A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters

Yimin Jiang, *Tsinghua University and ByteDance;* Yibo Zhu, *ByteDance;* Chang Lan, *Google;* Bairen Yi, *ByteDance;* Yong Cui, *Tsinghua University;* Chuanxiong Guo, *ByteDance*

https://www.usenix.org/conference/osdi20/presentation/jiang

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters

Yimin Jiang*†, Yibo Zhu†, Chang Lan‡, Bairen Yi†, Yong Cui*, Chuanxiong Guo†
*Tsinghua University, †ByteDance, ‡Google

Abstract

Data center clusters that run DNN training jobs are inherently heterogeneous. They have GPUs and CPUs for computation and network bandwidth for distributed training. However, existing distributed DNN training architectures, all-reduce and Parameter Server (PS), cannot fully utilize such heterogeneous resources. In this paper, we present a new distributed DNN training architecture called BytePS. BytePS can leverage spare CPU and bandwidth resources in the cluster to accelerate distributed DNN training tasks running on GPUs. It provides a communication framework that is both *proved optimal* and unified – existing all-reduce and PS become two special cases of BytePS. To achieve the proved optimality in practice, BytePS further splits the functionalities of a parameter optimizer. It introduces a *Summation Service* abstraction for aggregating gradients, which is common for all the optimizers. Summation Service can be accelerated by AVX instructions and can be efficiently run on CPUs, while DNN model-related optimizer algorithms are run on GPUs for computation acceleration. BytePS can accelerate DNN training for major frameworks including TensorFlow, PyTorch and MXNet. For representative DNN training jobs with up to 256 GPUs, BytePS outperforms the state-of-the-art open source all-reduce and PS by up to 84% and 245%, respectively.

1 Introduction

In recent years, research on Deep Neural Networks (DNNs) has experienced a renaissance. DNNs have brought breakthroughs to computer vision [32, 43], speech recognition and synthesis [33, 69], natural language processing (NLP) [26], and many other areas. Training these DNN models usually requires a huge amount of arithmetic computation resources. Consequently, GPUs are preferred. To run many such tasks and achieve high resource utilization, large GPU clusters with thousands or more GPUs are introduced [29, 35, 52, 71].

Such GPU clusters have not only GPUs, but also CPUs and high speed networks. GPU machines typically also have high-end CPUs [2, 11]. There may also be CPU-only machines used for training data pre-processing and generation, *e.g.,* in reinforcement learning. These GPU/CPU machines are connected by high-speed Ethernet or Infiniband network to facilitate distributed training. Based on our experience in operating production GPU clusters (§3.1) and recent literature from others [35], GPUs are usually better utilized while there are often spare CPU and bandwidth resources.

There are two major families of distributed training architectures, all-reduce [54] and Parameter Server (PS) [44]. They are both based on data parallelism (§2). In a task that uses all-reduce, only GPU machines are involved. In an iteration, GPUs compute the gradients of the model parameters independently, and then aggregate gradients using the all-reduce primitive. In PS tasks, both GPU machines and CPU machines can be used. Different from all-reduce, the gradients are sent to PS, which typically runs on CPU machines and aggregates the received gradients. PS then runs certain DNN training optimizer, *e.g.,* SGD [76] or Adam [42] and sends back the updated model. For both all-reduce and PS, the above happens in every iteration, until the training finishes.

All-reduce and PS are quite different, in both theory and practice. Given a set of GPU machines *without* additional CPU machines, all-reduce is proved to be bandwidth optimal [54]. However, *with* additional CPU and bandwidth resources, the optimality of all-reduce no longer holds – we find that, *in theory*, PS can offer even better performance by utilizing additional CPU machines to aid the GPU machines (§2). It seems to be a good opportunity to accelerate DNN training because GPU clusters indeed have spare CPU and bandwidth resources (§3.1). Unfortunately, *in practice,* all the existing PS have inferior performance for multiple design reasons, as we shall see soon in this paper. It is therefore not a surprise to see that distributed DNN training speed records are dominated by all-reduce [27, 49, 73].

We are thus motivated to design *BytePS* [1], an architecture that is communication-optimal, both in theory and in practice. Fundamentally, both all-reduce and PS are theoretically optimal only in very specific GPU/CPU setups, while are not

[1]The name BytePS was chosen in the early stage of this project [4]. However, it is conceptually different from the conventional PS architecture.

the optimal for more generic settings, *e.g.,* there are some finite additional CPU resources. By carefully allocating traffic loads, BytePS unifies the cases where PS or all-reduce is theoretically optimal, and generalizes the optimality to any given number of GPU/CPU machines with different PCIe/NVLink configurations, with analytical proofs.

On top of that, BytePS pushes its real-world performance close to the theoretical limit, by removing bottlenecks in existing PS designs. With fast high-speed networks, we found that CPUs are not fast enough for the full fledged DNN optimizers. We introduce a new abstraction, *Summation Service*, to address this issue. We split an optimizer into gradient aggregation and parameter update. We keep gradient aggregation in Summation Service running on CPUs and move parameter update, which is more computation intensive, to GPUs. In addition, in implementation, we incorporated the idea of pipelining and priority-scheduling from prior work [34, 55] and resolved multiple RDMA-related performance issues.

As a drop-in replacement for all-reduce and PS, BytePS aims to accelerate distributed training without changing the DNN algorithm or its accuracy at all. Prior work on top of all-reduce and PS, like tensor compression [21, 45], can directly apply to BytePS. Our BytePS implementation supports popular DNN training frameworks including TensorFlow [20], PyTorch [53], and MXNet [22] with Horovod-like [60] API and native APIs.

This paper makes the following contributions:

- We design a new distributed DNN training architecture, BytePS, for heterogeneous GPU/CPU clusters. With spare CPU cores and network bandwidth in the cluster, BytePS can achieve communication optimality [2] for DNN training acceleration. BytePS provides a unified framework which includes both all-reduce and PS as two special cases.
- We further optimize the intra-machine communication. We explain the diverse and complicated topology in GPU machines and present the optimal strategy and principles.
- We propose Summation Service, which accelerates DNN optimizers by keeping gradient summation running in CPUs, and moving parameter update, which is the more computation intensive, to GPUs. This removes the CPU bottleneck in the original PS design.

As a major online service provider, we have deployed BytePS internally and used it extensively for DNN training. We evaluate BytePS using six DNN models and three training frameworks in production data centers. The results show that with 256 GPUs, BytePS consistently outperform existing all-reduce and PS solutions by up to 84% and 245%, respectively. We also released an open source version [4], which attracted interests from thousands in the open source community, several top-tier companies and multiple research groups.

[2]The optimality means to achieve minimized communication time for data-parallel distributed DNN training, given a fixed number of GPUs.

2 Background

2.1 Distributed DNN Training

A DNN model consists of many *parameters*. DNN training involves three major steps: (1) *forward propagation (FP)*, which takes in a *batch* of training data, propagates it through the DNN model, and calculates the *loss function*; (2) *backward propagation (BP)*, which uses the loss value to compute the *gradients* of each parameter; (3) *parameter update*, which uses the aggregated gradients to update the parameters with a certain optimizer (e.g., SGD [76], Adam [42], etc.). Training a DNN refines the model parameters with the above three steps iteratively, until the loss function reaches its minimal.

On top of it, users can optionally run distributed training. The most popular distributed DNN training approach is *data parallelism*, which partitions the dataset to multiple distributed computing devices (typically GPUs) while each GPU holds the complete DNN model. Since the data input to each GPU is different, the gradients generated by BP will also be different. Thus data parallelism demands all GPUs to synchronize during each training iteration.

In large enterprises or in public clouds, users often run these DNN training tasks in shared GPU clusters. Such clusters are built with hundreds to thousands of GPU machines connected by high-speed RDMA networks [35, 52]. Those GPU machines typically have multiple GPUs, tens of CPU cores, hundreds of GB of DRAM, and one to several 100Gb/s NICs. These clusters run many training jobs simultaneously, with many jobs using GPUs intensively while not using CPUs heavily. A public dataset on a DNN cluster [35] indicates that 50% of hosts have CPU utilization lower than 30%.

For distributed training, there are two families of data parallelism approaches, *i.e.*, all-reduce and Parameter Server (PS). In what follows, we introduce all-reduce and PS and analyze their communication overheads. We assume that we have n GPU machines for a data-parallel training job. The DNN model size is M bytes. The network bandwidth is B.

2.2 All-reduce

Originated from the HPC community, *all-reduce* aggregates every GPU's gradients in a collective manner before GPUs update their own parameters locally. In all-reduce, no additional CPU machine is involved. *Ring* is the most popular all-reduce algorithm. All-reduce has been optimized for many years, and most state-of-the-art training speed records are achieved using all-reduce, including classical CNN-based ImageNet tasks [27, 36, 49, 73], RNN-based language modeling tasks [56], and the pre-training of BERT [26, 74].

Fig. 1 shows an example of ring-based all-reduce for three nodes. We can dissect an all-reduce operation into a *reduce-scatter* and an *all-gather*. Reduce-scatter (Fig. 1(a)) partitions the whole M bytes into n parts, and use n rings with different starting and ending point to reduce the n parts, respectively. Each node will send $(n-1)M/n$ traffic, because each node

acts as the last node for just 1 ring and thus sends 0, while for each of the other $n-1$ rings, it must send M/n bytes.

Next, all-gather requires each node to broadcast its reduced part to all other $(n-1)$ nodes using a ring. In the end, all nodes have identical data that have been all-reduced (Fig. 1(c)). Similar to reduce-scatter, each node also sends $(n-1)M/n$ egress traffic during this operation.

Adding the two steps together, in an all-reduce operation, each node sends (and receives) $2(n-1)M/n$ traffic to (and from) the network. With B network bandwidth, the time required is $2(n-1)M/nB$, which is proved to be the optimal in topologies with uniformed link bandwidth [54], assuming *no additional resources*.

In hierarchical topologies with non-uniformed link bandwidth, the optimal hierarchical strategy would require at least $2(n'-1)M/n'B'$ communication time, where B' is the slowest link bandwidth and n' is the number of nodes with the slowest links. In distributed DNN training, n' is usually the number of GPU machines and B' is usually the network bandwidth per machine. For simplicity and without impacting our analysis, below we assume each machine has just one GPU and is connected by the same network bandwidth, *i.e.,* $n=n', B=B'$.

All-reduce has no way to utilize *additional* non-worker nodes, since it was designed for homogeneous setup. Next, we will show that the $2(n-1)M/nB$ communication time is no longer optimal with additional CPU machines.

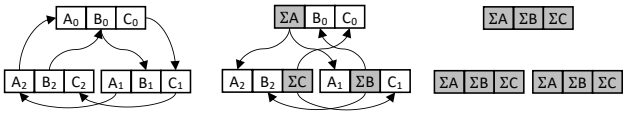
2.3 Parameter Server (PS)

The PS architecture [44] contains two roles: *workers* and *PS*. Workers usually run on GPU machines, perform FP and BP, and *push* the gradients to PS. PS aggregates the gradients from different workers and update the parameters. Finally, workers *pull* the latest parameters from PS and start the next iteration. According to our experience in industry, the PS processes usually run on CPUs because of cost-effectiveness. Since GPUs (and GPU memory) are much more expensive than CPUs,[3] we want GPUs to focus on the most computation-intensive tasks instead of storing the model parameters.

There are two placement strategies for PS. One is *non-colocated mode* (Fig. 2(a)), in which PS processes are deployed on dedicated CPU machines, separate from the GPU machines. Suppose that we have k CPU machines,[4] the DNN model will be partitioned into k parts and stored on the k machines, respectively. In every iteration, each GPU worker must send M bytes gradients and receives M bytes parameters back. Each CPU machine must receive in total nM/k gradients from the GPU workers and send back nM/k parameters.

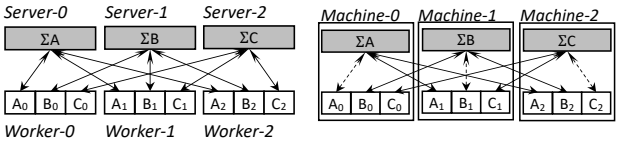
[3] AWS price sheet [18] shows that p3.16xlarge (8 NVIDIA V100 GPUs and 64 CPU cores) costs nearly $25 per hour. However, r4.16xlarge, which is the same as p3.16xlarge minus GPUs, costs only $4.2 per hour.

[4] In this paper, for simplicity, we assume that a CPU machine has the same network bandwidth as a GPU machine. If not, all analysis and design will remain valid as long as the number of CPU machines scales accordingly. For example, use 4× CPU machines if their bandwidth is 25% of GPU machines.



(a) Reduce-scatter (b) All-gather (c) Result

Figure 1: The communication workflow of all-reduce.



(a) Non-colocated mode (b) Colocated mode

Figure 2: The communication pattern of PS. A solid arrow line indicates the network traffic. A dashed arrow line represents the loop-back (local) traffic.

Assuming $k=n$, PS would theoretically be faster than all-reduce, as summarized in Table 1. In fact, PS is communication optimal in such setting, since M is the absolute lower bound each GPU machine has to send and receive. However, with fewer CPU machines (smaller k), the communication time nM/kB on CPU machines would increase and, if $k \le n/2$, become slower than all-reduce. The network bandwidth of GPU machines would become under-utilized because the CPU machines would be the communication bottleneck.

The other strategy is *colocated mode* (Fig. 2(b)), which does not use any CPU machines. Instead, it starts a PS process on every GPU worker and reuses its spare CPU resources. The PS and GPU worker on the same machine will communicate through loopback traffic. In this case, it is easy to calculate that communication time is the same as all-reduce (Table 1).

All-reduce vs. PS. They have different communication patterns. PS uses a bipartite graph. Non-colocated PS can leverage additional CPU and bandwidth resources to aid GPU machines, while may under-utilize the resources of GPU machines. Colocated PS and all-reduce utilize the GPU worker resources better, while cannot use additional CPU machines.

Another difference is that PS supports *asynchronous* training, which allows GPU workers to run at different speed and mitigates the impact of stragglers, while all-reduce does not support it. However, asynchronous training is less popular because it can slow down model convergence. We will mainly focus on synchronous training in this paper while briefly address asynchronous training in §5.

3 Motivation and BytePS Architecture

3.1 Motivation

Before the deployment of BytePS in our internal GPU clusters, our users mostly used all-reduce as the distributed training architecture due to its higher performance than existing PS designs. The remaining users choose PS for tasks where asynchronous training is acceptable or preferable. With multiple years of experience and efforts on accelerating DNN tasks and improving resource utilization, we have the following observation.

Table 1: The theoretical communication time required by each training iteration. n is the number of GPU machines. k is the number of additional CPU machines. M is the model size. B is the network bandwidth. We will revisit the *Optimal?* row in §4.1.

	All-reduce	Non-Colocated PS	Colocated PS
Time	$\frac{2(n-1)M}{nB}$	$\max(\frac{M}{B}, \frac{nM}{kB})$	$\frac{2(n-1)M}{nB}$
Optimal?	Only if $k=0$	Only if $k=n$	Only if $k=0$

Opportunity: there are spare CPUs and bandwidth in production GPU clusters. Large-scale GPU clusters simultaneously run numerous jobs, many of which do not heavily use CPUs or network bandwidth. Fig. 3 shows a 3-month trace collected from one of our GPU clusters that have thousands of GPUs. The GPUs have been highly utilized in that period (approaching 96% allocation ratio in peak times). We find that, 55%-80% GPU machines have been assigned as GPU workers for at least one *distributed* training task. This leaves the network bandwidth of 20%-45% GPU machines unused because they are running *non-distributed* jobs.[5] The cluster-wide average CPU utilization is only around 20%-35%. This aligns with the findings in prior work from Microsoft [35].

This observation, combined with the all-reduce vs. non-colocated PS analysis in §2.1, inspires us – if we can better utilize these spare CPUs and bandwidth, it is possible to accelerate distributed training jobs running on given GPUs.

Existing all-reduce and PS architectures are insufficient. Unfortunately, the analysis in §2.1 also shows that all-reduce and PS have a common issue: they do not utilize additional CPU and bandwidth resources well. All-reduce and colocated PS only use resources on GPU workers, and non-colocated PS may not fully utilize the CPU cores and NIC bandwidth on GPU workers. The former is communication optimal only when $k=0$, while the latter is optimal only when $k=n$. When the number of CPU machine k is $0<k<n$, neither would be optimal. We defer further analysis to §4.1. Here, we use an experiment to show the end-to-end performance of existing all-reduce and PS.

Fig. 4 shows the training speed of VGG-16 [63] using 32 V100 GPUs (4 GPU machines), with 100GbE RDMA network. The batch size is 32 images for each GPU. We run the latest MXNet native PS RDMA implementation [1] and (one of) the most popular all-reduce library NCCL-2.5.7 [13]. We also tested TensorFlow's native PS, and got similar results. We vary the number of additional CPU machines for each setup. All-reduce plot is flat because additional CPU machines are of no use, while PS has the worst performance even with additional CPU machines. Both of them are far from optimal. Even with ByteScheduler [55], which is a state-of-the-art technique that can improve the communication performance, both all-reduce and PS are still far from the linear scaling, *i.e.,* 32× of single-GPU training speed. This is because ByteScheduler works on top of PS or all-reduce, and thus has the same limitations. BytePS outperforms all of above at any given number of CPU machines (more in §7).

[5] Our machines have dedicated but slower NIC for data I/O. This is a common practice in industry [52]. In addition, data I/O traffic is usually much smaller than the distributed training traffic between GPU machines.

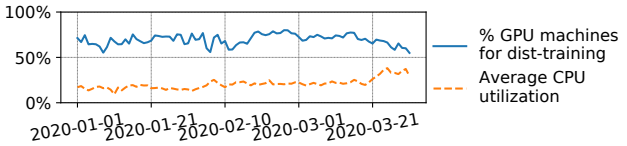


Figure 3: Daily statistics of our internal DNN training clusters from 2020-01-01 to 2020-03-31.

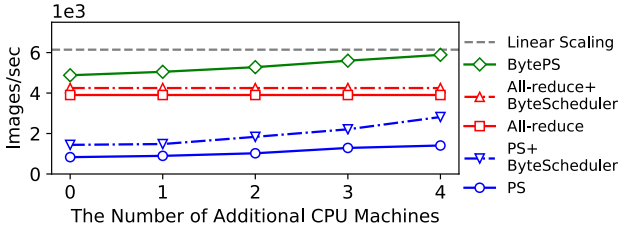


Figure 4: VGG-16 training performance of different architectures. We use 4 GPU machines with 32 GPUs in total. *Linear Scaling* represents the maximal performance (in theory) of using 32 GPUs.

Our solution: BytePS. It is a unified architecture for distributed DNN training that can leverage spare CPU and bandwidth resources. It achieves the following goals.

First, BytePS is always communication optimal with any additional CPU and bandwidth resources, *i.e.,* $0 \le k \le n$, allocated by the cluster scheduler. In practice, the volume of spare resources can be dynamic (Fig. 3), so BytePS must adapt well. In addition, the hardware setup of GPU machines can be diverse, especially the internal PCIe or NVLink topology. BytePS is also *proved* optimal in intra-machine communication. All-reduce and PS, when they are communication optimal, are two special cases of BytePS (§4).

Second, BytePS can achieve communication time very close to the theoretical optimal. This is important, as shown in the existing PS case – PS performance is far from its theoretical limit. We found that original PS designs have several implementation bottlenecks (which we will discuss in §6). But even after all the bottlenecks are removed, PS performance is still inferior to optimal. This leads to BytePS's second design contribution: *Summation Service*. We find that running the full optimizers on CPU can be a bottleneck. We divide the computation of optimizers and only put summation on CPUs. We will elaborate the rationale of this design in §5.

All the BytePS designs are generic to DNN training. BytePS can therefore accelerate various DNN training frameworks including TensorFlow, PyTorch, and MXNet. We start from presenting BytePS's architecture.

3.2 Architecture Overview

Fig. 5 shows the architecture of BytePS. BytePS has two main modules – *Communication Service (CS)* and *Summation Service (SS)*. In BytePS, we aim to leverage any CPU resources, whether on GPU machines or CPU machines, to

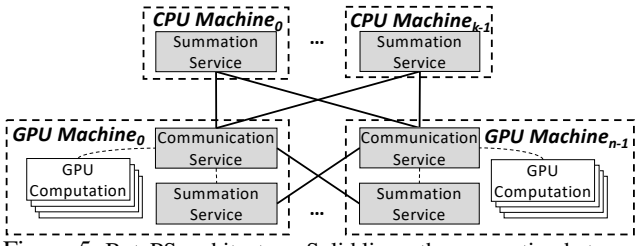


Figure 5: BytePS architecture. Solid lines: the connection between CPU machines and GPU machines. Dashed lines: the data flow inside GPU machines.

achieve the best communication efficiency. This is achieved by SS, which runs on the CPU of *every* machine, including the CPU machines and GPU machines. The CPU machines may not necessarily be actual CPU-only machines. For example, our in-house cluster scheduler can allocate CPUs on the GPU machines that run non-distributed jobs and have spare CPU cores and network bandwidth. This improves the overall cluster resource utilization.

Another important property of SS is that it is much simpler than common PS server processes, which run full fledged DNN algorithm optimizers. In contrast, SS is only responsible for receiving tensors that are sent by CS, summing up the tensors and sending them back to CS.

The other module, CS, is responsible for internally synchronizing the tensors among multiple (if there are) local GPUs and externally communicating with SS. Every training iteration, each CS must send in total M bytes (the DNN model size) to and receive M bytes from SS. In synchronous distributed training, the tensors are model gradients.

CS contains several design points of BytePS. First, it decides the traffic volume to each SS (both internal and external). The load assignment strategy is based on our analysis of the optimal communication strategy (§4.1). Second, it chooses the best local tensor aggregation strategy depending on different internal GPU and NIC topology (§4.2) of the GPU machines. Finally, both CS and SS should be optimized for RDMA in modern high-speed data centers (§6.2).

This architecture enables BytePS to flexibly utilize any number of additional CPU resources and network bandwidth. When the number of CPU machines is 0, *i.e.,* $k = 0$, the communication will fallback to only using SSs on GPU machines. When the number of CPU machines is the same as GPU machines, BytePS is as communication optimal as non-colocated PS. In other cases, BytePS can leverage SSs on all machines together. In fact, our analytical results will reveal the optimal communication strategy with any number of CPU machines, while PS and all-reduce are just two specific points in the whole problem space.

4 BytePS Communication Design

4.1 Inter-machine Communication

In BytePS, all networking communication is between CS and SS. To prevent a bottleneck node from slowing down the whole system, we must balance the communication time of all machines. In what follows, we assume the network has full bisection bandwidth, which is a common practice in deep learning clusters [52]. We also assume that the full bisection bandwidth can be fully utilized due to the newly introduced RDMA congestion control algorithms, e.g., DCQCN [75].

On each CPU machine, the summation workload of its SS determines the network traffic. For example, if a SS is responsible for summing up $x\%$ of the DNN model, the CPU machine would send and receive $x\% \times M$ bytes traffic to *every* GPU machine during each training iteration. However, the network traffic of a GPU machine is determined by the combination of CS and SS running on it. Due to this difference, BytePS classifies SS into SS_{CPU} and SS_{GPU} based on whether they run on CPU machines or GPU machines.

To minimize the communication time, BytePS assigns $M_{SS_{CPU}}$ bytes summation workload to each SS_{CPU}. $M_{SS_{CPU}}$ is given in Eq. 1, where $k \geq 1$ is the number of CPU machines and $n \geq 2$ is the number of GPU machines, and $k \leq n$. Outside these constraints, the communication time of BytePS falls back to trivial solutions like PS (when $k > n$) and all-reduce (when $k = 0$), as §4.1.1 shows.

$$M_{SS_{CPU}} = \frac{2(n-1)}{n^2+kn-2k}M \tag{1}$$

Similarly, BytePS assigns $M_{SS_{GPU}}$ bytes to each SS_{GPU}.

$$M_{SS_{GPU}} = \frac{n-k}{n^2+kn-2k}M \tag{2}$$

Eq. 1 and Eq. 2 show the workload assignment strategy that is optimal for minimizing the communication time. The analysis is in §4.1.1. In practice, the DNN model consists of tensors with variable sizes and may not allow us to perfectly assign workloads. BytePS uses an approximation method. It partitions the tensors into small parts no larger than 4MB.[6] Then, all CSs consistently index each part and hash the indices into the range of $[0, n^2+kn-2k)$. CSs will send and receive tensors to SSs based on the hash value and approximate the probabilities according to Eq. 1 and Eq. 2. Consistent indexing and hashing guarantee that the same part from all GPUs will be sent to and processed by the same SS.

4.1.1 Communication Efficiency Analysis

Next, we present the communication time analysis of BytePS. To simplify the analysis, we assume that the model size M is much larger than the partition size (4MB in our case). Partitioning enables BytePS not only to better balance the summation workloads, but also to well utilize the bidirectional network bandwidth by pipelining sending and receiving, as shown in [34, 55]. So, we further assume that sending and receiving the whole M bytes can fully overlap with negligible overhead. We have the following result.

[6] While we find that 4MB partition size works reasonably well in our environment, BytePS allow users to tune the partition size value.

Theorem 1. *The SS workload assignment given by Eq. 1 and Eq. 2 is optimal for minimizing communication time.*

Proof. We first consider the network traffic of a GPU machine. It runs a CS module and an SS module. CS should send and receive M bytes in total. However, when it communicates with the SS on the same GPU machine, the traffic does not go over the network. So, a CS module will send and receive $M - M_{SS_{GPU}}$ bytes. An SS module on a GPU machine must receive and send $M_{SS_{GPU}}$ from other $n-1$ GPU machines, *i.e.,* $(n-1)M_{SS_{GPU}}$ in total. Adding them together, a GPU machine with network bandwidth B requires communication time t_g:

$$t_g = \frac{M + (n-2)M_{SS_{GPU}}}{B} \tag{3}$$

Similarly, if $k > 0$, we can get that a CPU machine with network bandwidth B requires communication time t_c:

$$t_c = M_{SS_{CPU}}/B \tag{4}$$

In addition, the sum of all the SS workload should be equal to the total model size.

$$M = kM_{SS_{CPU}} + nM_{SS_{GPU}} \tag{5}$$

From Eq. 5, it is clear that the larger $M_{SS_{CPU}}$ is, the smaller $M_{SS_{GPU}}$ is. Consequently, when $n \geq 2$, the larger t_c is, the smaller t_g is (or t_g is unchanged if $n = 2$). In addition, we know that the final communication time is $\max(t_c, t_g)$.

To minimize the communication time, t_c and t_g need to be equal. If they are not equal, say $t_c > t_g$, it means the communication time can be further reduced by decreasing $M_{SS_{CPU}}$ and thus bring down t_c.

We let $t_c = t_g$ and combine Eq. 3, Eq. 4, and Eq. 5. Solving the equations with $M_{SS_{GPU}}$ and $M_{SS_{CPU}}$ as variables, we can get the optimal values as given by Eq. 1 and Eq. 2. □

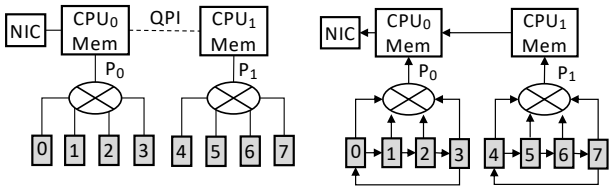
Based on Theorem 1, combine Eq. 3 and Eq. 2, we have the optimal communication time, which is used in Fig. 12.

$$t_{opt} = \frac{2n(n-1)M}{(n^2 + kn - 2k)B} \tag{6}$$

From Eq. 2, we can see that when the numbers of CPU machines and GPU machines are the same, $M_{SS_{GPU}} = 0$, which means that we do not need any SS_{GPU}. This is because the CPU machines already provide enough aggregate bandwidth. BytePS falls back to non-colcated PS. Similarly, when the number of CPU machines is 0, BytePS falls back to all-reduce and colocated PS.

Of course, the more interesting case is the general case when $0 < k < n$. We use the communication time of the plain all-reduce and non-colocated PS as the two baselines. We define the acceleration ratio γ_a as the communication time of the plain all-reduce divided by that of the general case. Similary, γ_p is defined as the acceleration ratio compared to the non-colocated PS case. We have

$$\gamma_a = \frac{n^2 + kn - 2k}{n^2}, \gamma_p = \frac{n^2 + kn - 2k}{2k(n-1)} \tag{7}$$



(a) PCIe-only topology (b) Outgoing data flow

Figure 6: PCIe-only machine topology and BytePS data flow. Gray boxes are GPUs. Only the outgoing direction (from GPUs to network) is shown in the data flow figure. Incoming is the opposite.

When $k = n$ and $n \to \infty$, $\gamma_a = 2$. When k is small, γ_p can be quite big, as the communication bandwidth is severely bottlenecked by the CPU machines in non-colocated PS. For example, when $n = 32$ and $k = 16$, we have $\gamma_a = 1.46$ and $\gamma_p = 1.52$, respectively. It means that BytePS can theoretically outperform all-reduce and PS by 46% and 52%, respectively.

We note that adding more CPU machines beyond $k = n$ does not help, since the communication bottleneck will become the NIC bandwidth of the GPU machines.

4.2 Intra-machine Communication

In §4.1, we design the optimal inter-machine communication strategy. In practice, we find that intra-machine communication is equally important. There are often multiple GPUs in a machine. CS must aggregate/broadcast the tensors before/after communicating with SS. This can create congestion on the PCIe links and prevent NIC from fully utilizing its bandwidth B. Moreover, the GPU machine's internal topology can be diverse in data centers. Below, we share the two most common machine setups in our environment and our corresponding solution. We present several principles that can apply to other machine setups in §4.2.3.

4.2.1 PCIe-only Topology

Fig. 6(a) shows a setup in our production environment. A GPU machine has two NUMA CPUs connected via QPI. The eight GPUs are split into two groups and connected to two PCIe switches, respectively. The NIC is 100Gbps and connected to the PCIe of one of the CPUs. All PCIe links in figure are 3.0 x16 (128Gbps theoretical bandwidth). The CPU memory and QPI has $> 300Gbps$ bandwidth, which are less likely the communication bottleneck. We call this *PCIe-only topology*. For this machine model, we measure that the throughput of GPU-to-GPU memory copy is ≈105Gbps within the same PCIe switch. The throughput of GPU-to-GPU memory copy across PCIe switches, however, is only ≈80Gbps.

Unfortunately, many existing training frameworks ignore such details of internal topology. For example, TensorFlow PS, MXNet PS and even the "hierarchical all-reduce" mode of Horovod use a straightforward reduce or reduce-scatter across *all* GPUs on the same machine. This would lead to cross-PCIe switch memory copy, which is unfortunately slower.

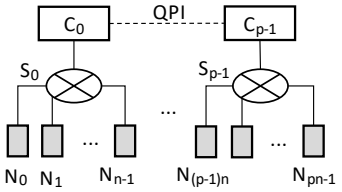


Figure 7: Notations of the PCIe-only topology.

In contrast, BytePS lets GPUs under the same PCIe switch sum the tensors first, then copy to CPU and let CPU do the global summation, and finally broadcast back the global sum. We call it *CPU-assisted aggregation.* Specifically, it consists of the following steps.

1. **Reduce-Scatter**: Suppose each PCIe switch has l GPUs. These l GPUs perform a reduce-scatter which incurs $(l-1)M/l$ traffic only inside the PCIe switch. When it finishes, each GPU should hold M/l aggregated data.
2. **GPU-CPU Copy**: Each GPU copies its M/l data to CPU memory, which incurs M/l traffic along the route. Every PCIe switch would generate M aggregated data.
3. **CPU-Reduce**: CPU reduces the data from all PCIe switches and generates the aggregated data across all GPUs. This reduction does not incur any PCIe traffic.
4. **Networking**: CS sends the data to SS and receives globally aggregated data from SS.
5. **CPU-GPU Copy**: Each GPU copies its M/l partition from CPU memory back to itself. This incurs M/l traffic from the CPU to each GPU.
6. **All-Gather**: Each GPU performs an all-gather operation with those that are under the same PCIe switch. This incurs $(l-1)M/l$ traffic inside the switch.

Fig. 6(b) shows the traffic of step 1 to 3. Step 4 to 6 use the same links but the opposite direction. With CPU-assisted aggregation, the PCIe switch to CPU link would carry only M traffic in each direction, much lower than doing collective operation directly on eight GPUs ($7M/4$ traffic). Meanwhile, the traffic on each PCIe switch to GPU link would be $(2l-1)M/l$. Let $l=4$ (each PCIe has four GPUs), this is $7M/4$, remaining the same as the existing approach. Fundamentally, BytePS leverages the spare CPUs on the GPU machine to avoid the slow GPU-to-GPU cross-PCIe switch memory copy.

Optimality Analysis. We now analyze the communication optimality of the above strategy. Fig. 7 shows a more generic PCIe-only topology with variable number of GPUs and PCIe switches. We do not plot the NIC as in Fig. 6(a) because under that topology, the NIC has dedicated PCIe lanes and will not compete for the PCIe bandwidth with GPUs. The system architecture is modeled as a hierarchical graph $G=(V,E)$. Denote N as the set of leaf nodes (GPUs), S as the set of intermediate nodes (switches), C as the set of CPU nodes. $V=N\cup S\cup C$. Each edge $e(v_x,v_y)$ in E represents the bandwidth from vertex v_x to v_y, and we denote $t(v_x,v_y)$ as the amount of traffic sent from v_x to v_y. We further define p as the number of switches ($p\geq 2$), and n as the leaf nodes that each switch connects ($n\geq 2$).

We assume the following features of G: (1) Each edge in E is duplex and the bandwidth of both directions are equal. Denote $b(v_x,v_y)$ as the bandwidth of $e(v_x,v_y)$, then $b(v_x,v_y)=b(v_y,v_x)$; (2) We assume G is symmetric. The bandwidth at the same layer of the tree is equivalent. For example, $b(S_j,C_j)=b(S_k,C_k)$ and $b(N_x,S_j)=b(N_y,S_j)$ hold for any $j,k\in[0,p-1]$, $x,y\in[jn,(j+1)n-1]$; (3) The memory and QPI bandwidth is much higher than the PCIe links and is less likely to be the bottleneck. In the following, we only focus on the PCIe links.

The GPUs from N_0 to N_{pn-1} need to sum their data. We can either use *CPU-assisted aggregation* mentioned before, or use *brute-force copy* that needs each GPU to copy its entire data to C directly. In practice, the optimal solution should be a combination of these two strategies, depending on the value of $b(S_j,Cj)$ and $b(N_i,S_j)$. The intuition is that we apply brute-force copy on x of the data, and CPU-assisted aggregation on y of the data ($x+y=1$). Under certain x and y, the job completion time J can be minimized. We calculate the traffic of two links respectively. On $e(S_j,C_j)$, the traffic is composed of n times brute-force copy plus the traffic of CPU-assisted aggregation. On $e(N_i,C_j)$, the traffic is composed of one brute-force copy and the complete traffic of CPU-assisted aggregation.

$$t(S_j,C_j)=n*xM+\frac{yM}{n}*n=(nx+y)M \tag{8}$$

$$t(N_i,S_j)=xM+(\frac{2(n-1)}{n}+\frac{1}{n})yM=(\frac{2n-1}{n}y+x)M \tag{9}$$

Since J is determined by $J=max(\frac{t(N_i,S_j)}{b(N_i,S_j)},\frac{t(S_j,C_j)}{b(S_j,C_j)})$, the optimal J is highly related to the two bandwidth terms. On our own PCIe machines (Fig. 6(a)), we measure that both $b(N_i,S_j)$ and $b(S_j,C_j)$ are 13.1GB/s (105Gbps). Let M=1GB and $n=4$, combining Equation (8), (9) and $x+y=1$, we are trying to find a $x\in[0,1]$ such that $\arg\min_x J(x)=max(\frac{3x+1}{13.1},\frac{7-3x}{52.4})$. Solve it and we will get the optimal solution is $x^*=1/5$ and $J^*=0.129s$. This means the optimal solution works like this: each GPU applies brute-force copy on its $1/5$ data, and uses CPU-assisted aggregation for the rest $4/5$ data. Therefore, we have the following key conclusions:

CPU-assisted aggregation is near-optimal. When $x=0$, the solution is our CPU-assisted aggregation, and the job completion time is $J(0)=0.141s$. As calculated, the optimal time is $0.129s$. Thus, our strategy closely approximates the optimal solution, with 9% difference on performance. However, in practice, brute-force copy heavily stresses the CPU memory – any tensor that uses brute-force copy would consume $4\times$ CPU memory bandwidth compared with CPU-assisted aggregation. CPU memory does not really have $4\times$ bandwidth of PCIe links, especially for FP16 summation (Fig. 9(b). Consequently, we choose not to use brute-force copy at all and stick

to CPU-assisted aggregation.

CPU-assisted aggregation is better than ring-based all-reduce. We have the job completion time for ring-based all-reduce as $J_{ar} = \frac{2(np-1)M}{np*b_{bottleneck}}$. Similarly, for CPU-assisted aggregation we have $J_{ca} = \frac{M}{b(S_j,C_j)} * max(1, \frac{2n-1}{kn})$, where $k = \frac{b(N_i,S_j)}{b(S_j,C_j)}$. In our case, $k = 1$ and $b_{bottleneck} < b(S_j,C_j)$, so it is easy prove that $J_{ca} < J_{ar}$ always holds for any $n, p \geq 2$. For example, using the value from our PCIe machines, let $p = 2$, $n = 4$, $b_{bottleneck} = 80Gbps$ (bandwidth of memory copy that crosses PCIe switches) and $b(S_j,C_j) = 105Gbps$ we get that J_{ca} is 23.7% smaller than J_{ar}.

4.2.2 NVLink-based Topology

Fig. 8(a) shows the other machine model in our data center – a GPU machine with NVLinks. There are four PCIe switches, each connecting two GPU cards. The GPUs are also connected via NVLinks. The NVLinks give every GPU in total 1.2Tbps GPU-GPU bandwidth, much higher than the PCIe link. The NIC is connected to one of the PCIe switches.

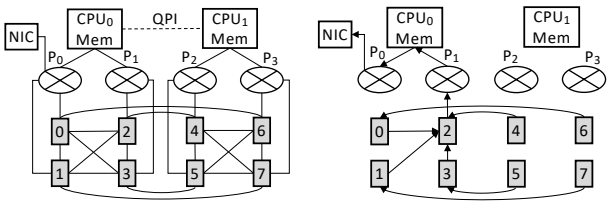
With NVLink, GPU-to-GPU communication can completely avoid consuming PCIe bandwidth. So, we no longer need CPU-assisted aggregation. However, we find that existing framework, including the most popular GPU all-reduce implementation NCCL (used by TensorFlow, PyTorch, MXNet and Horovod), is again sub-optimal.

The problem is that the topology is not symmetric considering the NIC, which is connected to only one (out of four) PCIe switch. The NIC and the two GPUs under the same PCIe switch have to compete for the PCIe bandwidth of $P_0 - CPU_0$. Remember that not only CS uses this PCIe bandwidth, but also the SS runs on this same GPU machine uses it! $P_0 - CPU_0$ again becomes the bottleneck in the whole communication.

Based on the analysis, we should leave as much $P_0 - CPU_0$ PCIe bandwidth as possible to the NIC during local aggregation. For this topology, BytePS uses reduce and broadcast instead of reduce-scatter and all-gather – tensors from all GPUs are first reduced to GPU_2 and the result is then copied to CPU_0 memory from GPU_2. Fig. 8(b) shows those steps. Later, when CS gets the aggregated results from SS, GPU_2 would copy the data into GPU memory and broadcast them to other GPUs. This way, we completely prevent GPUs from using the $P_0 - CPU_0$ bandwidth for communication, so the NIC can run to full 100Gbps bandwidth.

This approach seems to create traffic hotspots on GPU_2. However, NVLinks has much larger bandwidth than PCIe links, so inter-GPU communication is never the bottleneck even on the hotspots. Meanwhile, the $P_1 - CPU_0$ PCIe link used for GPU-CPU copy has approximately the same 100Gbps bandwidth as the NIC, so it is not a bottleneck either.

BytePS has achieved the optimal result – there is no intra-machine bandwidth bottleneck. Existing solutions like NCCL, unfortunately, tends to let GPUs use the $P_0 - CPU_0$ bottleneck link because of the proximity between GPU_0 and the NIC.



(a) NVLink-based topology (b) Outgoing data flow

Figure 8: NVLink-based machine topology and BytePS data flow. Only the outgoing direction is shown in the data flow figure.

Consequently, its communication performance is lower than our solution in the NVLink-based machines.

4.2.3 Discussion

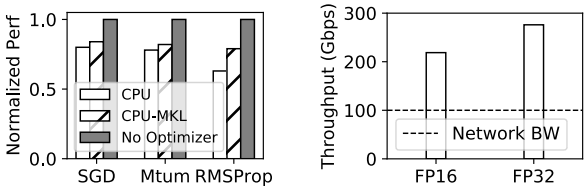
The solutions for PCIe-only and NVLink-based topology are quite different. This shows that there is no one-fit-all optimal solution. The intra-matchine communication must adapt to different internal topologies. Admittedly, there are certainly more topologies than the above two used in our environment. However, we believe that the above two are representative, since they are similar to the reference design recommended by server vendors [15] and NVIDIA [11], respectively.

Despite the difference, we summarize two principles – 1) always avoid direct GPU-to-GPU memory copy when the two GPUs are not under the same PCIe switch because it is slow in practice. 2) Always minimize traffic on the PCIe switch to CPU link that is shared by GPUs and NIC. We propose the following best practice procedure. Let S_n be the number of PCIe switches with GPUs and NIC, and S_g be the number of PCIe switches with only GPUs.

1. If $S_n > 0$ and $S_g > 0$, the topology is asymmetric like our NVLink-based topology. CS should use reduce and broadcast, with GPUs that are not competing with NICs as reduce or broadcast roots.
2. If $S_n = 0$ or $S_g = 0$, the topology is symmetric like our PCIe-only case. CS should use reduce-scatter and all-gather to balance traffic on all PCIe switches. CPU-assisted aggregation (§4.2.1) should be used if no NVLink.

Multi-NIC topology. Although the two specific topologies we discussed have only one NIC, the above principles can directly extend to multi-NIC topology – it only changes the value of S_n and S_g.

GPU-direct RDMA (GDR). GDR can potentially reduce the PCIe traffic. However, GDR requires the GPU and the RDMA NIC to be on the same PCIe switch, otherwise the throughput can be less than 50Gbps even with 100GbE NIC [12], which is also confirmed by our own measurements. Consequently, GDR does not benefit our settings – PCIe-only topology does not satisfy the requirement, and we already avoided any PCIe bottlenecks for NVLink-based topology. In addition, most clouds like AWS do not support GDR. Therefore, BytePS does not use GDR for now.



(a) Parameter update on different devices. (Mtum: Momentum [65])

(b) Throughput of CPU summation on different floating point tensors.

Figure 9: CPU is slow for optimizers but not for summation.

We can see that the optimal intra-machine communication strategy is tightly coupled with the internal topology. Building a profiler to automatically detect the topology, probe the bandwidth, and generate the best strategy is interesting future work.

5 Summation Service

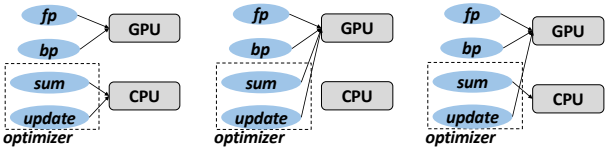
To get the optimal inter-machine communication time (§4.1), BytePS needs a module that can run on the CPU of every machine and communicate with CS. The question is, what is its role in the training algorithm? Our initial attempt was to follow the previous PS design [44], in which the PS processes are responsible for running the *optimizer*. The optimizer aggregates the gradients from all GPUs and updates the DNN model parameters using various optimizers.

The CPU bottleneck. Unfortunately, soon we found that the CPUs became a bottleneck in the system. We use an experiment to demonstrate this. We train the VGG16 DNN [63] using a typical non-colocated PS setting: using one Tesla V100 GPU machine and one CPU machine (Intel Xeon Platinum CPU, 32 cores with hyper-threading and Intel MKL [7]) connected by 100GbE Ethernet. The GPU machine runs the forward and backward propagation, and the CPU machine runs the optimizer using all the 32 CPU cores.

Fig. 9(a) shows that, even with 32 cores and MKL-enabled, running the optimizer on the CPU machine can slow down the end-to-end training speed. It means the CPU cannot match the network bandwidth and becomes a bottleneck (§6). As the optimizer algorithm gets more complicated (from simpler SGD to the more complicated RMSProp), the bottleneck effect becomes more severe.

The root cause. The CPU bottleneck is caused by the limited memory bandwidth. Popular optimizers such as Adam can easily exhaust the memory bandwidth of modern CPUs. For example, the peak transfer rate of a 6-channel DDR4-2666 memory setup is up to 1024 Gbps combining read and write [8]. It is easy to estimate that, for example, the Adam optimizer [42] requires more than 10x memory access (read+write) for applying every gradient update. Adding that 100Gbps NIC consumes 200 Gbps memory bandwidth (read+write), the 1024 Gbps memory bandwidth is simply not sufficient for Adam to process 100 Gbps gradient stream.

CPU is good at summation. The above experiment leads us to rethink the tasks placed on CPUs.The computation of an optimizer can be divided into two steps, *gradient summation* and *parameter update*, as Fig. 10 shows.



(a) PS (b) All-reduce (c) BytePS

Figure 10: Component placement comparison between all-reduce, PS and BytePS.

Fortunately, modern x86 CPUs are good at summation thanks to the highly optimized AVX instructions [47]. In Fig. 9(b), we show the summation throughput on the same CPUs as above, using synthetic floating point tensors. The throughput is more than 200Gbps for both FP16 and FP32 precision, higher than the 100Gbps NIC bandwidth. Therefore, summation on CPU will not be a bottleneck.

BytePS's solution. Based on these observations, BytePS decouples the two steps of optimizer. We move the computation-intensive parameter update to GPUs and places only summation on CPUs – this is why we name the CPU module Summation Service (SS). SS not only prevents the CPU from being the bottleneck, but also largely reduces the CPU overhead. With carefully implementation using AVX and OpenMP, SS only consumes fewer than 3 CPU cores when it runs at 100Gbps throughput. Fig. 10 gives a high-level comparison over PS, all-reduce and BytePS on how they place different components in DNN training onto GPU and CPU resources.

Since Summation Service moves parameter update to GPU machines, all the GPU machines need to perform the same parameter update calculation, whereas parameter update needs to be done only once in traditional PS. BytePS hence uses more computation cycles for parameter update than PS. This is a tradeoff we made willingly, to accelerate end-to-end training speed. We define SS overhead ratio as the FLOPs for parameter update over the sum of FP and BP FLOPS. The ratio is 138 MFLOPs / 32 GFLOPs, 26 MFLOPs / 7.8 GFLOPs, 387 MFLOPs / 494 GFLOPs for VGG-16, ResNet-50, BERT-large using SGD as the optimizer, all are less than 0.5%. The introduced overhead is negligible, compared to the training speedup (Fig. 9(a)). The above ratio definition assumes batch size of 1. DNN training typically uses batch size of tens or hundreds. Parameter update is done once per batch, hence the additional overhead is even smaller in practice.

We note that Horovod [60] has the option to move gradient aggregation to CPUs by first copying the tensors to CPU memory and then performing CPU-only all-reduce. Since it still only relies on the CPUs and bandwidth on GPU machines, it does not provide communication-wise advantages compared with directly all-reduce on GPUs. BytePS is different: it leverages *additional* CPU machines for gradient summation, while keeps parameter update on GPUs.

Support asynchronous training. Although separating the

summation and update brings us performance benefits, it breaks an important feature of the original PS: the support of asynchronous training like Asynchronous Parallel [25]. Asynchronous Parallel relies on the PS processes keeping the most updated model parameters, which is not directly compatible with the design of SS. To bridge this gap, we re-design a new workflow that can enable asynchronous training with SS, as shown in Fig. 11(b). In short, GPU updates parameters and computes the *delta parameters* first. CS sends them and receives *latest parameters*. SS keeps adding delta parameters to the latest parameters. Next, we prove that this new training workflow is equivalent to Asynchronous Parallel in terms of algorithm convergence.

Theorem 2. *The asynchronous algorithm for BytePS is equivalent to Asynchronous Parallel [25].*

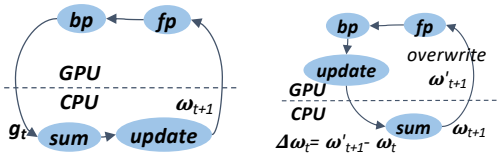
Proof. Consider one SS connected with n CSs. We say a CS stores the local model parameters, and a SS holds the latest version of parameters. The high level idea of our proof is to show that our algorithm generates identical state (*i.e.*, same parameter for the SS module and n CS modules) with Asynchronous Parallel, given the same communication order (push and pull order). We use f as a general representation of the optimizer. The optimizations thus can be represented as $w \leftarrow w + f(g_{i,t})$, where $g_{i,t}$ represents the gradients of CS_i ($i \in [0, n-1]$) at iteration t ($t \in [1, T]$). Denote w_{ps} and w_{byteps} as the parameter in PS and BytePS, respectively. And denote $w_{i,t}$ as the parameter on each $worker_i$ (for PS) or CS (for BytePS) at iteration t. The parameter is initiated to w_0 for all CSs and the SS. After T iterations, we can obtain the updated parameter as:

$$w_{ps} = w_0 + \sum_{t=1}^{T} \sum_{i=0}^{n-1} f(g_{i,t}) \tag{10}$$

$$w_{byteps} = w_0 + \sum_{t=1}^{T} \sum_{i=0}^{n-1} \Delta w_{i,t} \tag{11}$$

Next, we use induction to prove that $\Delta w_{i,t} = f(g_{i,t})$ holds for any i and t. (1) *Base case* $t = 1$: Given initial parameter w_0, we obtain the gradient $g_{i,1}$ from w_0. In Parameter Server, $worker_i$ pushes $g_{i,1}$ to the server and get updated as $w_{ps,1} = w_0 + f(g_{i,1})$. In BytePS, CS_i pushes $f(g_{i,1})$ to SS and get updated as $w_{byteps,1} = w_0 + f(g_{i,1})$. So $\Delta w_{i,t} = f(g_{i,t})$ holds for $t = 1$. Meanwhile, the parameter on $worker_i$ or CS_i is the same on both architectures after receiving the response from the server or SS. (2) *Inductive step*: If the lemma we want to prove holds for $t = k (k \geq 1)$, the gradient $g_{i,k+1}$ is computed from the same w_k. Similar to the base case, we obtain $w_{ps,k+1} = w_k + f(g_{i,k+1})$ and $w_{byteps,k+1} = w_k + f(g_{i,k+1})$. So $\Delta w_{i,t} = f(g_{i,t})$ holds for $t = k+1$. By the principle of induction, $\Delta w_{i,t} = f(g_{i,t})$ holds for all $t \in \mathbb{N}$.

Return to (10) and (11). Since $\Delta w_{i,t} = f(g_{i,t})$ holds for any i and t, we get $w_{ps} = w_{byteps}$. This completes the proof because the parameter of our algorithm and Asynchronous Parallel are equivalent after any T batches. □



(a) PS-async (b) BytePS-async

Figure 11: Asynchronous training workflow comparison between PS and BytePS. g is the gradients. ω is the parameters.

6 Implementation

While the core of BytePS is generic for any training framework, BytePS also implements plugins for TensorFlow, PyTorch and MXNet, for user-friendliness. The core is implemented in C++, while the framework plugins contain both C++ and Python. In total, BytePS consists of about 7.8K lines of Python code, and 10K lines of C++ code. As a major online service provider, we have deployed BytePS internally. BytePS has also been open-sourced [4] and attracted thousands of users.

6.1 Multi-Stage Pipeline

A common way to speed up a multi-step procedure is to build a multi-stage pipeline that overlaps the processing time of each step. We incorporated the idea of tensor partition and pipelining from prior work [34, 55]. For example, for PCIe-only topology, CS has six steps. It maps to a 6-stage pipeline in BytePS runtime. We implement BytePS to be flexible in constructing the pipeline without recompiling. Each stage in the pipeline is implemented as an independent thread with a priority queue of tensors. The priority is assigned similar to [34,55]. As analyzed in §4.1.1, large tensors are partitioned to multiple smaller tensors no more than 4MB. Next, each small tensor is enqueued to the first queue and moves towards the next queue once a stage finishes processing it, until it is dequeued from the last one.

6.2 Address RDMA Performance Issues

For inter-machine communication, we use RDMA RoCEv2. Each machine has one 100GbE NIC, and the RDMA network provides full bisection bandwidth. To get the full benefit of RDMA, we have gone through a full design and debug journey which we share as follows.

RDMA Memory Management. To improve the performance, we aim to avoid unnecessary memory copies [72] and achieve zero-copy on CPU memory. BytePS is based on RDMA WRITE because it is the most performant among common RDMA verbs [39]. Conventional one-sided RDMA operations (WRITE and READ) require at least two round-trips: getting the remote address, and writing (reading) the value to (from) that address [39, 40, 50, 70]. We optimize the process by leveraging the fact that DNN training always sends the same set of tensors in every iteration. Only at the

Table 2: BytePS throughput with a pair of CPU machine and GPU machine running microbenchmark.

Solution	baseline	+shm	+shm +aligned	all
Throughput in Gbps	41	52 (1.27x)	76 (1.85x)	89 (2.17x)

first iteration, BytePS initializes all the required tensors, register the buffer with RDMA NIC and exchange all the remote addresses. Then BytePS stores the remote buffer information and reuse it directly in the rest iterations.

Address Slow Receiver Symptom. We also run into the slow receiver symptom as reported in [30] – the NICs are sending out many PFCs into the network. Those excessive PFCs slow down tensor transmission can cause collateral damage to other traffic. Here we report several additional causes of such symptom and how we address them.

Our first finding is that internal RDMA loopback traffic can cause internal incast, and push the NIC to generate PFC. BytePS runs both CS and SS on each GPU machine. The traffic between them, which we call loopback traffic, does not consume NIC's external Ethernet bandwidth, but does consume internal CPU-NIC PCIe bandwidth. Initially, we did not add any special design – we stuck to RDMA verbs [9] for loopback traffic and thought the NIC DMA can handle it. However, we realize that it creates a 2:1 incast on the NIC, with RX and loopback as two ingress ports and the DMA to memory engine as one egress port!

To solve it, we implement a shared memory (*shm*) data path. When CS detects that SS is on the same machine as itself, CS simply notifies SS that the data is in shared memory. After SS finishes summation, SS copies the results from its own buffer back to CS's shared memory. Consequently, the loopback RDMA traffic is eliminated.

Our Second finding is that we need to use *page-aligned* memory for RDMA. Otherwise PFCs may be triggered. Our hypothesis is that hardware DMA aligns the transfer unit to the page size (*e.g.,* 4096 bytes). Therefore, using a page-aligned address is more friendly to DMA engine as it reduces the number of pages needed to be written.

Our third finding is that the RDMA NIC RX performance can be impacted by how the concurrent send is implemented! In the end, we not only use page-aligned memory, but also enforce only one scatter-gather entry (*sge*) per RDMA WRITE on the sender side.[7]

After all the optimization, BytePS implementation can run as expected. Table 2 shows the performance improvement after each of the above three optimizations is applied. The NIC generates negligible PFCs.

As we have discussed in §4.1, BytePS creates many many-to-one communication patterns in the network. Many-to-one is well-known for creating incast and packet loss in TCP/IP network [66]. But BytePS uses RDMA/RoCEv2 which depends on a lossless fabric and DCQCN [75] for congestion control. We do not observe incast issue in BytePS.

[7] In the whole process, we contacted with the NIC vendor and had lengthy discussion with their software and hardware experts. As of writing, we have not got the official root cause of the last two problems.

6.3 BytePS Usage

BytePS [4] is easy to use. We provide Python interfaces that are almost identical to Horovod, PyTorch native API and TensorFlow native API. Users can choose either of them and migrate to BytePS with minimal efforts. For example, for a Horovod-MNIST example [19], we only need to change one line of Python code, from *"import horovod"* to *"import byteps"*. In fact, we are able to convert most of our internal Horovod-based training tasks to BytePS automatically.

7 Evaluation

In this section, we show that BytePS not only achieves optimal communication performance in microbenchmarks, but also significantly accelerate training jobs in production environment. We list a few highlights regarding the high fidelity of the results.

- All resources used are allocated by the scheduler of production clusters. The scheduler uses non-preemptive resource scheduling – once a training job is scheduled, it will have a fixed number of CPU machines that will not change. Even the most large-scale tasks we show use $< 5\%$ GPUs of a cluster that runs many production tasks.
- We use large training batch sizes. Smaller batch sizes mean less GPU memory consumption but more communication, so the end-to-end improvement will be more evident. However, all our tasks use almost full GPU memory, so the speedup numbers against all-reduce and PS are the *lower bound* of BytePS.
- Although we cannot disclose any specific models that are used internally, the tasks and DNN model structures shown are highly representative of production workloads. The code is also available publicly for reproducibility [5].
- We compare BytePS with the state-of-the-art PS and all-reduce implementation without modification. For example, we do not apply the RDMA optimizations mentioned in §6.2 on native-PS and all-reduce.

The cluster we use has a RoCEv2 network with full bisection bandwidth. All the machines have one 100GbE NIC. We note that TensorFlow, PyTorch and MXNet can overlap the DNN computation and communication [34, 55], thus even a small improvement in end-to-end performance can indicate a large improvement in communication.

7.1 Inter-machine Microbenchmarks

First, we use microbenchmarks to show the pure inter-machine communication performance of different architectures. We allocate eight 1-GPU machines from the cluster scheduler. We run a dummy task in which all GPU workers

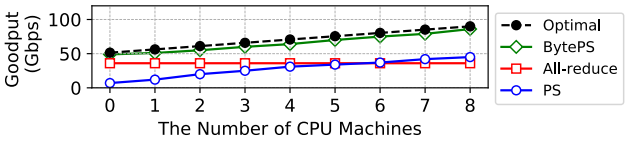
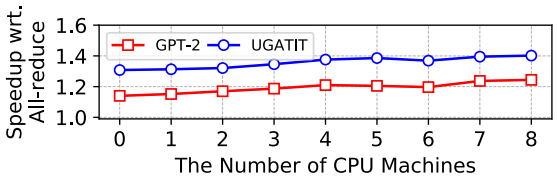
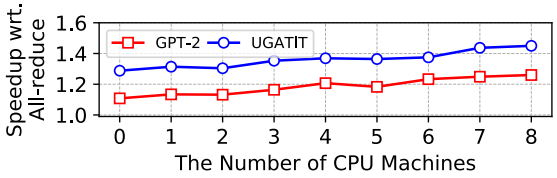


Figure 12: Communication goodput of 8× 1-GPU machines with varying number of additional CPU machines. The point-to-point RDMA goodput is ≈ 90*Gbps* in our network, so we plot the "optimal" line based on $B = 90Gbps$ and the analysis in §4.1.



(a) PCIe-only GPU machines



(b) NVLink-based GPU machines

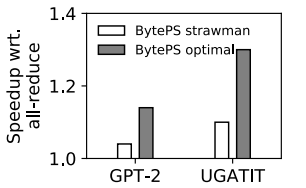
Figure 13: End-to-end performance with different number of CPU machines. The training is run with PyTorch on 8 GPU machines each with 8 GPUs. Each CPU machine uses < 4 cores.

just keep reducing large tensors on GPU and record the communication goodput. We verify that no other distributed job is placed on the same physical machines.

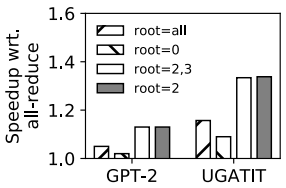
Fig. 12 shows that BytePS performance is very close to the theoretical optimum (§4.1), with 1-9% difference for different number of CPU machines. All-reduce, as expected, is close to the optimal only if there is no additional CPU machine, while remain the same even if there are CPU machines. The (MXNet) PS does not run optimizer in this case, but is mainly bottlenecked by issues described in §6.2. In practice, if PS runs DNN optimizer algorithms, the performance will be worse than all-reduce even with $k = n$ CPU machines (Fig. 4). In contrast, because of the Summation Service design, BytePS would not be affected in real training tasks shown below.

7.2 Leverage CPU Machines

Next, we show that BytePS can indeed leverage different numbers of CPU machines to speed up training. In Fig. 13, we use 8 GPU machines, each with 8 Tesla V100 32GB GPUs, and is either PCIe-only or NVLink-based topology. We vary the number of CPU machines from 0 to 8. We compare BytePS end-to-end training performance against state-of-the-art all-reduce implementation (Horovod 0.19 and NCCL 2.5.7) as the baseline. We test two DNN models, UGATIT GAN [41] (one of the most popular models for image generation) and GPT-2 [57] (one of the most popular NLP models for text generation), both implemented in PyTorch. The per GPU batch



(a) PCIe-only GPU machines



(b) NVLink-based GPU machines

Figure 14: Topology-aware intra-machine communication. The training is run with PyTorch on 8 GPU machines each with 8 GPUs and no additional CPU machine.

size is 2 images for UGATIT, and 80 tokens for GPT-2. We will evaluate more models, frameworks and machines in §7.4.

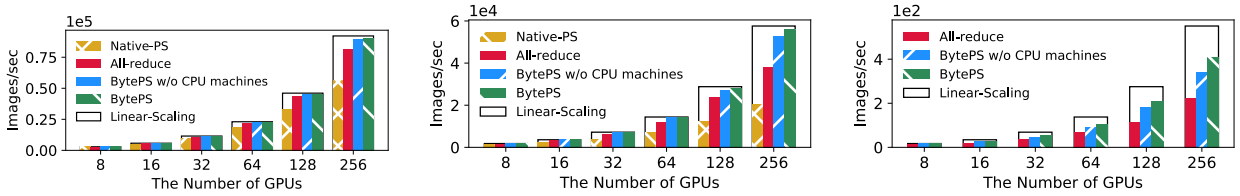
Fig. 13 shows that, with more CPU machines, BytePS can run faster – up to 20% than without CPU machines. The SS on each CPU machine only consumes no more than 4 CPU cores. It is usually easy for our scheduler to find sufficient CPUs that are on machines running non-distributed jobs. It is free (or << 10% costs compared with the expensive GPUs) speedup for the cluster. Compared with all-reduce, BytePS is consistently faster in any cases and can be up to 45% faster in the best case. On NVLink-based GPU machines, the speedup is higher because the communication bottleneck is more on the network instead of PCIe links. Finally, models have different speedup due to different model sizes and FLOPs. In the examples we show, GAN is more communication intensive, so the end-to-end gain of BytePS is larger.

7.3 Adapt to Intra-machine Topology

Next, we show the benefits of BytePS intra-machine communication strategy. The software and hardware configurations are the same as in §7.2. To better compare with the all-reduce baseline, we run the jobs without any CPU machines. Thus, BytePS does not take any advantages explained in §7.2. For PCIe-only GPU machines (Fig. 14(a)), we run BytePS with 1) strawman strategy, the same as common all-reduce or PS and 2) the optimal solution in §5. We see that the optimal intra-machine solution has up to 20% gain as well.

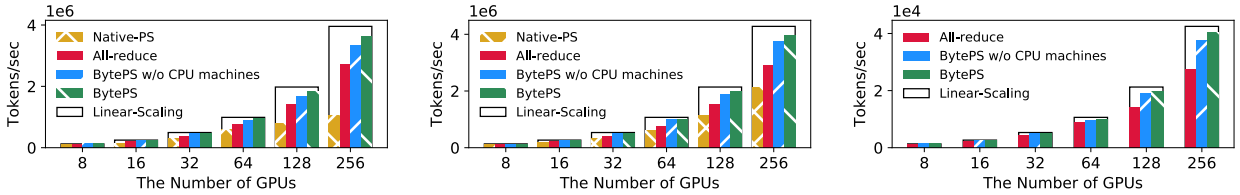
For NVLink-based GPU machines (Fig. 14(b)), we use different sets of GPUs as the local reduce roots. BytePS's optimal solution, as explained in §4.2.2, is $root = 2$. $root = 2,3$ means CS chooses GPU 2 and 3 as the reduce root in a round robin manner. It has almost the same performance because GPU 3 is not competing for PCIe bandwidth with the NIC, either. It is an alternatively optimal solution. However, $root = all$ has poorer performance. Communication-wise, it is equivalent to Horovod's hierarchical mode. $root = 0$ is the worst because it competes hardest with the NIC. Unfortunately, it is equivalent to Horovod's normal mode (plain NCCL all-reduce).

One thing to note is that even without any optimization, BytePS still outperforms all-reduce. We discuss this in §8.



(a) TensorFlow, ResNet-50, batch=256 images (b) MXNet, VGG-16, batch=96 images (c) PyTorch, UGATIT, batch=2 images

Figure 15: Computer Vision models. The batch sizes are per GPU.



(a) TensorFlow, Transformer, batch=3072 tokens (b) MXNet, BERT-Large, batch=8192 tokens (c) PyTorch, GPT-2, batch=80 tokens

Figure 16: NLP models. The batch sizes are per GPU.

7.4 Scalability

To demonstrate BytePS's performance at different scales, we run six different training jobs using 8 to 256 Tesla V100 32GB GPUs, *i.e.,* 1 GPU machine to 32 GPU machines. Due to the constraint of free resources, we only use NVLink-based GPU machines. The six different jobs cover three frameworks, TensorFlow, PyTorch and MXNet. We have introduced two of the models, UGATIT and GPT-2 in §7.2. The rest four models are ResNet-50 [32], VGG-16 [63] (two of the most popular models for image classification and extraction), Transformer [67] (one of the most popular models for machine translation) and BERT [26] (one of the most popular models for natural language understanding). We take the official implementation of these models and slightly modify them (no more than 20 lines of code) to use PS, all-reduce and BytePS, respectively.

For BytePS, we evaluate its performance with and without CPU machines. When there are CPU machines, the number of CPU machines is equal to GPU machines. For all-reduce, we use Horovod with NCCL for all cases. For PS, we show the native implementation from TensorFlow and MXNet with RDMA support enabled. PS uses the same resources as BytePS with CPU machines. PyTorch does not have official PS implementation, so it does not have PS results. We also provide the speed of linear scaling as the upper bound. We use trained images per second as the speed metric for computer vision models, and tokens per second for NLP models.

Fig. 15 and Fig. 16 show very consistent results – BytePS with CPU machines is always the best and BytePS without CPU machines is the second. The native PS of both TensorFlow or MXNet are always the poorest. All-reduce always has a clear advantage over PS, but is inferior to BytePS. When training with 256 GPUs, the speedup of BytePS over all-reduce is 10% to 84% with CPU machines, and 9% to 53% without CPU machines. From 8 GPUs to 256 GPUs, the speedup becomes larger. We expect that with even more GPUs, BytePS will have even larger advantage.

We see that models have different system scalability,[8] which is determined by the model sizes and FLOPs. The most scalable model is ResNet-50. BytePS achieves 97.5% scaling efficiency with 256 GPUs. All-reduce also performs well, achieving 88% scaling efficiency. It is not surprising that prior work is fond of training ResNet at large scale [49, 73] with all-reduce. Nevertheless, other models are more challenging, with UGATIT as the least scalable one. Even BytePS only achieves 74% scaling efficiency. For such communication intensive models, BytePS has the most gain over all-reduce (84% with 256 GPUs). Despite UGATIT, BytePS has at least 91.6% scaling factor for the rest five 256-GPU training jobs.

We analyze the breakdown of performance improvement by comparing native-PS and BytePS, since they both use the same number of additional CPU machines. For example, BytePS outperforms native-PS by 52% with 256 GPUs on VGG-16 (Fig. 15(b)). Among the 52% improvement, we find that 19% comes from optimal communication design (intra-server), 18% comes from Summation Service, and the rest 15% comes from better implementation mentioned in §6.

8 Observations and Discussion

In this section, we share several of our observations and discussions, with the aim to inspire future research.

BytePS outperforms all-reduce even without extra CPU machines. Theoretically, the communication time is the same for all-reduce and BytePS when no additional CPU machines are available (§4.1). In practice, we observe that BytePS still outperforms all-reduce significantly in this case. One reason is that BytePS has a better intra-machine communication strategy than all-reduce. However, even without intra-machine optimization, BytePS still outperforms all-reduce (see Fig. 14 in §7). We hypothesize that BytePS has the advantage of

[8] We focus on system scalability and do not discuss algorithm scalability, *i.e.* the hyperparameter tuning and convergence speed with more GPUs.

allowing more "asynchronicity" than all-reduce. All-reduce usually requires additional out-of-band synchronization to ensure the consistent order across nodes, while BytePS does not have this overhead. However, to analyze it, we need a distributed profiler that can build the complete timeline of the execution and communication across all nodes in distributed training.

GPU cluster scheduler should consider dynamic CPU resources. By leveraging additional CPU machines, BytePS can speedup DNN training. Since BytePS can adapt to any number of CPU machines, it enables elasticity – the cluster scheduler can scale in or out CPU machines for existing jobs based on real time conditions. Most existing schedulers keep the number of GPUs of a job *static* because of convergence problems [16, 74]. Fortunately, the number of CPU machines in BytePS only impacts system performance but not model convergence. We plan to add elasticity support to BytePS, which will enable BytePS to *dynamically* schedule CPU resources during the training process.

Model-parallelism support. BytePS can accelerate the communication when reducing tensors across GPUs. Some model parallelism methods, such as Megatron-LM [62] and Mesh-TensorFlow [61], also rely on the all-reduce primitive for communication. Therefore, BytePS can also accelerate them by replacing the all-reduce operations.

9 Related Work

Acceleration of computation: To accelerate the forward propagation and backward propagation, the community has worked out many advanced compilers and libraries, including cuDNN [10], MKL [7], TVM [23], XLA [17], Astra [64] and other computation graph optimization, *e.g.,* Tensor Fusion [14] and graph substitution [37]. They focus on speeding up DNN computation. They are complementary to and can be used with BytePS.

Acceleration of communication: There are several directions for accelerating communication: (1) *Gradient compression* [21, 45] is proposed to reduce the communication traffic, *i.e.*, using half precision for gradient transmission, at the cost of potential degradation of accuracy. (2) *Communication scheduling and pipelining*: Recent work explores to better overlap the computation and communication by priority-based scheduling and tensor partition [31, 34, 55]. The ideas are that tensor partition enables simultaneous bidirectional communication, and that during communication, the former layers have higher priority because they are needed sooner for FP of the next iteration. Those ideas are complementary to BytePS, and they have been integrated into our implementation. Pipedream [51] adds parallelism between multiple batches. BytePS can also potentially accelerate its data parallel stages.

Hierarchical all-reduce: Some work proposes to leverage the hierarchical topology [24, 49] during all-reduce, in order to minimize the traffic at bottleneck links. However, they still rely on the assumption that resources are homogeneous while overlooking CPU resources. BytePS can outperform them by leveraging the heterogeneous resources. In fact, the latest NCCL includes hierarchical, tree-based all-reduce, which does not differ much from the results in §7.

Intra-machine optimization: Blink [68] also optimizes multiple GPU communication inside a single machine, by leveraging hybrid transfers on NVLinks and PCIe links. However, Blink does not optimize the distributed training cases, where the main communication bottleneck is the NIC and its PCIe connection instead of the much faster NVLinks. BytePS carefully schedules the intra-machine traffic to utilize the bottleneck bandwidth better – the NIC bandwidth. Our intra-machine design also considers the PCIe bandwidth consumed by the NIC, while Blink is only focused on GPU's PCIe connections.

New hardware chips or architecture for accelerating DNN training: Recently, there are many new chips, like TPU [38] and Habana [6], that are specifically designed for DNN training. In fact, the design of BytePS is not GPU-specific, and should apply to them as long as they are also PCIe devices. Some also propose using InfiniBand switch ASIC [28] to accelerate all-reduce, or using P4 switches [58, 59] to accelerate PS. E3 [46] leverages SmartNICs to accelerate network applications, and can potentially benefit BytePS by offloading the gradient summation from CPUs to SmartNICs. PHub [48] proposes a rack-scale hardware architecture with customized network configurations, *e.g.,* 10 NICs on one server. BytePS focuses on using generally available CPU and GPU servers in commodity data centers.

10 Conclusion

BytePS is a unified distributed DNN training acceleration system that achieves optimal communication efficiency in heterogeneous GPU/CPU clusters. BytePS handles cases with varying number of CPU machines and makes traditional all-reduce and PS as two special cases of its framework. To further accelerate DNN training, BytePS proposes Summation Service and splits a DNN optimizer into two parts: gradient summation and parameter update. It keeps the CPU-friendly part, gradient summation, in CPUs, and moves parameter update, which is more computation heavy, to GPUs. We have implemented BytePS and addressed numerous implementation issues, including those that affect RDMA performance. BytePS has been deployed, extensively used and open sourced [4]. Multiple external projects have been developed based on it. The Artifact Appendix to reproduce the evaluation is at [3].

11 Acknowledgement

We thank our shepherd Rachit Agarwal and the anonymous reviewers for their valuable comments and suggestions. Yimin Jiang and Yong Cui are supported by NSFC (No. 61872211), National Key RD Program of China (No. 2018YFB1800303).

References

[1] A Light-weight Parameter Server Interface. https://github.com/dmlc/ps-lite.

[2] Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/.

[3] Artifact Appendix. https://github.com/byteps/examples/blob/master/osdi20ae.pdf.

[4] BytePS. https://github.com/bytedance/byteps.

[5] Evaluation Code. https://github.com/byteps/examples.

[6] Habana. https://habana.ai/.

[7] Intel MKL. https://software.intel.com/en-us/mkl.

[8] Intel Xeon Platinum 8168 Processor. https://ark.intel.com/content/www/us/en/ark/products/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz.html.

[9] Libibverbs. https://www.rdmamojo.com/2012/05/18/libibverbs/.

[10] NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[11] NVIDIA DGX-1. https://www.nvidia.com/data-center/dgx-1/.

[12] NVIDIA GPU Direct RDMA Benchmark. https://devblogs.nvidia.com/benchmarking-gpudirect-rdma-on-modern-server-platforms/.

[13] NVIDIA NCCL. https://developer.nvidia.com/nccl.

[14] NVIDIA TensorRT Inference Library. https://devblogs.nvidia.com/deploying-deep-learning-nvidia-tensorrt/.

[15] Supermicro PCIe Root Architectures for GPU Systems. https://www.supermicro.org.cn/products/system/4U/4029/PCIe-Root-Architecture.cfm.

[16] Train ImageNet in 18 Minutes. https://www.fast.ai/2018/08/10/fastai-diu-imagenet/.

[17] XLA. https://www.tensorflow.org/xla.

[18] Amazon EC2 Pricing on demand. https://aws.amazon.com/ec2/pricing/on-demand/, 2019.

[19] TensorFlow MNIST Example with Horovod. https://github.com/horovod/horovod/blob/master/examples/tensorflow_mnist.py, 2020.

[20] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI 2016*.

[21] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive Residual Gradient Compression for Data-parallel Distributed Training. In *AAAI 2018*.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *LearningSys 2015*.

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 2018*.

[24] Minsik Cho, Ulrich Finkler, and David Kung. BlueConnect: Novel Hierarchical All-Reduce on Multi-tired Network for Deep Learning. In *SysML 2019*.

[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[28] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *COMHPC 2016*.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI 2019*.

[30] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA Over Commodity Ethernet at Scale. In *SIGCOMM 2016*.

[31] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *SysML 2019*.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR 2016*.

[33] Geoffrey Hinton, li Deng, Dong Yu, George Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Phuongtrang Nguyen, Tara Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *Signal Processing Magazine, IEEE*, 2012.

[34] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In *SysML 2019*.

[35] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC 2019*.

[36] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-precision: Training Imagenet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[37] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *SOSP 2019*.

[38] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA 2017*.

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI 2016*.

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for Key-value Services. In *SIGCOMM 2014*.

[41] Junho Kim, Minjae Kim, Hyeonwoo Kang, and Kwanghee Lee. U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. *arXiv preprint arXiv:1907.10830*, 2019.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS 2012*.

[44] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI 2014*.

[45] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv preprint arXiv:1712.01887*, 2017.

[46] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *ATC 2019*.

[47] Chris Lomont. Introduction to Intel Advanced Vector Extensions. *Intel white paper*, 23, 2011.

[48] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter Hub: A Rack-scale Parameter Server for Distributed Deep Neural Network Training. In *SoCC 2018*.

[49] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, and Yuichi Kageyama. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233*, 2018.

[50] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *ATC 2013*.

[51] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP 2019*.

[52] Tony Paikeday. Steel for the AI Age: DGX SuperPOD Reaches New Heights with NVIDIA DGX A100. https://blogs.nvidia.com/blog/2020/05/14/dgx-superpod-a100/, May 2020.

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS 2019*.

[54] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 2009.

[55] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP 2019*.

[56] Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. Large Scale Language Modeling: Converging on 40GB of Text in Four Hours. In *SBAC-PAD 2018*.

[57] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 2019.

[58] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network Computation Is A Dumb Idea Whose Time Has Come. In *HotNets 2017*.

[59] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-network Aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[60] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *CoRR*, 2018.

[61] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *arXiv preprint arXiv:1811.02084*, 2018.

[62] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv: 1909.08053*, 2019.

[63] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. Astra: Exploiting Predictability to Optimize Deep Learning. In *ASPLOS 2019*.

[65] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. In *ICML 2013*.

[66] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM 2009*.

[67] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *NIPS 2017*.

[68] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In *MLSys 2020*.

[69] Yuxuan Wang, R. J. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc V. Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: A Fully End-to-End Text-To-Speech Synthesis Model. *CoRR*, abs/1703.10135, 2017.

[70] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *OSDI 2018*.

[71] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI 2018*.

[72] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards Zero Copy Dataflows Using RDMA. In *SIGCOMM 2017 Posters and Demos*.

[73] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. *arXiv preprint arXiv:1811.06992*, 2018.

[74] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[75] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM 2015*.

[76] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.