running on a single GPU to an optimized DDL program *with zero user involvement*. As a unified communication library, AIACC-Training provides a single, highly-optimized framework to meet diverse DNN workloads while simplifying the library maintenance cost.

A key innovation of AIACC-Training is adopting a multi-streamed gradient communication strategy to improve network bandwidth utilization. With prior work, the training worker only participates in a gradient communication operation at a time. Because a single communication process cannot fully utilize the network bandwidth, existing solutions leave much room for improvement. Our work is based on the observation that gradients are often produced faster than their exchange speed, but a single communication stream cannot fully utilize the network bandwidth offered by modern cloud infrastructures. By allowing a training worker to participate in multiple concurrent gradient communication operations, we can better utilize the network bandwidth to reduce communication latency to improve the training throughput and speed. AIACC-Training achieves this by carefully packing the computed gradients to multiple communication units – each is handled by a concurrently running communication thread.

AIACC-Training develops an auto-tuning technique to find

[1]Other DDL approaches include model, pipeline, and asynchronous-data parallelism. While these are supported by AIACC-Training, they are not as common as data parallelism and hence are not the focus of this paper.

the optimal number of concurrent communication threads and the gradient communication granularity, depending on DNN workload and network topology (that can vary during runtime). Specifically, AIACC-Training formulates the parameter selection problem as a multi-armed bandit problem [13]. It then uses a carefully designed meta solver to automatically determine the right parameter setting within a search time budget during the initial warm-up phase. Crucially, the results of parameter search also contribute to the final training outcome, so no computation cycle is wasted.

AIACC-Training was the first unified communication library that supports multiple deep learning frameworks in a single infrastructure. It is compatible with the Horovod API for DDL and provides a source-to-source tool to translate the sequential model code for DDL automatically. It supports communication optimization techniques like gradient compression and can be used with data, model and pipeline parallelisms or a mixture of these parallelization strategies. It provides a new parameter optimizer to improve the training speed.

We evaluate AIACC-Training on representative DNN models and datasets, including a production DNN system. Experimental results show that AIACC-Training consistently outperforms existing approaches, improving the training throughput by up to 3.3x on public DNNs using 256 GPUs (and 13.4x on an internal production DNN system). As a major online service and cloud provider, we have deployed AIACC-Training on Alibaba's internal GPU servers and public GPU cloud, with over 3000 GPUs executing a diverse set of AIACC-Training optimized workloads at *any time* on the Alibaba GPU cloud.

This paper makes the following contributions:

- It presents a new gradient communication scheme for accelerating DDL (Section V);
- We demonstrate how auto-tuning techniques can be employed to optimize hyperparameters for a DDL communication library (Section VI);
- We present a summary of the critical design decisions, and quantitative analysis of observations learned from operational experience in production environments when developing a DDL communication library.

II. Background

A. DNN Model Training

DNN training often consists of millions of iterations across multiple training epochs. An iteration processes a small part of the entire training data, known as *minibatch*. Each training iteration contains three stages: forward, backward and parameter update. During the forward stage, the training samples (e.g., images or sentences) are passed through the DNN layers to compute a loss (or error) using an objective function (or loss function). During the backward stage, the loss is backwards propagated through the DNN layers to compute the *gradients*. At the parameter update stage, an *optimizer* then updates the model weights based on the gradients. Training a DNN involved iteratively updating perform these three steps.

During backward propagation, gradients are computed along the reversed direction of the network, starting from the output layer. A network layer can produce more than one gradient, e.g., a linear layer - $y = ax + b$ - will produce two gradients: one for the weight, a, and one for the bias, b, where each gradient is a layer-dependent tensor (a multi-dimensional array). Our work focuses on optimizing gradient communications at backward propagation, which is well-known to be the major performance bottleneck of DDL [1], [2], [4], [5].

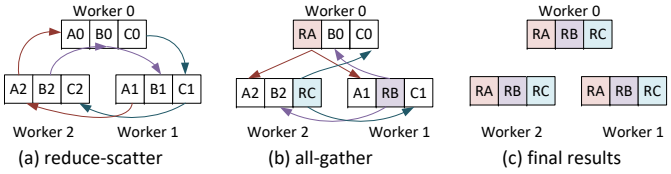


Fig. 1: The ring all-reduce operation.

B. Data Parallelism

Data parallelism is a mainstream paradigm for DDL [2]. This is achieved by partitioning and distributing the training samples across different training workers running on different GPUs, where each GPU holds a complete DNN model (and its parameters). Since each training worker works on a subset of the training data, the gradients generated by different training workers will be different. At the parameter update stage of a training iteration, gradients from different training works need to be aggregated to update the model weights before the next training iteration. This is achieved by either applying an `all-reduce`[2] operation across parallel processes or using a *parameter server* [14] to aggregate the gradients from different processes. This procedure is network IO intensive as it requires performing data communications and synchronization across different GPUs and computing servers. Furthermore, all-reduce is the most popular gradient communication scheme due to its higher performance over parameter servers.

C. All-reduce

The ring all-reduce [15] is a dominant approach for implementing all-reduce for DDL. Fig. 1 depicts the process of applying ring all-reduce to three parallel workers. At the *reduce-scatter* stage, the ready gradients are partitioned into n chunks (where n is the number of parallel workers), creating n rings with different starting and ending points (e.g., $C0 \rightarrow C1 \rightarrow C2$ in Fig. 1a). Each data chunk is sent along a ring. When a worker receives the data from another worker, it will apply a reduced operator and then proceed to send the reduced data to the next worker in the ring. The reduce-scatter phase finishes when each worker holds the complete reduction of chunk i. In the *all-gather* step (Fig. 1b), each worker broadcasts the completely reduced chunk (e.g., RA) to all other workers. At the end of reduce-gather (Fig. 1c), all workers will have the complete set of reduced data.

Most distributed training frameworks adopt the ring all-reduced. However, as we will show in Section VIII, existing implementations give poor bandwidth utilization, leaving much room for improvement in a GPU cloud environment.

[2]All-reduce performs a chosen reduction operator (e.g., sum, min, max) on data across parallel workers and then sends the global result to *all* workers.

D. CUDA Streams

Although numerous neural network accelerators have been developed [16], [17], the NVIDIA GPU remains the de-facto platform for training DNNs due to its availability and matured software ecosystem. For this reason, our work primarily targets NVIDIA GPUs, but our techniques can also be transferred to other architectures (e.g., the Alibaba NPU).

NVIDIA GPUs consist of a large number of processing units, which are organized as streaming multiprocessors (SMs). For example, the NVIDIA Tesla V100 GPU supports 80 SMs, where each SM has a fixed number of cores. In the CUDA programming model, instructions placed within a single CUDA stream are executed sequentially, following their issued order. However, code offloaded to different CUDA streams can be dispatched by the GPU instruction scheduler to different hardware SMs to be executed *concurrently* on the same GPU. As a departure from all distributed communication libraries, AIACC-Training utilizes multiple CUDA streams to perform concurrent gradient communications for a GPU worker during backward propagation. The GPU hardware scheduler automatically schedules a certain number of CUDA streams to run on multiple SMs, depending on hardware resource contention. AIACC-Training uses an auto-tuning technique (Section VI) to determine the optimal number of CUDA streams for gradient communications.

E. Distributed Communications in GPU Clouds

TCP/IP network. Like most cloud providers, Alibaba cloud instances are organized as a virtual private cloud (VPC) to provide a private communication tunnel for a user. VPC builds upon the traditional TCP/IP network and tunnel technology and is commonly available and low cost to both the cloud provider and end-user. AIACC-Training is designed to optimize gradient communications over the TCP/IP network because it remains the dominant communication infrastructure in public GPU clouds.

RDMA. GPUs within a single computing host can communicate via NVIDIA's Nvlink [18], [19] or PCIe. GPUs *across computing nodes* can communicate via either remote direct memory access (RDMA) or a TCP/IP network. However, GPU RDMA requires deploying dedicated host bus adaptors and network adaptors and switches, e.g., InfiniBand and NvSwitch for NVIDIA GPUs [20]. While RDMA is generally faster than a TCP/IP network, it incurs significant infrastructure and operational costs over the TCP/IP solution. As a result, not all Alibaba GPU cloud servers are equipped with RDMA components (indeed, most cloud providers do not promise RDMA). As we will show later, AIACC-Training can also improve the use of RDMA when it is available.

III. Motivation

The initial design of AIACC-Training was to utilize Horovod [9], a popular DDL library, for distributed communications. However, we found that Horovod (and the underpinning NVIDIA Collective Communications Library - NCCL) gives poor scalability in a typical GPU cloud environment.

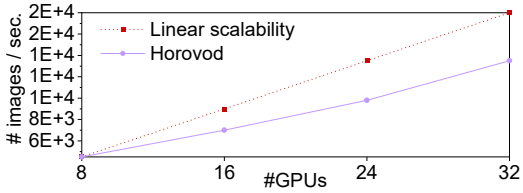


Fig. 2: The training throughput delivered by Horovod versus the theoretical linear speedup.

As a motivation example, consider Fig. 2. This diagram compares the throughput (i.e., the number of training images processed per second) when applying Horovod to train the ResNet-50 DNN [9] using multiple GPUs. In this example, each GPU server has 8x 32GB NVLink-enabled NVIDIA V100 GPUs, and servers are connected through a 30Gbps TCP/IP network. We compare the Horovod achieved throughput against a theoretically perfect linear improvement as we increase the number of GPUs. While using more GPUs leads to higher throughput, Horovod gives a *scaling efficiency*[3] of 75% when using 32 GPUs, exhibiting poor scalability. We also observe a similar scalability issue on the Pytorch and MXNet distributed training engine, which gives a scaling efficiency of less than 79% under the same setting. We stress that such a poor scaling efficiency is not unique to a single DNN. For example, for VGG-16 and BERT, another two popular DNN architectures, Horovod gives a scaling efficiency of 40% under the same setup. The scaling efficiency also further deteriorates when using more GPUs.

After a close examination, we found that the poor scaling efficiency is largely due to two fundamental drawbacks. First, existing distributed training frameworks only utilize a single communication link for gradient synchronization. Unfortunately, a single communication stream can only utilize at most 30% of the bandwidth provided by the TCP/IP link (and can be as low as 10% to 5% of RDMA). Such under-utilization leads to long gradient communication latency, causing frequent GPU stalls and wasting the expensive computation cycles. This is a massive missed opportunity. Secondly, existing all-reduce-based approaches require a single master node (i.e., a synchronization point) to ensure all parallel workers have produced the required gradients. We observe from real-life use cases that the master node can quickly become a bottleneck as the number of GPUs increases (e.g., when using more than 128 GPUs), further deteriorating the scalability.

In light of these observations, AIACC-Training aims to provide a fully decentralized gradient communication scheme by utilizing multiple communication streams. As we will show in Section VIII, AIACC-Training gives a scaling efficiency of over 0.96, leading to 1.3x and 1.8x improvement over Horovod on ResNet-50 and VGG-16 respectively with 32 GPUs, and 3.3x improvement with 256 GPUs.

[3]We use the definition in [4] where the scaling efficiency is computed as T_N/N_T. Here, T_N is the single GPU throughput and N_T is the measured throughput when using N training workers (GPUs).

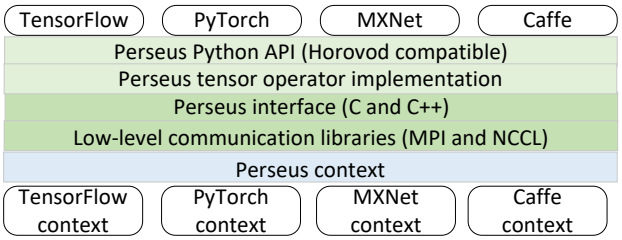


Fig. 3: The AIACC-Training software stack.

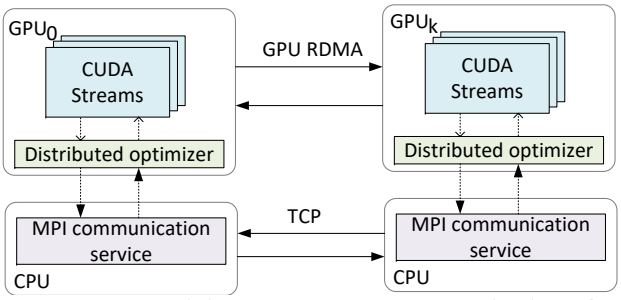


Fig. 4: AIACC-Training components. Communication of each GPU is managed by an MPI process running on the CPU.

IV. Overview of AIACC-Training

Fig. 3 gives an overview of the AIACC-Training stack, a DDL component of Alibaba's AIACC framework[4]. It is designed to optimize DDL in a cloud environment with TCP and RDMA links. It supports multiple deep learning frameworks: Tensorflow, Pytorch, MXNet and Caffe. AIACC-Training provides a unified communication API (named Perseus) to all supported programming models.

The core idea of AIACC-Training is to implement a fully decentralized and concurrent all-reduce based gradient communication scheme. Communication concurrency is realized by employing a fine-grained gradient partitioning strategy to allow a training worker to participate in multiple all-reduce communications at the same time. This is different from all prior work, where a GPU can only participate in one gradient communication at any time. Communication concurrency not only improves the network bandwidth utilization but also leads to higher training throughput and faster training time.

Programming interface. Porting model code to AIACC-Training is straightforward and does not require user involvement. For vanilla *sequential* DNN code written in Tensorflow, Pytorch, MXNet and Caffe, AIACC-Training uses a compiler-based source-to-source translator to automatically convert the user program to AIACC-Training's Perseus API for *distributed training*, eliminating the need for manual code refactoring. As the AIACC-Training API is fully compatible with the Horovod API [9], porting Horvod distributed training programs to AIACC-Training is also simple. In practice, this means just changing one line of the code by replacing the import package from Hrovod to Perseus. This is also automatically handled by AIACC-Training. Porting MXNet's parameter server-based code to AIACC-Training can be realized using the MXNet key value store interface for parameter synchronization.

[4]https://www.alibabacloud.com/help/en/doc-detail/198783.html

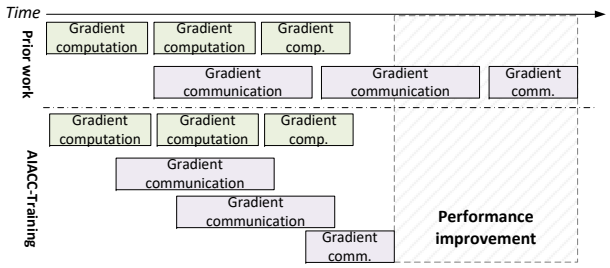


Fig. 5: AIACC-Training exploits asynchronous communication concurrency to improve training throughput.

Main components. As shown in Fig. 4, AIACC-Training has two main components, a communication servicing process built upon the Message Passing Interface (MPI) and a parameter optimizer chosen by the user code or the AIACC-Training runtime. The MPI process runs on the host CPU, and the optimizer runs together with each training worker on a GPU. We have one MPI process for each GPU worker. We use MPI to facilitate inter-node and intra-node communications. We found that using OpenMP or Pthread for inter-process communication with the same host offers little benefit for DDL, but doing so will increase the complexity of code development and maintenance.

Advantages. Compared to existing distributed communication libraries, AIACC-Training offers several advantages that were motivated by real-life use cases. AIACC-Training improves network bandwidth utilization by allowing a GPU worker to participate in multiple all-reduce operations at once using asynchronous, parallel communications. Unlike Horovod that uses a fixed-sized communication window, AIACC-Training implements an adaptive scheme to find, *during runtime*, the optimal granularity for gradient communication and aggregation, further enhancing the training throughput.

Other features and optimizations. As a production library, AIACC-Training also provides fault-tolerance to restart the training process from the last checkpoint upon node failure and elastic deployment by propagating training parameters into newly added computing nodes. It offers debugging support like identifying NaN (not a number) values from individual gradients - a headache for many users during DDL. It implements a new optimizer by combining Adaptive Moment Estimation (Adam) [21] and Stochastic Gradient Descent (SGD) [22]. It uses linear decay to adjust the learning rate rather than the commonly used step decay [23] because we found linear decay works better with the communication optimization and gradient compression implemented in AIACC-Training. Building upon the multi-streamed concurrent communication optimization technique focused in this paper and the new optimizer, we have demonstrated that it is possible to train ResNet50 on ImageNet in 158 seconds using 128 NVIDIA V100 GPUs. The was the state-of-the-art training speed on the DAWNBench league table [24].

V. Gradient Communications in AIACC-Training

As depicted in Fig. 5 and Fig. 6, AIACC-Training decouples gradient computation and communication so that computation

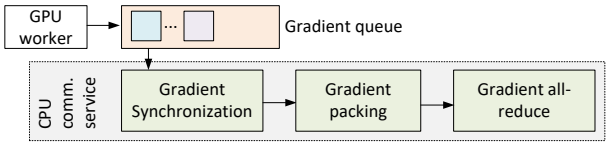


Fig. 6: Overview of the AIACC-Training gradient communication process. Distributed communications and local gradient computation run concurrently in an asynchronous manner.

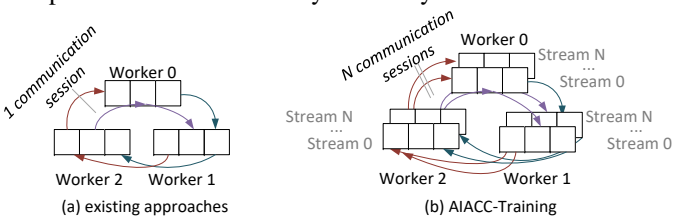


Fig. 7: Unlike prior work that only utilizes one communication link for all-reduce (a), AIACC-Training performs N concurrent all-reduce operations on N communication links (over the same physical link) through N CUDA streams (b).

runs concurrently with distributed data communications. Gradient communication in AIACC-Training consists of multiple stages that iterate over training steps, described as follows:

Gradient synchronization. As gradients can be produced in arbitrary order for independent parameters (e.g., parameters 4 and 5 in Fig. 8) during backward propagation, all training workers need to agree on what gradients to participate in a single all-reduce operation. This is managed by an MPI process that communicates with the GPU worker via a CUDA-MPI aware message queue. Gradient synchronization will be triggered when the size of the locally computed gradients meets the communication granularity - AIACC-Training automatically chooses this parameter during runtime (see Section VI). To this end, the MPI communication process uses a ring all-reduce operation to check if gradient values of a parameter, like like a linear layer's weights (see Section II-A), are ready among all training workers. If a gradient has been produced (i.e., *synchronized*) by all workers, a follow-up all-reduced operation can then be applied to this gradient without needing to wait until other gradients to be computed across the DNN layers. This strategy permits the use of multiple concurrent gradient communications to speedup the training process as shown in Fig. 5. This is detailed in Section V-A.

Gradient packing. A ring all-reduce operation can be performed on a synchronized gradient after all workers have computed their locally corresponding values. Because the tensor size of gradients can vary, and the optimal communication granularity depends on the communication network, the AIACC-Training runtime may choose to split the tensor into multiple units or merge multiple tensors across multiple synchronized gradients to form a suitable all-reduce unit.

Gradient all-reduce. Once an all-reduce unit is ready, it is dispatched to a communication kernel executed by a CUDA stream to perform an all-reduce operation among parallel workers. As highlighted in Fig. 7, unlike prior work, AIACC-

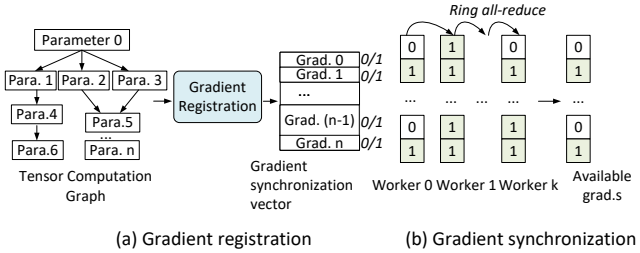


Fig. 8: When loading a model, each training worker registers the model parameters to participate in communication (a) through a gradient synchronization vector. The MPI process then uses a ring all-reduce to agree on the available gradient values for gradient aggregation among training workers (b).

Training enables a worker to participate in more than one all-reduced operation over the same network. This strategy utilizes the hardware parallelism to multiplex the communication resources. To support concurrent all-reduce operations, AIACC-Training manages a communication thread pool, where each thread runs within a CUDA stream. The AIACC-Training runtime automatically dispatches an all-reduce unit to an available thread, which then takes care of gradient communication by creating or participating in a new all-reduce ring. To support multi-streamed communications, AIACC-Training amends NCCL's low-level communication primitives, but such changes are transparent to the user's code.

A. Gradient Registration and Synchronization

Fig. 8 summarizes the gradient registration and synchronization processes of AIACC-Training.

1) Gradient registration: When loading a DNN model, the training worker registers the parameters to participate in all-reduced gradient aggregation. This will generate a $n-element$ gradient synchronization vector (where n is the number of gradients generated during backward propagation) stored in the CPU memory, as shown in Fig. 8a. Each vector element takes a bit-wise value of 0 or 1, where a value of 1 indicates a gradient value is computed locally and ready to be reduced. During gradient registration, parameters are sorted and assigned a unique index in the gradient synchronization vector. Before each backward stage, elements of the gradient synchronization vector are set to zeros.

2) Gradient synchronization: The GPU-based training worker and the CPU-based MPI communication process talk through a gradient queue implemented using CUDA-aware MPI. The CUDA-aware MPI feature provides a virtual single memory space, where the underlying CUDA runtime automatically manages the data transfer between the GPU and the CPU memory space. After a local gradient is computed, a callback function then pushes the gradient tensor (i.e., a multi-dimensional array) into the gradient message queue. This callback function is automatically registered by AIACC-Training through a customizable hook function similar to the callback mechanism supported by Tensorflow and PyTorch.

A gradient push operation will wake up the CPU-based MPI process to update the gradient synchronization vector, setting

the corresponding bit to 1 to indicate that a corresponding local gradient value is ready. Meanwhile, the gradient tensor will be removed from the gradient queue to be stored in a gradient communication bucket. If GPU-directed RDMA is available, the bucket will be allocated in the GPU memory for GPU-directed RDMA. Otherwise, it will be stored in the CPU memory. If the gradient bucket size meets the minimum communication granularity, the MPI communication process then triggers the gradient synchronization process. As illustrated in Fig. 8b, this is achieved by performing a ring all-reduce among MPI communication processes - where an MPI daemon process links to a training worker. To check if a gradient has been computed by all training workers, we apply a `min` reduction operator to each element of the gradient synchronization vector. Since a `min` operator is used, a gradient in the all-reduced, synchronization vector will be marked as 0 (not ready) if it has not been computed by any of the workers. Gradient synchronization is asynchronously performed by another process managed by the underlying collective primitive library (i.e., the MPI process is not blocked on a synchronization operation). Because gradient synchronization and computation are performed on two computing devices (CPU and GPU), and the multi-core CPU is mainly idle during the backward stage, both processes run concurrently, incurring negligible overhead. It also has a low network overhead because we only perform all-reduce on a bit vector.

Unlike AIACC-Training, Horovod and other communication frameworks require having a single master (or root) node to determine what gradients are ready. However, the master node can quickly become a communication bottleneck at large-scale DDL as all workers need to communicate with it. In contrast, AIACC-Training takes a fully distributed approach for gradient synchronization, preventing a single node from becoming the communication bottleneck.

B. Gradient Packing and All-reduce

After gradient synchronization, all training workers agree on what gradients to participate in the follow-up gradient all-reduce process. AIACC-Training then determines how to split or pack tensors of ready gradients to form an optimal all-reduce unit. This communication parameter is automatically determined by AIACC-Training during the warm-up phase and is used by all participating communication threads. Multiple small tensors will be packed to form a large tensor in an all-reduce unit, while a large tensor can be breakdown into multiple all-reduce units. As the size of ready gradients is often larger than the chosen all-reduce size, there are likely to be multiple all-reduce units to be communicated. Furthermore, because ready gradients are packed or sliced according to the gradient id (given during parameter registration), all workers also implicitly agree on gradient communication order.

AIACC-Training utilizes and extends the collective communication primitives (like all-reduce, broadcast, and scatter) of NCCL and Gloo respectively for GPU- and CPU-based communications. Unfortunately, NCCL only supports one communication link, which can utilize up to 10Gbps bandwidth of a TCP/IP network, leading to poor bandwidth utilization in a GPU cloud environment. AIACC-Training addresses this issue by issuing multiply communication links over the network by extending the low-level NCCL implementations.

Algorithm 1: Multi-streamed gradient communication

```
   Input: L: list of synchronized gradients;
          N: number of threads in P
   Data: la: list of available all-reduce units;
          lr: list of received all-reduced units;
          P: communication thread pool
1  P ← initCUDAStreams(N);
2  while !empty(L) do
3      la ← GradientPacking(L);
4      foreach u ∈ la do
5          p ← getAvailThread(P);
6          if p == null then
               // no more free thread
7              break from while loop;
8          end
9          p− > all_reduce(u);
10     end
11 end
12 if all gradients have been communicated then
13     gradients[] ← gradient_unpack(lr);
14     gradient_callback_func();
15 end
```

As described in Algorithm 1, multi-streamed gradient communication is achieved by first creating a thread pool with multiple CUDA stream contexts (line 1). Each CUDA stream corresponds to an underlying communication buffer used for an RDMA or TCP/IP network. The MPI communication process automatically dispatches an all-reduce unit to an available CUDA stream (line 2) which then applies an all-reduce operation to perform gradient aggregation among training workers (line 9). AIACC-Training currently supports two all-reduce algorithms, a ring all-reduce and tree all-reduce. The latter first performs a ring all-reduce operation among GPUs of the same computing node and then uses ring all-reduce to communicate across computing nodes. It is useful when some of the physical network links become congested due to burst communications from other shared cloud users. AIACC-Training automatically determines which all-reduce algorithm to use during the auto-tuning phase without user intervention.

Within a ring, a CUDA stream of training worker p sends data to worker id: $(p+1)\%p$, forming a ring as depicted in Fig. 1. Unlike traditional ring all-reduce, a worker can participate in multiple all-reduce rings as shown in Fig. 8b. We stress that gradient synchronization and communications are asynchronous operations, and the MPI process is not blocked on the operation. This allows gradient synchronization and multi-streamed communication operations to run concurrently on a multi-core CPU (see also Fig. 5).

Once the all-reduce operation has been performed on every gradient, AIACC-Training will unpack and regroup the data back to individual gradient tensors, which are passed to an optimizer (for parameter update) via a call back function. Using multiple CUDA streams (and communication buffers) improves bandwidth utilization by multiplexing the commu-

nication network. By issuing multiple concurrently GPU-CPU memory transfers (because TCP/IP communications go through the CPU), multiple CUDA streams can also hide the GPU-CPU communication overhead.

VI. Auto-tuning Communication Parameters

Hyperparameters like the all-reduce unit size, the number of CUDA streams used and the all-reduce algorithm can have an impact on the communication efficiency. The combination of possible parameter values results in a large optimization space, where the optimal setting depends on the cloud instances, the network topology and bandwidth, and the DNN workload characteristics. AIACC-Training automatically finds the suitable parameters at runtime using an ensemble of search techniques. We use an ensemble approach because it allows us to plug in a new search technique easily. Using a collection of search algorithms also improves the robustness parameter search when the network bandwidth, topology, and DNN workload change. We formulate the parameter search problem as a multi-armed bandit (MAB) problem [13] and use a meta solver to find the best parameters under a predefined number of training iterations during the warm-up phase. Note that the results of the warm-up phase also contribute to the final outcome, so no computation cycle is wasted.

Our current search ensemble considers four established search techniques: grid-search, population based training (PBT) [25], Bayesian optimization [26], and Hyberband [27], but other search techniques can be added. Our meta solver is a *MAB with a sliding window, area under the curve (AUC) credit assignment algorithm*. A similar technique was used in prior work [28]–[30] for compiler optimization. Given a budget of n training iterations and k search techniques ($k = 4$ and $n = 100$ by default in our current implementation), the meta solver allocates the training iterations among search techniques to test their effectiveness. After n iterations, we choose the best performing parameters to use for the remaining training iterations. Like [28], our meta solver aims to maximize term: $\arg\max_t(AUC_t + C\sqrt{\frac{2lg|H|}{H_t}})$, where t is a search technique used for the current iteration, $|H|$ is the length of a sliding history window, H_t is how often the technique has been used in that history window, C (set to 0.2 by default) is a constant controlling the exploration/exploitation trade-off, and AUC_t is the credit assignment term quantifying the performance of the technique in the sliding window. The second term in the formula is the exploration ratio which becomes smaller the more often a technique is used. We compute the AUC curve by looking at the history of a technique. If the technique delivered a new global best, we draw an upward line on the AUC curve. Otherwise, we draw a flat line. We then compute the area size under the AUC curve – the larger the area is, the more efficient a technique is likely to be.

When used in a GPU cloud, AIACC-Training also stores the previously-found best parameter setting for a given DNN computation graph, cloud instance and network topology. It then uses this setting as a starting point for a similar cloud instance deployment to boost the search. To quantify the similarity of a DDL deployment and a previously seen one, we measure the similarity of the DNN computation graph and the network topology. The latter is an undirected graph where graph nodes are GPU instances and edges are the network bandwidth. We use the graph edit distance [31] to measure graph similarities and choose a previously found setting that is most similar to the input user code and cloud instance.

TABLE I: DNN model characteristics

Model	#Param.s	#FLOPs	Model	#Param.s	#FLOPs
VGG-16	138.3M	31G	ResNet-50	25.6M	4G
ResNet-101	29.4M	8G	Transformer	66.5M	145G
BERT-Large	302.2M	232G			

VII. Evaluation Setup

A. Evaluation Platform

We evaluate AIACC-Training on the ecs.gn6e-c12g1.24xlarge instance of Alibaba GPU cloud. Each instance is equipped with 4x 24-core vCPUs (2.5GHz Intel Xeon Platinum 8163 CPU), 736GB of DDR4 RAM, and 8x NVLink-enabled 32GB NVIDIA V100 GPUs. Unless stated otherwise, computing nodes are connected via VPC with a TCP/IP network bandwidth of 30Gbps, but in Section VIII-D, we evaluate on RDMA connections. Each computing node runs Linux kernel v3.10.0, and we use CUDA v11.4. All our experiments run on isolated machines with no other job running at the same time to ensure reproducibility.

B. DNN Workloads

Our main evaluation considers five representative DNN architectures. These include three popular convolutional neural networks (CNNs) for computer vision (CV): VGG-16 [32], ResNet-50 [33], and ResNet-101 [33], and two state-of-the-art architectures for natural language process (NLP): Transformer [34] and BERT-Large [35]. We use the ImageNet dataset for CV models and the Wikitext-en dataset for NLP models. Table I summarizes the model size and computation requirement - the number of floating-point operations (FLOPs) of these models. In Section VIII-D, we also report performance on GPT2-XL (with 1,558M parameters) and a warehouse-scale click to recommendation (CTR) system that supports billion-scale transactions in an Alibaba production environment, although we cannot disclose the specific model structure used by CTR. Unless stated otherwise, we use data parallelism when using multiple GPUs.

C. Competing Methods

We compare AIACC-Training to the latest version of two state-of-the-art distributed gradient communication libraries: Horovod (v0.23) and BytePS (v0.2) [2]. Like AIACC-Training, BytePS supports Tensorflow, PyTorch, and MXNex with a Horovod compatible API. We also compare AIACC-Training with DDL implementation based on the latest PyTorch (V1.10) distributed data-parallel (DDP) API. We refer this scheme to as PyTorch-DDP. Both PyTorch-DDP and Horovod rely on all-reduce for gradient communication, while BytePS uses parameter servers. We omit Caffe in our evaluation because it has been merged to PyTorch. For a

fair comparison, we use the same training hyper-parameters (optimizer, learning rate and batch size) for all DDL methods. We turn off the optimizer optimization offered by AIACC-Training to focus on evaluating gradient communications.

D. Performance Report

Like [2], we observed that training throughput (the number of training samples processed per unit of time) stabilizes after the first 100 iterations for all methods. Thus, we report performance after the first 100 iterations for the subsequent 200 iterations. We run each experimental setup 5 times and report the geometric mean performance for each test case. To provide a fair comparison, we follow the DNN hypermeter setting used by BytePS in [2], with a large training batch size. We stress that smaller batch sizes mean less GPU computation[5] but more communication, where the improvement of AIACC-Training will be more evident. Since the chosen batch size uses almost full GPU memory, the improvement over competing methods is the *lower bound* of AIACC-Training.

VIII. Experimental Results

A. Overall Performance

Fig. 9 and Fig. 10 show the training throughput on PyTorch-based CV and NLP models, respectively, when we vary the number of GPUs (a computing node has 8 GPUs). AIACC-Training gives consistently good results across evaluation settings. It starts exhibiting stronger performance when using more than 8 GPUs with more than one computation node. This is because gradient communications quickly become the performance bottleneck for DDL with multiple computing nodes. BytePS gives poor performance because it requires additional CPU servers to minimize the bottleneck overhead of the parameter servers, which is in line with an independent study [36]. To achieve improved performance for BytePS will incur an extra financial cost for CPU machine subscription. Horovod and PyTorch-DDP also deliver better scalability than BytePS, showing the advantage of all-reduce for DDL in a typical GPU cloud setup over parameter servers. AIACC-Training further improves Horovod and PyTorch-DDP by better utilizing the network bandwidth to reduce the communication overhead, leading to a higher throughput with multiple computing nodes. Such performance advantage is more evident with a large number of GPUs. For example, when using 256 GPUs, AIACC-Training improves Horovod and PyTorch-DDP by up to 1.68x and 2.68x. We anticipate AIACC-Training to have greater advantages with more computing nodes.

We also observe that different DNN models manifest different scaling efficiency. This is not surprising because the model scalability depends on the model's size and communication patterns. The most scalable model is ResNet-50 with 256 GPUs, where AIACC-Training achieves over 95% scaling efficiency. This result is in line with prior studies [1], [2], which show that ResNet-50 has better scalability than other models. Achieving higher scalability for other larger models is challenging. This is because large models are typically more computation-intensive, having more floating-point operations. Computation-intensive models limit the number of CUDA streams that can be executed concurrently for gradient communications. Nonetheless, AIACC-Training gives the highest throughput for all models. As future-generation GPUs are likely to provide more parallel execution units, we expect AIACC-Training will deliver better performance on future high-end GPUs by leveraging the hardware parallelism.

[5]Less GPU computation means that there will be a higher chance for the GPU hardware scheduler to dispatch more CUDA streams to run concurrently on the hardware for gradient communications.

B. Other DL Frameworks

We now apply the unified AIACC-Training library to DNN models written with Tensorflow and MXNet. AIACC-Training automatically converts the sequential model code for DDL. Fig. 11 and Fig. 12 show that AIACC-Training also gives consistently good performance on Tensorflow and MXNet, with similar improvements seen on PyTorch. Once again, AIACC-Training demonstrates greater performance as the number of GPUs (and computing nodes) increases, with a speedup of 3.3x over Horovod when using 256 GPUs. We can also see that the parameter server approach used by MXNet gives a lower throughput compared to the all-reduce used by Tensorflow and PyTorch. The results suggest that AIACC-Training gives portable performance across DL frameworks. As a unified communication framework, AIACC-Training reduces the development and maintenance cost, as the same optimization can be applied to a range of DL frameworks to meet the needs of different GPU cloud users.

C. Other DNN Workloads and Metrics

We also evaluated AIACC-Training in other evaluation scenarios. When applying AIACC-Training to the hand-tuned ResNet-50 of the InsightFace library [37] (with DDL enabled) on face preconization datasets, AIACC-Training improves the hand-tuned DDL code by 3.8x when using 128 GPUs. We also evaluate AIACC-Training on the DAWNBench metrics [24]. The metrics include time and the total cost of public cloud instances to train ResNet-50 to reach a top-5 validation accuracy of 93% or greater on ImageNet. An earlier version of AIACC-Training was top in the DAWNBench league board for both training time and cost. Specifically, AIACC-Training achieved the training goal within 158 seconds using 128 V100 GPUs across 16 computing instances with a training cost of $7.43. AIACC-Training has also been deployed to support a wide range of internal machine learning systems within Alibaba. One such example is a click-to-recommend (CTR) system. On this industry e-commercial workload, AIACC-Training improves the previously hand-tuned Horovod-DDL implementation by 13.4x when using 128 V100 GPUs, allowing the learning system to process 100+ billion entries in 5 hours to quickly update the model. For this workload, Horovod's master node strategy is a bottleneck during gradient synchronization. By adopting a decentralized synchronization scheme, AIACC-Training significantly improves the DDL scalability. These additional use cases confirm the good generalization ability of AIACC-Training.

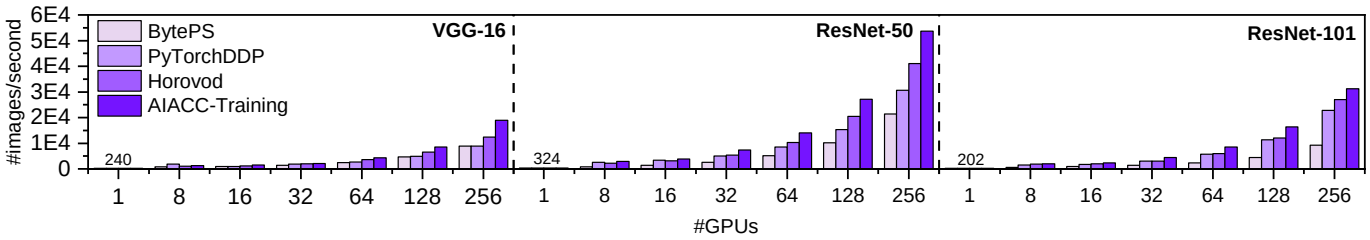


Fig. 9: Performance on PyTorch based CV models.

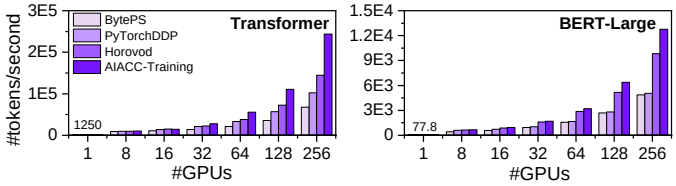


Fig. 10: Performance on PyTorch based NLP models.

D. Further Analysis

Hybrid parallelism. So far, our evaluation has focused on data parallelism for DDL. Fig. 13 shows the performance for applying AIACC-Training to ResNet-50 using a hybrid data and model parallelism. In this experiment, we use the MXNet implementation of ResNet-50 by replacing the MXNet's KVStore interface with AIACC-Training. AIACC-Training consistently improves the MXNet DDL implementation, improving the throughput by 2.8x when using 64 GPUs.

Impact of batch size. Fig. 14 shows the throughput improvement given by AIACC-Training over Horovod as we vary the training batch size using 16 GPUs (2 computing nodes) on BERT-large. Here, a batch contains an average of 128 tokens. Because of the small number of computing nodes used, the results represent a low-bound performance improvement of AIACC-Training. AIACC-Training gives better performance on small batch sizes due to the more frequent gradient communications. We stress that using a large batch size increase the GPU memory pressure and can slow down the model convergence [38]. Therefore, it is common to use a modest batch size for training and fine-tuning, for which AIACC-Training will manifest better advantages.

Performance on RDMA. Fig. 15 shows the performance improvement on 8 RDMA-enabled computing nodes (64 GPUs) over PyTorch-DDP. On the large GPT-2 DNN, AIACC-Training gives a 9.8x speedup over PyTorch-DDP. We also observe a similar performance trend on other deep learning frameworks and when using a different number of GPUs, where AIACC-Training gives around 10% extra improvement on RDMA on top of the improvement seen on TCP/IP networks (Section VIII-A). This experiment confirms that AIACC-Training can effectively utilize the fast RDMA network to deliver scalable performance.

Auto-tuning parameters. As we have discussed in Section VI, AIACC-Training automatically chooses the gradient communication hyperparameters during runtime. We observe that the chosen parameters vary across DNN workloads and GPU instances. In our evaluation, AIACC-Training chooses to use ring all-reduce instead of tree-based all-reduce, but the number of concurrent CUDA streams varies between 2 and 24, whereas AIACC-Training tends to use a larger number of CUDA streams when a higher number of GPUs is available. This is expected because computation per GPU decreases with more GPUs, leaving further room for concurrent gradient communications. Similarly, the chosen gradient communication granularity also changes. The chosen communication granularity is larger for the Transformer-based model over VGG and ResNet because of the larger number of gradients generated by Transformer-based DNNs. By employing auto-tuning techniques, AIACC-Training automatically chooses the right parameter setting without user intervention.

IX. Discussions

Naturally, there is room for improvement and further work. We discuss a few points here.

Collective communications. AIACC-Training builds upon low-level collective communication primitives (all-scatter, all-gather, etc.) for gradient synchronization and communications. Techniques for improving these communication primitives [1], [6] are thus orthogonal to AIACC-Training.

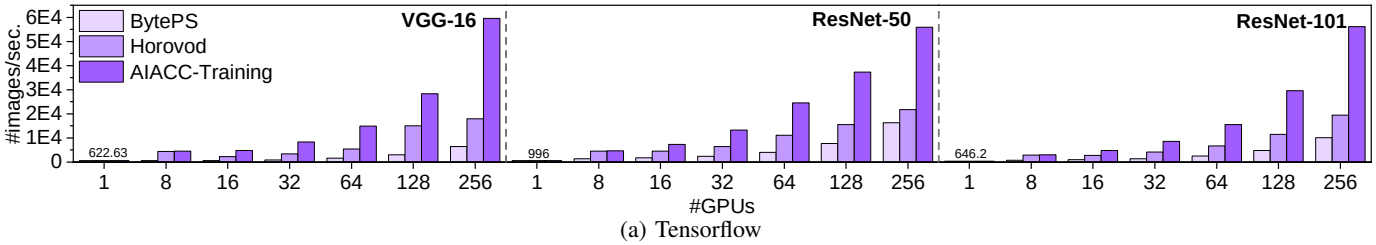
Tensor graph optimizaiton. An ongoing work of AIACC-Training is to exploit compiler optimization to transform the tensor operators to co-optimize computation with communications. By splitting or merging tensor operations, it is possible to better overlap GPU computation with network IO to improve the training throughput further [39].

Utilizing multi-core CPUs. An interesting future direction is to better utilize the multi-core CPU for training. For example, some operations of gradient reduction and parameter updates can be performed on the CPU. Doing so can reduce the GPU memory footprint and utilize the multi-core CPU computation capability. However, care must be taken to make sure the CPU-GPU data transfer does not become a bottleneck.

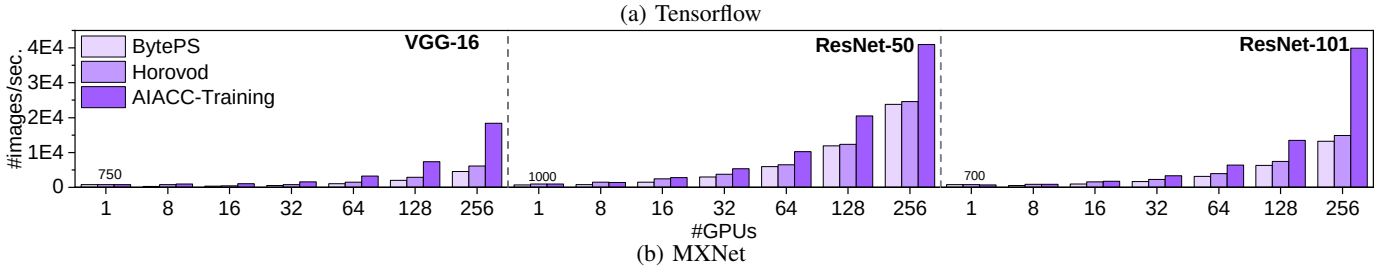
Exploiting DNN characteristics. New DNN workloads exhibit new characteristics. For example, inputs to the graph neural network are sparse matrices [40]. It would be interesting to understand how to exploit the workload characteristics to improve the training speed. For example, can we have a new sparse matrix storage format designed for DDL?

X. Related Work

Efforts have been made to accelerate DDL communication. These include works on overlapping the computation and communication through tensor partitioning and computation

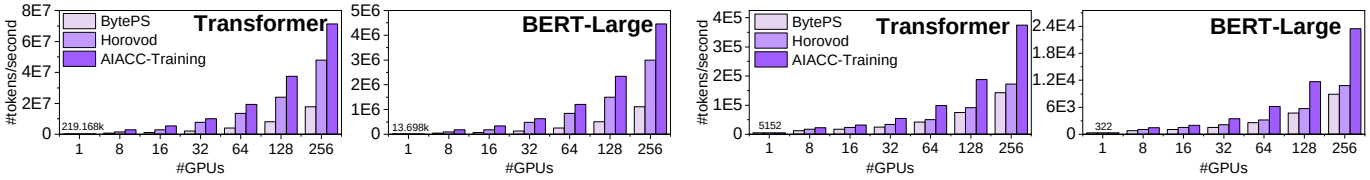


(a) Tensorflow



(b) MXNet

Fig. 11: Performance on Tensorflow (a) and MXNet (b) for CV models.



(a) Tensorflow

(b) MXNet

Fig. 12: Performance on Tensorflow (a) and MXNet (b) for NLP models.

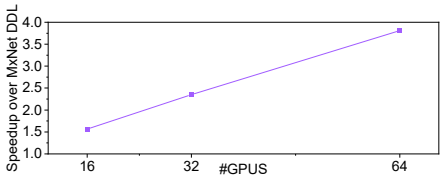


Fig. 13: Throughput improvement over MXNet (DDL) for ResNet-50 with data and model parallelisms.

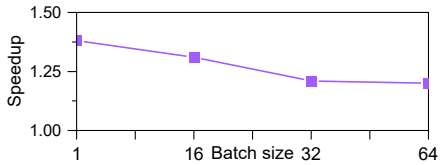


Fig. 14: Speedup over Horovod on BERT-Large with different batch sizes on 16 GPUs (two computing nodes).

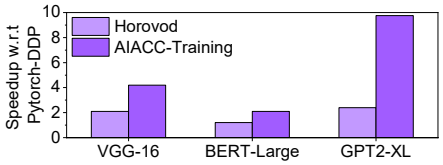


Fig. 15: Throughput improvement over Pytorch-DDP on 64 GPUs with RDMA connections.

scheduling [39], [41], [42]. These approaches are complementary to AIACC-Training. Other techniques apply gradient compression to reduce the amount of data to be communicated among training workers [7], [8]. AIACC-Training adopts a similar idea by using half-precision representation to accelerate gradient transmission, but this is not the focus of this paper.

Our work is closely related to works on optimizing communication frameworks for distributed training [43], [44]. Horovod [9] is a unified communication framework for DDL, supporting TensorFlow, PyTorch, and MXNet. AIACC-Training is compatible with the Horovod API, making it easy to port existing Horvod code to AIACC-Training. However, Horovod fails to capitalize on the abundant network bandwidth and GPU parallelism in a distributed cloud environment. AIACC-Training advances Horovod by developing a fully decentralized gradient synchronization and communication scheme by leveraging multiple CUDA streams across multiple gradient communications, leading to significantly higher throughput and better scalability. Our work is also related to prior studies on optimizing all-reduce operations by exploiting the network topology [5]. AIACC-Training advances these prior methods by leveraging multiple CUDA streams and autotuning to accelerate gradient communications.

BytePS leverages additional CPU servers to improve parameter-server-based DDL [2]. Blink optimizes communications within a single node by carefully utilizing NVLinks and PCIe [6]. Unlike AIACC-Training, BytePS and Blink do not leverage multiple communication streams for all-reduce. Nonetheless, their techniques can be leveraged by AIACC-Training to better utilize the CPU computing resources.

DeepSpeed [45] supports the training of large-scale DNNs using data and model parallelism. Unlike AIACC-Training, DeepSpeed requires heavy user involvement to change the model training pipeline and implement a standard ring-based all-reduced operation. In contrast, AIACC-Training requires no change to the user code and offers a highly optimized all-reduced algorithm. We are working on extending our code translator to use DeepSpeed API for DDL automatically.

There is also a growing interest in designing specialized hardware to accelerate DNN training. Examples of such neural network accelerators like TPU [16], Habana [46] and Alibaba

Hanguang NPU [17], and programmable network switches [47], [48]. While AIACC-Training primarily targets NVIDIA GPUs for GPU clouds, the techniques can be applied to specialized accelerators. For example, AIACC-Training has been ported to the Alibaba Hanguang NPU.

XI. Conclusion

We have presented AIACC-Training, a unified communication library for distributed deep learning training. AIACC-Training provides a single, unified communication interface for mainstreamed deep learning programming frameworks. It aims to improve network bandwidth utilization by exploiting GPU hardware parallelism. It achieves this by decoupling gradient computation from communications and carefully partitioning the gradients to be sent through multi-streamed, concurrent communications. As a departure from prior work, AIACC-Training implements a fully decentralized approach for gradient communication. It employs auto-tuning to dynamically determine the suitable communication parameters to adapt to changes in runtime deployment.

Our experiments on public and production DNN workloads show that AIACC-Training achieves better scaling efficiency than existing distributed training frameworks. AIACC-Training has been deployed and extensively used by Alibaba's internal and external users. In the Alibaba public GPU cloud, there are currently more than 3000 GPUs executing a diverse set of AIACC-Training optimized models *at any time*, and we expect this number to continuously grow as more advanced optimizations are introduced to AIACC-Training.

Acknowledgment

This work was supported in part by an Alibaba Innovative Research Programme between the University of Leeds and Alibaba. For any correspondence, please get in touch with Zheng Wang (Email: z.wang5@leeds.ac.uk).

References

[1] J. Fei *et al.*, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *SIGCOMM*, 2021.
[2] Y. Jiang *et al.*, "A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters," in *OSDI*, 2020.
[3] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, 2019.
[4] Z. Zhang *et al.*, "Is network the bottleneck of distributed training?" in *NetAI*, 2020.
[5] M. Cho *et al.*, "Blueconnect: Novel hierarchical all-reduce on multi-tired network for deep learning," in *MLSys*, 2019.
[6] G. Wang *et al.*, "Blink: Fast and generic collectives for distributed ml," *Proceedings of Machine Learning and Systems*, 2020.
[7] Y. Lin *et al.*, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *ICLR*, 2018.
[8] C.-Y. Chen *et al.*, "Adacomp: Adaptive residual gradient compression for data-parallel distributed training," in *AAAI*, 2018.
[9] A. Sergeev *et al.*, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv*, 2018.
[10] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.
[11] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NIPS*, 2019.
[12] T. Chen *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv*, 2015.
[13] A. Fialho *et al.*, "Analyzing bandit-based adaptive operator selection mechanisms," *Annals of Mathematics and Artificial Intelligence*, 2010.
[14] M. Li *et al.*, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, 2013.
[15] A. Gibiansky, 2017. [Online]. Available: https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/
[16] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.
[17] Y. Jiao *et al.*, "Hanguang 800 npu–the ultimate ai inference solution for data centers," in *HCS*, 2020.
[18] "Nvlink and nvswitch the building blocks of advanced multi-gpu communication." [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/
[19] A. Li *et al.*, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE TPDS*, 2019.
[20] T. Shanley, *InfiniBand network architecture*. Addison-Wesley Professional, 2003.
[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv*, 2014.
[22] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*.
[23] Y. Lin *et al.*, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv*, 2017.
[24] C. Coleman *et al.*, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, 2017.
[25] M. Jaderberg *et al.*, "Population based training of neural networks," *arXiv*, 2017.
[26] H. Ha *et al.*, "Bayesian optimization with unknown search space," *NIPS*, 2019.
[27] L. Li *et al.*, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, 2017.
[28] J. Ansel *et al.*, "Opentuner: An extensible framework for program autotuning," in *PACT*, 2014.
[29] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, 2018.
[30] H. Wang *et al.*, "Automating reinforcement learning architecture design for code optimization," in *CC*, 2022.
[31] X. Gao *et al.*, "A survey of graph edit distance," *Pattern Analysis and applications*, 2010.
[32] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv*, 2014.
[33] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[34] A. Vaswani *et al.*, "Attention is all you need," in *NIPS*, 2017.
[35] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv*, 2018.
[36] S. Gan *et al.*, "Bagua: Scaling up distributed learning with system relaxations," *arXiv*, 2021.
[37] "Insightface: an open source 2d&3d deep face analysis library." [Online]. Available: https://insightface.ai/
[38] "Effect of batch size on training dynamics." [Online]. Available: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e
[39] A. Jayarajan *et al.*, "Priority-based parameter propagation for distributed dnn training," *arXiv*, 2019.
[40] S. Qiu *et al.*, "Optimizing sparse matrix multiplications for graph neural networks," 2021.
[41] S. H. Hashemi *et al.*, "Tictac: Accelerating distributed deep learning with communication scheduling," *SysML*, 2019.
[42] Y. Peng *et al.*, "A generic communication scheduler for distributed dnn training acceleration," in *SOSP*, 2019.
[43] Y. Bao *et al.*, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *INFOCOM*, 2020.
[44] S. Shi *et al.*, "Mg-wfbp: Merging gradients wisely for efficient communication in distributed deep learning," *TPDS*, 2021.
[45] J. Rasley *et al.*, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020.
[46] "Habana homepage." [Online]. Available: https://habana.ai/
[47] M. Liu *et al.*, "E3: Energy-efficient microservices on smartnic-accelerated servers," in *ATC*, 2019.
[48] B. Klenk *et al.*, "An in-network architecture for accelerating shared-memory multiprocessor collectives," in *ISCA*, 2020.