TOWARDS SCALABLE DISTRIBUTED TRAINING OF DEEP LEARNING ON PUBLIC CLOUD CLUSTERS

Shaohuai Shi [*1] **Xianhao Zhou** [*2] **Shutao Song** [*2] **Xingyao Wang** [3] **Zilin Zhu** [2] **Xue Huang** [2] **Xinan Jiang** [2] **Feihu Zhou** [2] **Zhenyu Guo** [2] **Liqiang Xie** [2] **Rui Lan** [2] **Xianbin Ouyang** [2] **Yan Zhang** [2] **Jieqian Wei** [2] **Jing Gong** [2] **Weiliang Lin** [2] **Ping Gao** [2] **Peng Meng** [2] **Xiaomin Xu** [2] **Chenyang Guo** [2] **Bo Yang** [2] **Zhibo Chen** [2] **Yongjian Wu** [2] **Xiaowen Chu** [4]

ABSTRACT

Distributed training techniques have been widely deployed in large-scale deep models training on dense-GPU clusters. However, on public cloud clusters, due to the moderate inter-connection bandwidth between instances, traditional state-of-the-art distributed training systems cannot scale well in training large-scale models. In this paper, we propose a new computing and communication efficient top-k sparsification communication library for distributed training. To further improve the system scalability, we optimize I/O by proposing a simple yet efficient multi-level data caching mechanism and optimize the update operation by introducing a novel parallel tensor operator. Experimental results on a 16-node Tencent Cloud cluster (each node with 8 Nvidia Tesla V100 GPUs) show that our system achieves 25%-40% faster than existing state-of-the-art systems on CNNs and Transformer. We finally break the record on DAWNBench on training ResNet-50 to 93% top-5 accuracy on ImageNet.

1 INTRODUCTION

Due to the increase of deep learning (DL) models and data sets, training deep neural networks (DNNs) using stochastic gradient descent (SGD) algorithms, which requires to iteratively update the model parameters to converge, is a compute-intensive process and would be very time-consuming. For example, training a BERT model on a single TPU takes more than 1.5 months (Devlin et al., 2019). Distributed training techniques are the common practice to accelerate the training by exploiting multiple processors in a distributed system collaborating on training the model (Dean et al., 2012; Goyal et al., 2017; You et al., 2018).

In the large-scale training with distributed clusters, synchronous SGD with data parallelism is the main training algorithm that has been widely used in industry (e.g., MLPerf) and academia (Goyal et al., 2017; Jia et al., 2018; Mikami et al., 2018; You et al., 2020). For a P-worker distributed system, the P workers collaborate on training the model $\boldsymbol{w}_t$ to minimize the objective function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ with the following update formula

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \sum_{p=1}^{P} \boldsymbol{g}_t^p, \tag{1}$$

where $\boldsymbol{g}_t^p = \nabla f(\boldsymbol{w}_t, X_t^p)$ is the stochastic gradient at worker p with sampled data X_t^p, and η_t is the learning rate. It generally takes tens to hundreds of epochs to converge to a good solution, and one epoch indicates traversing the whole data samples of the data set once. It is of importance to fully exploit the overall computing power to reduce the iteration time to accelerate the training process. According to Eq. (1), each iteration requires the gradient aggregation of distributed workers, which introduces the data communication between GPUs. The gradient aggregation can be implemented with parameter servers (Li et al., 2014) or an All-Reduce operation (Baidu, 2017; Awan et al., 2017), among which the All-Reduce operation is more widely used in large-scale training (Goyal et al., 2017; You et al., 2018; Lin et al., 2018). Yet, the gradient aggregation generally introduces high communication overheads compared to the GPU computing time.

On one hand, there exists much work (Jouppi et al., 2017; Jia et al., 2018; Wang et al., 2018; Mikami et al., 2018; Ueno & Yokota, 2019; Cho et al., 2019; Wang et al., 2020; Luo et al., 2020; Chu et al., 2020) in providing performance optimizations for the All-Reduce collective for different environments, but the existing state-of-the-art training systems still scale badly on public cloud clusters that are with fast

[*]Equal contribution [1]The Hong Kong University of Science and Technology, Hong Kong, China. [2]Tencent Ltd., Shenzhen, China. [3]University of Michigan, Ann Arbor, Michigan, USA, work done while at Tencent. [4]Hong Kong Baptist University, Hong Kong, China. Correspondence to: Shaohuai Shi <shaohuais@cse.ust.hk>, Xianhao Zhou <jathonzhou@tencent.com>, Shutao Song <sampsonsong@tencent.com>.

Proceedings of the 4[th] MLSys Conference, San Jose, CA, USA, 2021.

Table 1. 8 V100 GPUs computing instances on clouds.

Cloud	Instance	Memory (GiB)	Storage Type	Network (Gbps)
AWS	p3.16xlarge[a]	488	EBS	25
Aliyun	c10g1.20xlarge[b]	336	OSS	32
Tencent	18XLARGE320[c]	320	CFS	25

[a] https://amzn.to/33GQ32w; [b] https://bit.ly/3iRQITn; [c] https://bit.ly/3nyVzvV.

interconnects (e.g., NVLink) within nodes and slow interconnects (e.g., Ethernet) between nodes. On the other hand, recently there are many studies on extensive gradient compression techniques that can significantly reduce the communication traffic with very slight loss of accuracy (Alistarh et al., 2017; Lin et al., 2018; Shi et al., 2019d; Karimireddy et al., 2019; Renggli et al., 2019). The top-k sparsification algorithm (Lin et al., 2018) with good convergence guarantees (Stich et al., 2018; Alistarh et al., 2018) is one of the aggressive algorithms that can only send a small proportion of messages to others with little impact on the convergence.

However, it is non-trivial to achieve real performance gain with top-k sparsification compared to the All-Reduce counterpart due to two main reasons: 1) the top-k selection is very inefficient on GPUs, and 2) the sparsified communication generally requires an All-Gather operation (Renggli et al., 2019) which achieves very low performance on public GPU clusters. Furthermore, I/O with networked file systems (NFS) should also be carefully designed to achieve higher throughput on large-scale training due to the extensive data access at every training iterations. Table 1 presents some popular public cloud instances which are with moderate storage and network connections on high-end computing servers. An experiment of training ResNet-50 with the ImageNet data set using TensorFlow and Horovod with highly optimized configurations shows that 128 Nvidia V100 GPUs in Tencent Cloud (details in §5) only achieve about $40\times$ speedup over a single V100 GPU, which results in a very low scaling efficiency of 31%. We will study the training performance of existing state-of-the-art systems in §2.

To this end, in this paper, we design an efficient communication library with top-k sparsification, in which we propose 1) a novel approximate top-k operator that is friendly to the GPU architecture and 2) a hierarchical top-k communication to better utilize the connection bandwidth resources. To further improve the system scalability, we propose 1) a simple yet effective mechanism for I/O reading with multi-level data caching and 2) parallel gradient post-processing for learning rates calculation. The technical contributions of this paper are summarized as follows:

- We propose an efficient approximate top-k gradient sparsification algorithm on GPUs to compress the communication data with very slight computation overheads.
- We present a novel hierarchical communication algorithm to aggregate sparsified gradients to better utilize the bandwidth on GPU clusters that are with fast interconnects within nodes and slow interconnects between nodes.
- We design an efficient distributed training system for DNNs atop TensorFlow and Horovod, in which we propose a multi-level data caching mechanism and a generic parallel tensor operator to further improve the system scalability.
- We perform distributed training experiments with two types of models, CNNs and Transformer, to demonstrate the effectiveness of the proposed techniques on a Tencent Cloud cluster with 16 nodes connected with 25 Gbps Ethernet (each node has 8 Nvidia Tesla V100-32GB GPUs with NVLink). Experimental results show that our system achieves $25\% - 40\%$ faster than existing state-of-the-art systems.
- We finally demonstrate a case study on the DAWNBench[1] leader-board to achieve the fastest training time on ResNet-50 even with a slower inter-node connection than existing work.

2 Background and Motivation

2.1 Synchronous SGD with Data Parallelism

Synchronous SGD with data parallelism iteratively updates the model parameters with Eq. (1). The training process at each iteration can be decoupled into several steps. 1) Each worker loads a mini-batch of training data from the file system, and the data is then pre-processed (e.g., decoding and argumentation) as the input for the deep model. 2) Each GPU performs feed-forward and backpropagation to computes the local gradients. 3) The local gradients are then aggregated through an All-Reduce operation such that all GPUs have consistent global gradients. 4) The global gradients are then used to compute the model updates along with the learning rate, e.g., layer-wise adaptive rate scaling (LARS) (You et al., 2018). Let b and B denote the local batch size and the global batch size of one training iteration respectively. For a P-worker cluster, $B = b \times P$. Assume that the throughput of the system is T, the number of epochs to train a model is E, and the total number of samples of the data set is N, we can represent the budget of training a model by $\frac{N \times E}{T}$. Therefore, given a P-worker cluster, we should try to reduce the iteration time t_{iter} to increase the

[1] https://dawn.cs.stanford.edu/benchmark/ImageNet/train.html

system throughput $T = \frac{b \times P}{t_{iter}}$, where b is generally set to maximally occupy the GPU memory.

2.2 Existing Problems on Public Clouds

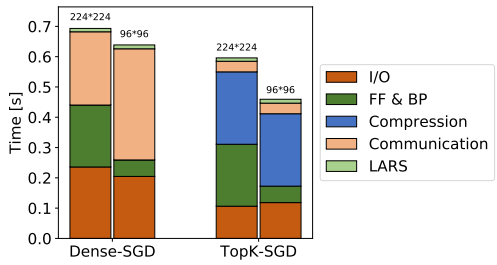


Figure 1. Time breakdown of one iteration with existing training schemes. FF&BP indicates the feed-forward and backpropagation computations. The numbers on the top of the bars indicate the resolution of input images.

To demonstrate the training efficiency of existing state-of-the-art systems on public clouds, we choose the highest configuration on Tencent Cloud with a 16-node GPU cluster connected with 25Gbps Ethernet (25GbE), where each node has 8 Nvidia V100 GPUs connected with NVLink, to measure the training performance of ResNet-50 (He et al., 2016) on the ImageNet (Deng et al., 2009) data set. As for large-batch training, layer-wise adaptive rate scaling (LARS) (You et al., 2018) or LAMB (You et al., 2020) is required to preserve the model generalization ability, so we include the LARS computation in the training. We use Dense-SGD to denote the training scheme of synchronous SGD with the full dense gradients for aggregation. The results are shown in Fig. 1. Note that the feed-forward and backpropagation computing tasks, the gradient communication tasks, and the LARS computing tasks may be executed in parallel if possible (e.g., wait-free backpropagation (Zhang et al., 2017; Awan et al., 2017) and tensor fusion (Shi et al., 2019b; 2020)), so the time breakdown shown in Fig. 1 is the elapsed time that cannot be overlapped by tasks pipelining.

From Fig. 1, we can observe that the I/O time and communication time occupy a large portion of the overall iteration time. For the gradient communication, gradient compression, such as top-k sparsification (TopK-SGD) (Lin et al., 2018; Renggli et al., 2019), can reduce the communication traffic with little impact on the model accuracy. However, there exist two main problems in TopK-SGD, 1) it requires exact top-k selections on GPUs, which could be very slow with the naive implementation, and 2) the top-k elements in each GPUs might have different indices in the original dense gradients so that we need to use the All-Gather collective instead of All-Reduce to aggregate the gradients (Renggli et al., 2019). As shown in Fig. 1, TopK-SGD significantly reduces the communication time, but it introduces an extra top-k compression overhead, which is around 0.239 seconds, while the total of feed-forward and backpropagation time is only 0.204 seconds. On the cases with the small resolution input 96×96, feed-forward and backpropagation time is very small so that the LARS computing time is also relatively significant compared with the feed-forward and backpropagation time.

In summary, it requires careful design to optimize gradient communication on public cloud clusters, and I/O and LARS overheads should also be optimized to improve the overall system throughput.

3 CommLib: An Efficient Gradient Communication Library

As we demonstrated in Section 2.2, the communication cost is significant on the 16-node GPU cloud cluster. We propose a novel sparse gradient communication scheme, which contains two parts: 1) an approximate top-k operator that is friendly to the GPU architecture, and 2) a hierarchical top-k communication algorithm that can better utilize the bandwidth resources.

3.1 MSTopK: an Approximate Top-k Operator

The top-k sparsification requires top-k operations on the gradient tensors which are stored on GPUs, which can be formally defined as follows (Lin et al., 2018; Alistarh et al., 2018; Stich et al., 2018; Shi et al., 2019d). For any input vectors $\boldsymbol{x} \in \mathbb{R}^d$, the top-k operator $\text{TopK}(\boldsymbol{x}, k) \in \mathbb{R}^d$ whose i^{th} element is

$$\text{TopK}(\boldsymbol{x}, k)^{(i)} = \begin{cases} \boldsymbol{x}^{(i)}, & \text{if } |\boldsymbol{x}^{(i)}| > thres \\ 0, & \text{otherwise} \end{cases}, \quad (2)$$

where $\boldsymbol{x}^{(i)}$ is the i^{th} element of $\boldsymbol{x}$, and $thres$ is the k^{th} largest value of $|\boldsymbol{x}|$. Due to the irregular access of GPU memory in the top-k selection, it is non-trivial to implement an efficient top-k selection algorithm on GPUs (Shanbhag et al., 2018; Lin et al., 2018). To this end, we design an approximate top-k selection algorithm with multiple samplings, named MSTopK, which is friendly to many-core processors. The key idea of MSTopK is to use a binary search to find thresholds that are close to the exact threshold (say $thres$).

Assume that we want to select k elements from the input vector $\boldsymbol{x} \in \mathbb{R}^d$. We first use the average value ($\overline{a}$) of the absolute values of $\boldsymbol{x}$ (say $\boldsymbol{a} = |\boldsymbol{x}|$) as the threshold ($thres1$) to select the elements (say $\boldsymbol{\kappa}$) whose values are not smaller than $\overline{a}$. If the dimension of $\boldsymbol{\kappa}$ is smaller (or larger) than k, then we half (or double) $thres1$ as $thres2$. If both $thres1$ and $thres2$ are smaller (or larger) than $thres$, we repeat the above search. After several trials, $thres$ should be located between $thres1$ and $thres2$ (as shown in Fig. 2), then

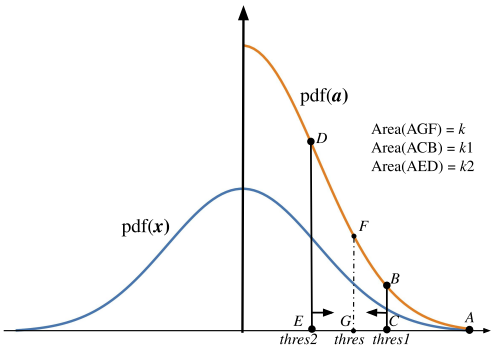


Figure 2. Illustration of the threshold search. Note that the exact threshold $thres$ is unknown, and we use the number of selected elements (e.g., $k1$ and $k2$) with each time's threshold to determine whether the chosen threshold is larger or smaller than the exact threshold. Every sampling of selection, we move $thres1$ and $thres2$ to be close to $thres$.

we can further narrow down the thresholds with the same pattern of search as previous. We set a fixed number of searches (say N), and finally we select k elements using the chosen two thresholds. The pseudo-code of MSTopK is shown in Algorithm 1. Note that there are no expensive memory access operations (e.g., sort) in Algorithm 1, so it would be efficient on GPUs.

3.2 Hierarchical Top-k Communication

In top-k sparsification, the number of elements for worker p to be transmitted becomes $2k$, which contains the vector of the selected k values, $\boldsymbol{v}^p$, and their corresponding indices, $\boldsymbol{i}^p$. $k = \rho \times d$, where $0 < \rho < 1$ is called the density. Due to the irregular indices $\boldsymbol{i}^p$ at different workers, the values $\boldsymbol{v}^p$ cannot be aggregated through the All-Reduce collective. The efficient way is to use two All-Gather operations to aggregate values and indices respectively (Renggli et al., 2019), and then the values are accumulated to local gradients with the corresponding indices. However, each All-Gather operation for P workers has a time complexity of

$$\alpha_{inter} \log P + 4(P-1)\beta_{inter} k, \tag{3}$$

where α_{inter} is the startup time of transmitting a message between two nodes, β_{inter} is the transfer time per byte, and each element is represented by a 32-bit floating-point number (FP32). Note that β_{inter} is the transmission speed between GPUs that are located in different nodes, which would be much slower than the intra-node transmission speed (e.g., NVLink). Therefore, directly using the All-Gather collective on a large-scale cluster connected with low-bandwidth and high-latency networks is very inefficient. To this end, we propose a hierarchical top-k communication (HiTopKComm) algorithm, which can better utilize both bandwidth resources of intra-node and inter-node connections.

Algorithm 1 MSTopK

Input: $\boldsymbol{x} \in \mathbb{R}^d, k, N$

$\boldsymbol{a} = \text{abs}(\boldsymbol{x})$;
$\overline{a} = \text{mean}(\boldsymbol{a})$;
$u = \max(\boldsymbol{a})$;
$l = 0; r = 1$;
$k1 = 0; k2 = \text{len}(\boldsymbol{x})$;
$thres1 = 0; thres2 = 0$;
for $i = 1 \rightarrow N$ **do**
  $ratio = l + (r - l)/2$;
  $thres = \overline{a} + ratio * (u - \overline{a})$;
  $nnz = \text{count\_nonzero}(\boldsymbol{a} \geq thres)$;
  if $nnz \leq k$ **then**
    $r = ratio$;
    if $nnz > k1$ **then**
      $k1 = nnz$;
      $thres1 = thres$;
    end if
  else if $nnz > k$ **then**
    $l = ratio$;
    if $nnz < k2$ **then**
      $k2 = nnz$;
      $thres2 = thres$;
    end if
  end if
end for
$\boldsymbol{\iota}1 = \text{nonzero\_indices}(\boldsymbol{a} \geq thres1)$;
$\boldsymbol{\iota}2 = \text{nonzero\_indices}((\boldsymbol{a} < thres1) \text{ and } (\boldsymbol{a} \geq thres2))$;
$rand = \text{random}(0, \text{len}(\boldsymbol{\iota}2) - (k - k1) + 1)$;
$\boldsymbol{\iota} = \text{concat}(\boldsymbol{\iota}1, \boldsymbol{\iota}2[rand : rand + k - k1]$;
$\boldsymbol{\kappa} = \boldsymbol{x}[\boldsymbol{\iota}]$;
Return $\boldsymbol{\kappa}, \boldsymbol{\iota}$;

Assume that the cluster has m nodes and each node has n GPUs, we use $\boldsymbol{g}_{i,j} \in \mathbb{R}^d$ to denote the local gradients at the i^{th} node's j^{th} GPU, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Our HiTopKComm algorithm contains four steps as shown in Fig. 3. 1) All GPU nodes perform an intra-node Reduce-Scatter operation with $\boldsymbol{g}_{i,j}$ in parallel so that each GPU contains a $\frac{1}{n}$ summation of the d elements, after which GPU j at node i has the updated gradients $\boldsymbol{g}_{i,j}^{[j]} \in \mathbb{R}^{d/n}$ and

$$\boldsymbol{g}_{i,j}^{[j]} = \boldsymbol{g}_{i,j}^{[(j-1)d/n, jd/n]} = \sum_{q=1}^{n} \boldsymbol{g}_{i,q}^{[(j-1)d/n, jd/n]}. \tag{4}$$

2) Each GPU performs the top-k selection

$$\boldsymbol{\kappa}_{i,j}, \boldsymbol{\iota}_{i,j} = \text{MSTopK}(\boldsymbol{g}_{i,j}^{[j]}, \rho \times d/n) \tag{5}$$

using Algorithm 1, which indicates that the MSTopK operation has n times smaller dimensions of input data and the selected elements than selecting top-k elements from the original gradients $\boldsymbol{g}_{i,j}$. 3) Invoking n communication streams for inter-node communications. That is, for the j^{th} communication stream, the j^{th} GPUs in all nodes perform an All-Gather operation with their $\boldsymbol{\kappa}_{i,j}$ and $\boldsymbol{\iota}_{i,j}$. As

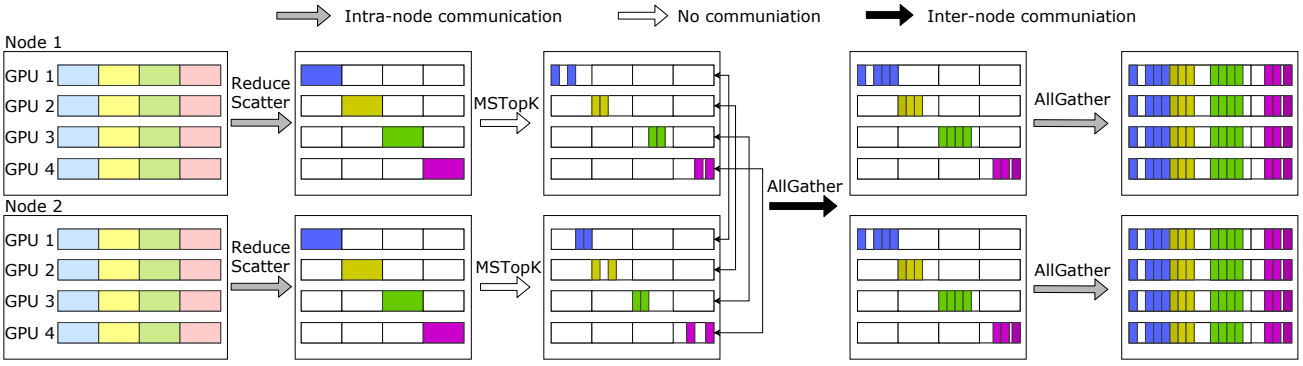


Figure 3. An example of hierarchical top-k communication with two nodes and each node has 4 GPUs.

gradients from different GPUs in different nodes may have different indices, the gathered gradients should be accumulated with their corresponding indices, which results in a maximum of $\frac{\rho d}{n}m$ accumulated elements on each GPU. Formally

$$\boldsymbol{g}_{i,j}^{[j]} = \sum_{p=1}^{m} \text{TopK}(\boldsymbol{g}_{i,j}^{[j]}, \rho d/n). \tag{6}$$

4) All GPU nodes perform an intra-node All-Gather operation with $\boldsymbol{g}_{i,j}^{[j]}$ to construct $\boldsymbol{g}_{i,j}$. The pseudo-code of the HiTopKComm algorithm is shown in Algorithm 2.

Algorithm 2 HiTopKComm

Input: $\boldsymbol{g}_{i,j} \in \mathbb{R}^d, \rho, m, n$

Initiate $\tilde{\boldsymbol{g}}_{i,j} = [0] \in \mathbb{R}^d$;
for $i \in [m]$ in parallel **do**
  $\boldsymbol{g}_{i,j}$=Reduce-Scatter($\boldsymbol{g}_{i,j}$);
end for
$\tilde{k} = \rho \times d/n$;
for $i \in [m], j \in [n]$ in parallel **do**
  $\boldsymbol{\kappa}_{i,j}, \boldsymbol{\iota}_{i,j} = \text{MSTopK}(\boldsymbol{g}_{i,j}^{[j]}, \tilde{k})$;
end for
Initiate $\tilde{\boldsymbol{\kappa}}_{i,j} = [0] \in \mathbb{R}^{m\tilde{k}}$;
Initiate $\tilde{\boldsymbol{\iota}}_{i,j} = [0] \in \mathbb{N}^{m\tilde{k}}$;
for $j \in [n]$ in parallel **do**
  $\tilde{\boldsymbol{\kappa}}_{i,j}$=All-Gather($\boldsymbol{\kappa}_{i,j}$);
  $\tilde{\boldsymbol{\iota}}_{i,j}$=All-Gather($\boldsymbol{\iota}_{i,j}$);
end for
for $i \in [m], j \in [n]$ in parallel **do**
  for $p = 1 \rightarrow m$ **do**
    $\boldsymbol{\kappa} = \tilde{\boldsymbol{\kappa}}_{i,j}^{[p]}$; $\boldsymbol{\iota} = \tilde{\boldsymbol{\iota}}_{i,j}^{[p]}$;
    $\tilde{\boldsymbol{g}}_{i,j}[\boldsymbol{\iota}] += \boldsymbol{\kappa}$;
  end for
end for
for $i \in [m]$ in parallel **do**
  $\tilde{\boldsymbol{g}}_{i,j}$=All-Gather($\tilde{\boldsymbol{g}}_{i,j}$);
end for
Return $\tilde{\boldsymbol{g}}$;

Time complexity: In HiTopKComm, the first step Reduce-Scatter is a ring-based algorithm, which takes a time complexity of

$$t_1^{HiTopKComm} = (n-1)\alpha_{intra} + \frac{4(n-1)d}{n}\beta_{intra}, \tag{7}$$

where α_{intra} and β_{intra} are the startup time and transfer time per byte two GPUs in a single node, respectively. Each element is represented by FP32. The second step is MSTopK with GPU computation, whose time complexity is proportional to the dimension of the input data, that is

$$t_2^{HiTopKComm} \propto \mathcal{O}(\frac{d}{n}). \tag{8}$$

The third step is the inter-node communication with an All-Gather operation, which has a time complexity of

$$t_3^{HiTopKComm} = \alpha_{inter}\log m + 4(m-1)\frac{\rho d}{n}\beta_{inter}. \tag{9}$$

Finally, the last step is an intra-node All-Gather operation with the time complexity of

$$t_4^{HiTopKComm} = \alpha_{intra}\log n + 4(n-1)\frac{\rho dm}{n}\beta_{intra}, \tag{10}$$

in which we assume the indices of the third step are all different so that the number of elements for All-Gather on each GPU in the last step is $\frac{\rho dm}{n}$. Since the inter-node communication is much slower than the intra-node communication, the main time-consuming part is the All-Gather operation in the third step.

4 System Overview

We build our training system which integrates CommLib atop the widely used deep learning frameworks TensorFlow and Horovod as shown in Fig. 4. Besides CommLib, we propose two novel components to further improve the system scalability: 1) Data caching (DataCache), which provides automatically multi-level caching for efficient data reading, 2) Parallel tensor operator (PTO), which enables the computation on high dimensional tensors or a list of tensors to be computed on multiple GPUs in parallel.

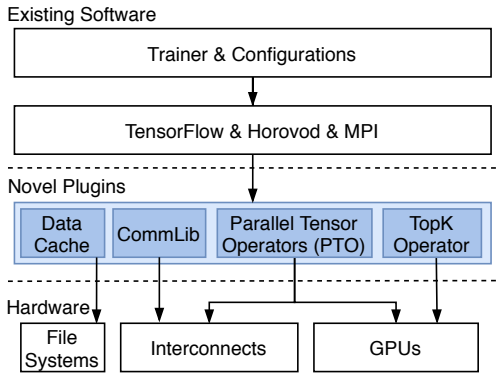


Figure 4. System overview with new proposed optimized plugins.

4.1 DataCache: Caching for Efficient Data Reading

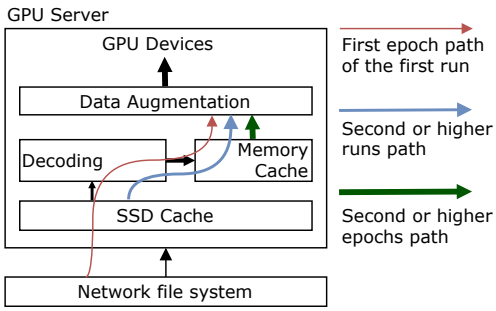


Figure 5. DataCache: Two-level caching for training data.

On public clouds, training data is generally stored in a central storage system (e.g., networked file system, NFS), whose reading performance may be limited by the network bandwidth and latency. On the other hand, training samples should be pre-processed before they are sent as the input of deep models. For example, in CNNs, the pre-processing process includes the decoding of input images (e.g., JPEG files) and normalization. Then the pre-processed data should be augmented (e.g., mirror, crop, etc.) before sent to GPU for training. To address the low performance of data access through the central storage and data pre-processing with CPU, we design a simple yet effective mechanism, DataCache, as shown in Fig. 5.

In the first epoch, the training data should be read from the central storage into the local machine with the local file system cache. The data should be further decoded to the format that can be fed as the inputs of the deep models, which could consume many CPU resources. We further cache the pre-processed data into memory using the key-value store, where the key is the sample index and the value is the pre-processed data. In this way, the memory cache can significantly reduce the I/O time during the training process. To reduce memory consumption, the full data set is split into multiple parts that are separately stored on multiple nodes. Starting from the second epoch, all data has been stored in the memory cache. With pipelining between data reading and GPU computations, the time cost of data reading from the memory cache can be almost fully overlapped by GPU computations. In summary, on one hand, enabling the local file system cache to store the data makes the multiple runs for hyper-parameter tuning be more efficient on data reading. On the other hand, using the memory cache makes the data reading from the second epochs at each run be fast.

4.2 Parallel Processing of Tensors

Processing tensors that are with the same data is very common in distributed training. In general, after the gradient aggregation, all GPUs have the same gradients and model parameters which should be further processed before update the model according to the optimizer. For example, the layer-wise adaptive rate scaling (LARS) (You et al., 2018) algorithm needs to calculate the layer-wise learning rates by

$$\lambda_t^{(l)} = \gamma \times \eta_t \times \frac{\|\boldsymbol{w}_t^{(l)}\|}{\|\boldsymbol{g}_t^{(l)}\| + \epsilon\|\boldsymbol{w}_t^{(l)}\|}, \tag{11}$$

where γ is a hyperparameter and ϵ is the weight decay. In the traditional training mechanism, all GPUs perform Eq. (11) in parallel. Since the input and output are the same in all GPUs, we can partition the workloads for different GPUs who first process different parts of data and then aggregate the generated results. We generalize this scheme as parallel processing tensors.

For any tensor $\boldsymbol{g} \in \mathbb{R}^d$ that has been stored in all P workers, if there is an operation $\text{OP}(\cdot)$ that takes $\boldsymbol{g}$ as the input and generates the same output, i.e.,

$$r = \text{OP}(\boldsymbol{g}), \tag{12}$$

then $\boldsymbol{g}$ can be partitioned into P pieces and each GPU only computes one piece with OP. Formally, for $p = 1, 2, ..., P$,

$$r^{[p])} = \text{OP}(\boldsymbol{g}^{[p]}) \tag{13}$$

can be parallelized on P GPUs, and the results are then aggregated by

$$r = \text{All-Gather}(r^{[p]}). \tag{14}$$

We use the parallel tensor operator (PTO) to denote the process of Eq. (13) (14). Obviously, using PTO can reduce the computation workload on each GPU by P times while it introduces an extra All-Gather communication overhead. In practice, if the time cost of the All-Gather operation is smaller than the time reduction of computing, PTO can accelerate the computation of Eq. (12).

PTO for LARS: For the LARS computation as shown in Eq. (11), we should calculate the norms of each layer's gradients and weights to generate the layer-wise learning rates.

We partition the workload in terms of the layer for different GPUs, which means different GPUs calculate different layers' learning rates that are finally gathered for every GPU. For example, in our 128-GPU experiments, the computations layer-wise learning rates for the ResNet-50 model, which has 161 layers, are distributed to the 128 GPUs. The first GPU calculates 1 to 2 layers' learning rates, the second one calculates layer 3 to 4, and so on. Finally, the layer-wise learning rates on the GPUs are all-gathered, which is with very low communication traffic as each layer's learning rate is a scalar. It would be similar to handle the case of LAMB (You et al., 2020) using PTO.

5 Experimental Studies

In this section, we first introduce our testbed, and then demonstrate the experimental results on the efficiency of our CommLib, DataCache, and PTO. After that, we compare the end-to-end training efficiency by putting the optimizations together. Finally, we present a case study in training ResNet-50 by breaking the record of DAWNBench in terms of the training time to the top-5 accuracy of 93%.

5.1 Experimental Environments

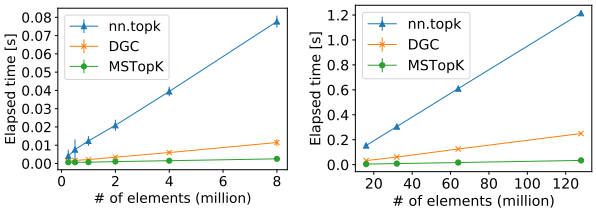
Testbed: We choose the cluster from Tencent Cloud with 16 GPU instances, each instance is a virtual machine equipped with 8 Nvidia Tesla V100-32GB GPUs connected with NVLink. The 16 instances are connected with the virtual private connection on 25Gbps Ethernet (25GbE). The hardware and software are the same in all instances. The hardware is shown in Table 1. The performance related libraries are: CUDA-10.1, cuDNN-7.6, NCCL-2.5.6, TensorFlow-1.15, and Hovorod-0.19.1.

DNNs: We choose two popular deep learning applications of computer vision using CNNs and natural language processing using Transformer. For CNNs, we choose ResNet-50 and VGG-19 on the ImageNet data set, while for Transformer, we choose Transformer[2] from (Vaswani et al., 2017) on the WMT17 [3] data set.

5.2 Top-k Operator Comparison

We compare the performance of our MSTopK operator with the naive top-k operator of TensorFlow (i.e., nn.topk) on a Tesla V100 GPU with different dimensions of vectors from 256 thousand to 128 million. We also implement the top-k selection with double sampling in (Lin et al., 2018), which we denote as DGC. The number of selected elements is thousandths of the dimension of the vector, that is $k = 0.001 \times d$. The experimental results are shown in Fig. 6.

[2] https://bit.ly/34H7tLB
[3] https://bit.ly/34Epbzx



(a) Small tensors. (b) Large tensors.

Figure 6. Time performance between MSTopK, DGC (Lin et al., 2018), and nn.topk. The elapsed time is the average time of 5 independent experiments, and for each experiment, we run 5 warmup iterations and 100 iterations to measure the average. The number of samplings for MSTopK is 30.

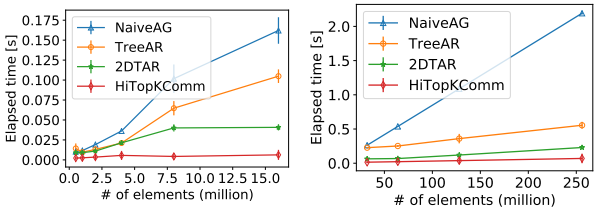
It can be seen that the exact top-k operator is very slow, while our MSTopK only requires negligible computing time. The exact top-k selection on the GPU generally requires irregular memory access which is not friendly to the GPU architecture (Shanbhag et al., 2018; Mei & Chu, 2016). DGC is much better than the naive implementation, but it is still not fast enough as it also requires the exact top-k selections. Our MSTopK is an approximate operator that eliminates the irregular memory access using the multiple thresholds, which significantly improves the GPU memory access bandwidth with coalesced access of a large number of threads (Cook, 2012). The results show that MSTopK significantly reduces the top-k selection time on GPUs.

In terms of the selected k elements from a vector with the approximate operator, our MSTopK typically generates more than 99% same results as the exact top-k operator. Therefore, SGD with MSTopK should be able to achieve similar convergence performance with TopK-SGD (details in §5.5.1).

5.3 Performance of HiTopKComm

To show the effectiveness of our proposed HiTopKComm, we compare the communication efficiency with the original sparsified aggregation with All-Gather (NaiveAG) using NCCL and the tree-based All-Reduce (TreeAR) from NCCL. We also implement the 2D-Torus All-Reduce (2DTAR) algorithm (Mikami et al., 2018; Cho et al., 2019) in our CommLib. 2DTAR can also exploit the hierarchical network connections to perform more efficient all-reducing. The results are shown in Fig. 7. Note that we use the 16-bit floating point (FP16) for each element which is widely used in V100 GPU clusters. It is seen that NaiveAG is extremely inefficient due to the traditional All-Gather is not friendly to the cloud GPU clusters that are with imbalance bandwidth in intra-node and inter-node connections. For TreeAR which is highly optimized in NCCL, it is also not that efficient in the cloud environment. 2DTAR can better utilize the bandwidth resource to achieve efficient data communication. HiTopKComm, as expected, is the most efficient communication

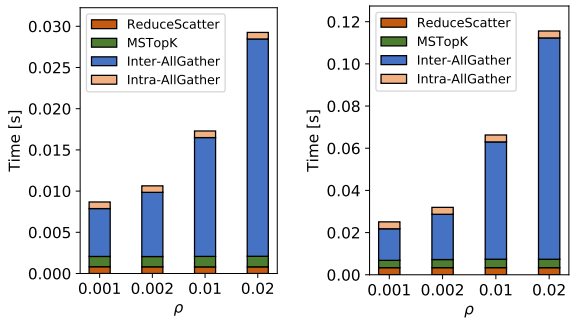
scheme among the four schemes we tested, which would help improve the system scalability for distributed training.



(a) Small tensors. (b) Large tensors.

Figure 7. Data aggregation time of different methods (NaiveAG, TreeAR, 2DTAR, and HiTopKComm). For the sparse communication, we use the density $\rho = 0.01$.

Time Breakdown of HiTopKComm. To understand the details of the efficiency of HiTopKComm, we breakdown the elapsed time for particular sizes of vectors according to the four steps (as shown in Fig. 3) of HiTopKComm. We use two specific cases (i.e., 25 million parameters for ResNet-50 and 110 million parameters for Transformer) using $k = 0.01d$, both of which are with FP32 for each element. The results are shown in Fig. 8. It is seen that the most time-consuming part is the inter-node communication with the All-Gather operation due to the low bandwidth of interconnects between multiple nodes. The time cost of top-k compression using MSTopK is very small, which is negligible. Due to the high-bandwidth and low latency of the intra-node connections between GPUs, the intra-node Reduce-Scatter and All-Gather operations also have a very small overhead.



(a) ResNet-50. (b) Transformer.

Figure 8. Time breakdown of HiTopKComm with different density.

5.4 DataCache and PTO

To show the benefits of data caching during the training process, we measure the end-to-end iteration time w/o and w/ our DataCache. The result is shown in Fig. 9. The I/O time is reduced over 10 times by using DataCache, and the

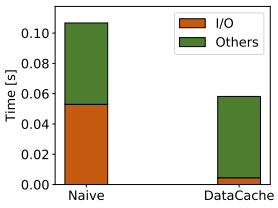


Figure 9. The performance comparison of training w/o (Naive) and w/ (DataCache) data caching. Using a V100 GPU on ResNet-50 with the input resolution of 96×96.

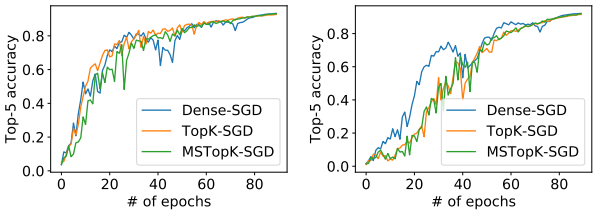
end-to-end training throughput is improved about 2 times.

To compare the performance of PTO, we also choose ResNet-50 and Transformer models to calculate LARS of Eq. (11) with randomly generated $\boldsymbol{w}^{(l)}$ and $\boldsymbol{g}^{(l)}$. The original LARS computing time on ResNet-50 and Transformer is 11ms and 30ms respectively, while our PTO takes 7ms and 14ms respectively. Our PTO achieves about $2\times$ speedups on our 128 GPU cluster on both ResNet-50 and Transformer.

5.5 End-to-end Training

Putting all optimizations together, we evaluate the end-to-end training time improvement over the existing solutions. Since our MSTopK is an approximate top-k selection operator which may generate different results compared to the exact top-k operator, we first compare the convergence of our MSTopK solution on the large-scale training, and then we compare the end-to-end training efficiency on the 128-GPU cluster.

5.5.1 Convergence



(a) ResNet-50. (b) VGG19.

Figure 10. Convergence comparison.

For the convergence evaluation, we measure two kinds of applications (CNNs and Transformer) using three training algorithms of the original dense version (Dense-SGD) with TreeAR, the original top-k sparsification (TopK-SGD), and our proposed top-k compression (MSTopK-SGD). For CNNs, we train the ResNet-50 and VGG-19 models on the 128-GPU cluster in 90 epochs with the standard input resolution of 224×224 and the local batch size is 256 (i.e., the global batch size is 32K). The validation accuracy during

the training process is shown in Fig. 10.

Table 2. Validation performance (top-5 accuracy for CNNs and BLEU for Transformer).

Model	2DTAR-SGD	TopK-SGD	MSTopK-SGD
ResNet-50	93.31%	92.68%	93.12%
VGG-19	92.19%	91.55%	91.94%
Transformer	26.74	24.42	24.16

In terms of validation performance, for CNNs, we use top-5 validation accuracy on the validation data set, and for Transformer (Vaswani et al., 2017), we measure BLEU on the validation data set by training 22k steps. The validation performance is shown in Table 2. Both the naive top-k sparsification and our MSTopK-SGD have slight accuracy loss compared to the dense version, which is reasonable due to the slower convergence speed of top-k sparsification than the dense version if the number of iterations is not large enough (Karimireddy et al., 2019).

5.5.2 *Training Efficiency*

We compare the end-to-end training efficiency on the 128-GPU cluster and scaling efficiency over the single GPU counterpart. Note that we enable the mixed-precision training technique so that the tensor cores of V100 GPUs can be utilized. The baseline throughput of single-GPU on ResNet-50, VGG-19, and Transformer are 1150, 560, and 32 samples/s, respectively. One sample indicates one image in CNNs, while it indicates one sentence with 256 words in Transformer. On our 128-GPU cluster, the performance comparison is shown in Table 3. The results show that Dense-SGD with TreeAR is very inefficient. For 2DTAR-SGD, it is slightly faster than our MSTopK-SGD in the case of ResNet-50 with the input resolution of 224*224 because the computing time is long enough to overlap some communication overheads. In all other cases, our MSTopK-SGD achieves 25%-40% improvement over 2DTAR-SGD.

5.6 A Case Study: Break the Record of DAWNBench

To further verify the effectiveness of our training system with the proposed novel optimizations, we present a case study on breaking the training time record of the ImageNet dataset on the leader-board of DAWNBench, which requires to achieve 93% top-5 accuracy on $100,000$ validation samples. Existing top-4 leaderships on DAWNBench all exploit 128 Tesla V100 GPUs with 100Gbps InfiniBand (100GbIB) or 32Gbps Ethernet (32GbE) interconnects, and the current fastest training speed on DAWNBench is 158 seconds by the team in Aliyun who exploited a 16-node GPU cluster (also with 128 V100 GPUs) connected with 32GbE. However, we exploit the Tencent Cloud cluster that is also with 16 GPU nodes while the interconnect between nodes is 25GbE, which is more challenging than that from Aliyun.

In recent practice, 93% top-5 accuracy is achievable within only about 30 epochs using dynamic input image size during training. For example, in the current top-1 leadership on DAWNBench, the Alibaba team trained the model to achieve 93% top-5 accuracy on the ImageNet data set with 28 epochs, where the first 13 epochs are with 96×96 input resolution, the second 11 epochs with 128×128 input resolution, the third 3 epochs with 224×224 input resolution, and the last epoch with 288×288 input resolution. On one hand, the small input size can achieve higher system throughput, but it makes the scaling be more difficult. On the other hand, the large input size has smaller system throughput, but it is helpful to achieve high validation accuracy. In the ImageNet training, it is known that the warmup process is necessary to preserve the model accuracy (Goyal et al., 2017; You et al., 2018; Jia et al., 2018), so recent studies [4] notice that in the first 10-20 epochs, smaller input size can also warmup the training while achieving higher system throughput. We follow the 28-epoch training practice to 93% top-5 accuracy on our 16-node cluster (128 V100 GPUs) with lower bandwidth interconnect (i.e., 25GbE).

In the first 13 epochs, the computing time of 96×96 input is short on the V100 GPU, which makes the scaling efficiency very low using dense gradient aggregation. Therefore, we use MSTopK-SGD to train the model in the first 13 epochs, which exploits HiTopKComm for gradient aggregation to achieve higher system scaling efficiency. After that, we switch to use 2DTAR-SGD to balance the convergence speed and the system throughput. We cannot fully use MSTopK-SGD in the whole of 28 epochs because it would cause accuracy loss. Since when the input resolution is not smaller 128×128, the scaling efficiency is acceptable, we start to use dense gradient communication so that we can achieve 93% top-5 accuracy in 28 epochs. The training throughput with different input resolution is shown in Table 4, and the training time using 128 Nvidia V100 GPUs to achieve 93% validation accuracy of the ImageNet data set is shown in Table 5. The results show that our method achieves faster training time even with slower interconnects between GPU nodes.

6 Related Work

Much work has tried to improve scaling efficiency of distributed training while preserving convergence properties.

Regarding gradient aggregation, there are many algorithms to increase communication efficiency and reduce the communication traffic. For example, double binary trees (Sanders et al., 2009) All-Reduce has been integrated in NCCL to support large-scale communication in the HPC

[4] https://bit.ly/33KVbCO

Table 3. System throughput and scaling efficiency of different SGD algorithms.

Model	System Throughput (samples/s)			Scaling Efficiency (%)		
	Dense-SGD	2DTAR-SGD	MSTopK-SGD	Dense-SGD	2DTAR-SGD	MSTopK-SGD
ResNet-50 (224*224)	64000	**134656**	133376	43.5	**91.4**	90.6
ResNet-50 (96*96)	113280	313600	**396800**	20.1	56.7	**70.5**
VGG-19	17920	47616	**57600**	25	66.4	**80.4**
Transformer	678	2534	**3502**	16.5	61.6	**87.8**

Table 4. System throughput (samples/s) with different input resolutions. BS indicates the local batch size at each GPU, and SE is shorten for scaling efficiency of the 128-GPU system.

# Epochs	Input	BS	Single-GPU	128-GPU (SE)
13	96×96	256	4400	366,208 (65%)
11	128×128	256	3010	269,696 (70%)
3	224×224	256	1240	131,712 (83%)
1	288×288	128	710	72,960 (80%)

Table 5. Time to 93% top-5 accuracy with 128 Tesla V100 GPUs.

Team	Date	Interconnect	Time (seconds)
FastAI	Sep 2018	100GbIB	1086
Huawei	Dec 2018	-	562
Huawei	May 2019	100GbIB	163
Alibaba	Mar 2020	32GbE	158
Ours	Aug 2020	**25GbE**	**151**

environment. Some All-Reduce algorithms (Goyal et al., 2017; Jia et al., 2018; Mikami et al., 2018; Cho et al., 2019; Wang et al., 2020; Luo et al., 2020; Dong et al., 2020; Chu et al., 2020) have been proposed to efficiently aggregate data in different environments. To reduce the communication traffic, gradient compression techniques including quantization and sparsification are widely studied (Alistarh et al., 2017; Lin et al., 2018; Shi et al., 2019d; Karimireddy et al., 2019; Shi et al., 2019c; Vogels et al., 2019).

Large-batch training is an effective way to improve the system scalability by increasing the workload of GPUs and thus reducing the communication-to-computation ratio, but it requires careful hyperparameter tuning to preserve the model performance. LARS (You et al., 2018) and LAMB (You et al., 2020) are two main techniques to address the convergence problem in large-batch training. AdaScale SGD (Johnson et al., 2020) is another technique to make large-batch training be easier to converge.

In GPU computing, most tensor operations highly rely on CUDA libraries like cuDNN, cuBLAS, etc. Most operators in training deep models are well supported by cuDNN, but the top-k operator is extremely inefficient. (Shanbhag et al., 2018) tried to optimize the top-k query in the database area. The double sampling technique was proposed in (Lin et al., 2018) to reduce the number of elements for the top-k operator, but it also requires at least two times of top-k operations on GPUs. In (Fang et al., 2019; Shi et al., 2019a), the authors proposed multiple sampling methods to approximate the top-k operation, but they may result in a different number of elements at different workers, which could make the data aggregation be inefficient. Simultaneous to our work, (Abdelmoniem et al., 2021) proposed a multi-stage threshold estimator to overcome the far tail estimation problem.

As for I/O optimizations, (Pumma et al., 2019) identified the inefficiency of data access for multi-GPU training, but they mainly focused on the problem of LMDB data that is relatively old in the Caffe framework. Recently, (Zhang et al., 2020) proposed FanStore for efficient I/O that is particularly used on supercomputers.

7 Conclusion

In this paper, we proposed a novel gradient communication library for distributed training systems, in which we proposed a GPU-friendly approximate top-k operator for gradient sparsification and a hierarchical communication for sparsified gradients. and we also optimize I/O and GPU computation to further improve the system scalability. Specifically, first, for I/O, we proposed a multi-level data caching scheme to reduce the I/O time on public clouds that generally use central servers for data storage. Second, to improve the GPU efficiency on some operators, we design a GPU-friendly top-k sparsification operator and a parallel tensor operator that can better utilize multi-GPU computing power. Third, we proposed a hierarchical top-k communication for sparsified gradients. Extensive experiments were conducted on a 128 Tesla V100 GPU cluster (16 nodes) with 25Gbps Ethernet. Experimental results showed that our system outperforms the existing state-of-the-art solutions and breaks the record on training ResNet-50 to 93% top-5 accuracy on DAWNBench.

Acknowledgements

The research was supported in part by Hong Kong RGC GRF grants under the contracts HKBU 12200418 and RMGS2019_1_23 from Hong Kong Research Matching Grant Scheme.

REFERENCES

Abdelmoniem, A. M., Elzanaty, A., Alouini, M.-S., and Canini, M. An efficient statistical-based gradient compression technique for distributed training systems. In *Proc. Machine Learning and Systems (MLSys) Conference*, 2021.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pp. 1709–1720, 2017.

Alistarh, D., Hoefler, T., Johansson, M., Konstantinov, N., Khirirat, S., and Renggli, C. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*, pp. 5973–5983, 2018.

Awan, A. A., Hamidouche, K., Hashmi, J. M., and Panda, D. K. S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 193–205, 2017.

Baidu. *Baidu Ring All-Reduce*, 2017. URL https://github.com/baidu-research/baidu-allreduce.

Cho, M., Finkler, U., Kung, D. S., and Hunter, H. C. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, 2019.

Chu, C.-H., Kousha, P., Awan, A. A., Khorassani, K. S., Subramoni, H., and Panda, D. K. Nv-group: link-efficient reduction for distributed deep learning on modern dense GPU systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pp. 1–12, 2020.

Cook, S. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

Dong, J., Cao, Z., Zhang, T., Ye, J., Wang, S., Feng, F., Zhao, L., Liu, X., Song, L., Peng, L., et al. EFLOPS: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 610–622. IEEE, 2020.

Fang, J., Fu, H., Yang, G., and Hsieh, C.-J. RedSync: Reducing synchronization bandwidth for distributed deep learning training system. *Journal of Parallel and Distributed Computing*, 133:30–39, 2019.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Jia, X., Song, S., Shi, S., He, W., Wang, Y., Rong, H., Zhou, F., Xie, L., Guo, Z., Yang, Y., Yu, L., Chen, T., Hu, G., and Chu, X. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. In *Proc. of Workshop on Systems for ML and Open Source Software, collocated with NeurIPS 2018*, 2018.

Johnson, T. B., Agrawal, P., Gu, H., and Guestrin, C. AdaScale SGD: A user-friendly algorithm for distributed training. In *International Conference on Machine Learning*, 2020.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

Karimireddy, S. P., Rebjock, Q., Stich, S., and Jaggi, M. Error feedback fixes SignSGD and other gradient compression schemes. In *International Conference on Machine Learning*, pp. 3252–3261, 2019.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 583–598, 2014.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

Luo, L., West, P., Nelson, J., Krishnamurthy, A., and Ceze, L. PLink: Efficient cloud-based training with topology-aware dynamic hierarchical aggregation. In *Proceedings of the 3rd MLSys Conference*, 2020.

Mei, X. and Chu, X. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

Mikami, H., Suganuma, H., Tanaka, Y., Kageyama, Y., et al. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

Pumma, S., Si, M., Feng, W.-C., and Balaji, P. Scalable deep learning via I/O analysis and optimization. *ACM Transactions on Parallel Computing (TOPC)*, 6(2):1–34, 2019.

Renggli, C., Ashkboos, S., Aghagolzadeh, M., Alistarh, D., and Hoefler, T. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2019.

Sanders, P., Speck, J., and Träff, J. L. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

Shanbhag, A., Pirk, H., and Madden, S. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1557–1570, 2018.

Shi, S., Chu, X., Cheung, K. C., and See, S. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772*, 2019a.

Shi, S., Chu, X., and Li, B. MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 172–180. IEEE, 2019b.

Shi, S., Wang, Q., Zhao, K., Tang, Z., Wang, Y., Huang, X., and Chu, X. A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2238–2247. IEEE, 2019c.

Shi, S., Zhao, K., Wang, Q., Tang, Z., and Chu, X. A convergence analysis of distributed sgd with communication-efficient gradient sparsification. In *IJCAI*, pp. 3411–3417, 2019d.

Shi, S., Wang, Q., Chu, X., Li, B., Qin, Y., Liu, R., and Zhao, X. Communication-efficient distributed deep learning with merged gradient sparsification on GPUs. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020.

Stich, S. U., Cordonnier, J.-B., and Jaggi, M. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*, pp. 4452–4463, 2018.

Ueno, Y. and Yokota, R. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Vogels, T., Karimireddy, S. P., and Jaggi, M. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, pp. 14259–14268, 2019.

Wang, G., Venkataraman, S., Phanishayee, A., Thelin, J., Devanur, N., and Stoica, I. Blink: Fast and generic collectives for distributed ML. In *Proceedings of the 3rd MLSys Conference*, 2020.

Wang, S., Li, D., Cheng, Y., Geng, J., Wang, Y., Wang, S., Xia, S.-T., and Wu, J. BML: A high-performance, low-cost gradient synchronization algorithm for DML training. In *Advances in Neural Information Processing Systems*, pp. 4238–4248, 2018.

You, Y., Zhang, Z., Hsieh, C.-J., Demmel, J., and Keutzer, K. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10, 2018.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. Large batch optimization for deep learning: Training BERT in 76 minutes. In *International Conference on Learning Representations*, 2020.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 181–193, 2017.

Zhang, Z., Huang, L., Pauloski, J. G., and Foster, I. T. Efficient I/O for neural network training with compressed data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 409–418. IEEE, 2020.