ZEBRA: Accelerating Distributed Sparse Deep Training with In-network Gradient Aggregation for Hot Parameters

Heng Pan*, Penglai Cui†, Zhenyu Li†, Ru Jia†, Penghao Zhang*, Leilei Zhang†, Ye Yang†
Jiahao Wu†, Mathy Laurent‡, Gaogang Xie*
*CNIC, CAS, China †ICT, CAS, China ‡University of Liege, Belgium

Abstract—**Distributed sparse deep learning has been widely used in many Internet-scale applications. Network communication is one of the major hurdles for training performance. In-network gradient aggregation on programmable switches is a promising solution for speeding up the performance. Nevertheless, existing in-network aggregation solutions are designed for the *dense* deep training, and fall short when used for the *sparse* training. To address this gap, we present ZEBRA based on our key observation on the extremely biased update frequency of parameters in distributed sparse deep training. Specifically, Zebra offloads only the aggregation for "hot" parameters that are updated frequently onto programmable switches. To enable this offloading and achieve high aggregation throughput, we propose solutions to address the challenges related to hot parameter identification, parameter orchestration and gradient aggregation as well as system reliability. We implemented Zebra on Intel Tofino switches and integrated it with PS-lite. Finally, we evaluate Zebra's performance through extensive experiments and show that it can speed up the gradient aggregation by 1.5∼4× and the end-to-end performance by 1.4∼2.6×.**

I. INTRODUCTION

High-dimensional *sparse* data widely exists in Internet-scale deep learning (DL) applications, such as search engine, recommendation systems and online advertising [1], [2]. To enable efficient DL training with sparse data, sparse DL models [1], [3], [4] typically use a two-tier architecture, where the first tier is a *SparseNet* that embeds high-dimensional sparse data into a low-dimensional space via representation learning [5], and the second is a *DenseNet* that models the relationship between the dense embedding representation and supervised labels. Two unique features distinguish the sparse DL models from the dense ones: (*i*) *data sparsity* that the samples from training data contain a large number of features, but only a few are non-zero; (*ii*) *model sparsity* that the most gradients are zero in each training iteration.

We have witnessed a huge increase in the size of sparse DL models and the training datasets in recent years. For example, for a business advertising system [1], petabytes (PB) of training data is generated everyday, and the trained DL model consists of billions of features. Indeed, training a sparse DL model is a time-consuming task, and the distributed sparse DL training, which leverages a cluster of nodes to perform training tasks cooperatively, emerges and becomes a practice [6].

A popular architecture for distributed DL is the parameter server (PS) architecture with data parallelism [1], [7]. The training dataset is divided into equal-sized parts (called chunks). In each iteration, workers pull the up-to-date model from parameter servers and perform local training with a chunk of data. The local training results are then sent to parameter servers for global DL model update. Distributed sparse DL training also follows the above procedure. It has been found that with the increasing of used workers, the communication between workers and parameter servers will be one of the major hurdles for training performance [8]. Specifically, a recent measurement study [9] reveals that the intensive and bursty communications are the two factors that hurt the performance.

A straightforward way to mitigate intensive communication is to reduce transmission data volume via gradient compression (*e.g.,* gradient quantization [10] and sparse parameter synchronization [11]). Other solutions aiming at mitigating communication burstiness decouple the dependency between computation and communication [8], [12] through scheduling. These solutions improved the performance at the end node side (*i.e.,* workers). Another recently emerging direction is to explore the computation capacity in network devices, especially programmable switches, for gradient aggregation [13]–[15], which can further accelerate the training process. Nevertheless, *all* the previous in-network aggregation solutions are targeted for *dense* DL models. We found they indeed fall short in accelerating the training for sparse DL models, because their streaming-based aggregation assumes synchronising *all* the local updates (*i.e.,* gradients) in every worker in each training iteration, regardless of whether individual updates are zero or non-zero. This assumption no longer holds in distributed sparse DL training, as only the non-zero embedding vectors and non-zero gradients are transmitted in $\langle key, value \rangle$ pairs.

To address the above gap, we design and implement *Zebra* to enable in-network gradient aggregation for distributed sparse DL training in data parallelism. Zebra is built on our key observation from industrial sparse DL applications that the update frequencies of parameters in sparse deep models are extremely biased. Taking the sparse DL model for online recommendation as an example, about 50% of updates are for only the top 30,000 parameters (out of 150 million parameters) (§ III-A). Zebra thus performs gradient aggregation on programmable switches only for these *hot* parameters, and let the servers aggregate the gradients for the remaining cold ones. That said, rather than offloading the gradient aggregation task for *all* the parameters in [13]–[15], Zebra only offloads the task for the hot parameters, while keeping the aggregation task for the cold parameters in PS servers.

We address several challenges to implement Zebra. First, we respectively propose a sampling-based (offline) and an iteration-based (online) mechanisms to identify hot model parameters (§ III-C). Second, we design a heat-based parameter placement mechanism at the switch side and a parameter layout (on switch registers) aware gradient packaging mechanism at the worker side to cooperatively reduce the probability that the associated parameters of one gradient packet belong to one register (§ III-D). In doing so, we reduce the chances that one packet writes a register multiple times in a pipeline, which is not supported by programmable switches. Third, inspired by [16], we adopt a table-lookup mechanism to enable floating-point summation on switches, which is essential for gradients aggregation (§ III-E). Last but not least, we enhance the reliability of Zebra from the perspectives of packet loss recovery (§ III-F).

We implement Zebra and integrated it with PS-Lite [17] and Intel Tofino [18] programmable switches (§ IV). We perform extensive experiments using a benchmark that includes various sparse DL training tasks (§ V). The results demonstrate the superior performance of Zebra with limited extra overhead, in comparison with the state-of-the-art solutions. In summary, our contributions are three-fold:

- We design Zebra that accelerates the distributed sparse DL training with in-network gradient aggregation on programmable switches. Specifically, it offloads the aggregation task for “hot” parameters from PS servers to programmable switches.
- We propose the solutions that include the sampling-/iteration-based hot parameter identification, the heat-based parameter placement on switches, the parameter layout aware gradient packaging on workers, and the enhancement for improving reliability.
- We implement Zebra, and integrate it with PS-lite and Intel Tofino switches. Extensive experiments with real-world sparse DL applications demonstrate that Zebra improves the aggregation throughput by 1.5$\sim$4$\times$ and the end-to-end performance by 1.4$\sim$2.6$\times$ with limited extra overhead.

II. Background and Motivation

A. Distributed Deep Learning

Distributed deep learning (DDL) leverages a cluster of training nodes (called *workers*) to cooperatively train DL models. We consider the widely used *data parallelism* mode, where the training dataset is divided into equal-sized parts to feed training nodes for local training. A widely adopted DDL architecture in industry is the parameter server (PS) architecture; Microsoft Multiverso [20], Alibaba XDL [1] and ByteDance BytePS [12] are all built with this architecture. In the PS architecture, workers train their local DL models, while individual PS servers manage globally shared but non-overlapped DL model parameters. That said, the parameters that a PS server manages are non-overlapped with the parameters of any other PS servers. A scheduler is in charge of managing workers and PS servers (*e.g.,* node addition/removal), and synchronizing data among them. Comparing with other architectures (e.g., the all-reduce architecture), the PS architecture provides high scalability, enables strong fault tolerance and achieves dynamically load balancing; this is indeed friendly for heterogeneous training platform. In each iteration of training, workers first pull the up-to-date model from parameter servers, and then perform forward-backward computation with one chunk of training data locally. At the end of an iteration, workers push the trained results (*i.e.,* gradients) to the servers for updating the DL model. Most of training jobs are usually trained in the synchronous mode since it has been observed often to achieve faster convergence than asynchronous distributed training over GPUs [21].

B. Sparse Deep Learning

High-dimensional sparse data widely exists in Internet-scale applications (*e.g.,* search engine and online advertising). Data sparsity could cause low training efficiency if not handled properly. To this end, several sparse DL models have been proposed [1], [3], [4]. These models typically follow a two-tier architecture (see Figure 1): representation learning (SparseNet) and function fitting (DenseNet). The representation learning embeds high-dimensional sparse data into a low-dimensional space via embedding layers, while the function fitting models the relationship between the dense embedding representation and supervised labels.

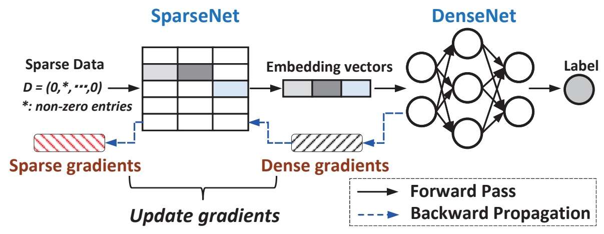


Figure 1. A conceptual two-tier architecture of sparse deep training.

Sparse model training. Specifically, in the forward pass, a training node reads a batch of sparse data samples, maps them into dense embedding vectors via SparseNet[1], and then feeds the results into DenseNet. The backward propagation reverses this flow and output gradients. The generated gradients in each iteration consist of two parts: the *sparse gradients* for SparseNet and the *dense gradients* for DenseNet; only a few vectors of the SparseNet gradients are non-zero.

Two unique features of the SparseNet distinguish *sparse* deep training from *dense* deep training. First, as listed in

[1]Usually, one can look up a huge dictionary or utilize a few CNN/RNN models to implement data mapping.

Table I
NEURAL NETWORK CHARACTERISTICS OF MULTIPLE LAYER PERCEPTION [4] (MLP), CROSSMEDIA [23] (CM) AND DEEP INTEREST NETWORK [24] (DIN).

Deep Model	Neural Net.	# parameters
MLP	SparseNet	18 Billion
	DenseNet	1.2 Million
CM	SparseNet	5 Billion
	DenseNet	1 Million
DIN	SparseNet	18 Billion
	DenseNet	1.7 Million

Table I which shows the model characteristics of three popular sparse DL models, the SparseNet uses a much larger training network with billions of parameters than that in DenseNet (millions of parameters). Second, while many gradients of SparseNet in each iteration are zero (*i.e.,* sparse), there are still many heavy non-zero vectors. Let us take a NCF [3] model with a typical training dataset [25] as an example. In each iteration, out of 680MB of SparseNet gradients, ~104MB are for non-zero vectors; in comparison, the DenseNet only generates 0.4MB of gradients.

Distributed training of sparse DL models. In distributed sparse training, each worker runs the local training job showed in Figure 1, and synchronizes with other workers for both the SparseNet model and DenseNet model. To reduce the amount of gradient data for transmission, individual workers encode gradient vectors as a list of key-value pairs, and push only the non-zero vectors to PS servers in each iteration [26]. Consequently, the parameters (indices) involved in the transmitted gradients from different workers may not be overlapped.

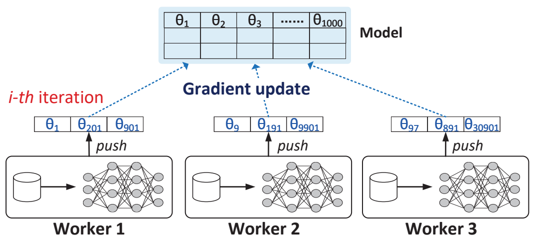


Figure 2. Gradient updates in distributed sparse training.

Figure 2 shows such an example. In the *i-th* iteration, *Worker* 1 pushes 3 non-zero gradients (θ_1, θ_{201} and θ_{901}) for updating the global model, while *Worker* 2 and *Worker* 3 transmit non-zero gradients for other parameters (*e.g.,* $\{\theta_9$, θ_{191}, $\theta_{9901}\}$ from *Worker* 2). It is worth noting that the parameters with non-zero gradients in individual workers are unknown in advance.

C. Limitations of Prior In-network Aggregation

Recently, a large amount of research effort [13]–[15] has been devoted to *in-network gradient aggregation* in order to reduce the communication volume and improve the overall training performance. Specifically, programmable switches are exploited to sum the gradients sent by workers; the summation results are either directly returned back to workers as the updated global model [13], or sent to PS servers [15]. By doing so, the data volume to PS servers is greatly reduced.

We particularly focus on two state-of-the-art systems that leverage programmable switches for in-network gradient aggregation: SwitchML [13] and ATP [15]. While being effective for dense training, they fall short in speeding up **sparse** DL training because of their streaming-based aggregation.

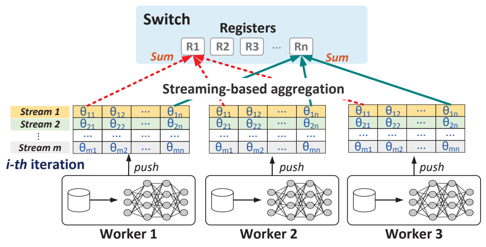


Figure 3. The streaming-based in-network gradient aggregation in [13], [15].

To show this, we briefly describe their workflow. As illustrated in Figure 3, the gradients are chunked into m streams at each worker, where the j-th ($j < m$) stream in each worker contains the gradients for the same set of parameters. Each worker sends one stream at each time slot to programmable switches for aggregation. Programmable switches will aggregate (*i.e.,* sum) the gradients of the j-th streams from all workers. This approach works well in dense deep learning, where in each iteration, the gradients of all the parameters need to be sent out for aggregation. Besides, because of the limited storage space in programmable switches, some approaches (*e.g.,* ATP) require synchronization of time slots among workers — all workers not to transmit the next stream of gradients until the programmable switches relieve the last aggregated results.

The above streaming-based approach require each worker to synchronize all the trained results in each iteration, including both the zero and non-zero gradients. Thus, it does not work for deep sparse training since only the non-zero gradients are sent to PS servers for aggregation. Worse still, the parameters with non-zero gradients in individual workers are unknown in advance. Simply applying existing approaches will lead to limited chances of aggregations on programmable switches, or even incorrect aggregation.

III. ZEBRA DESIGN

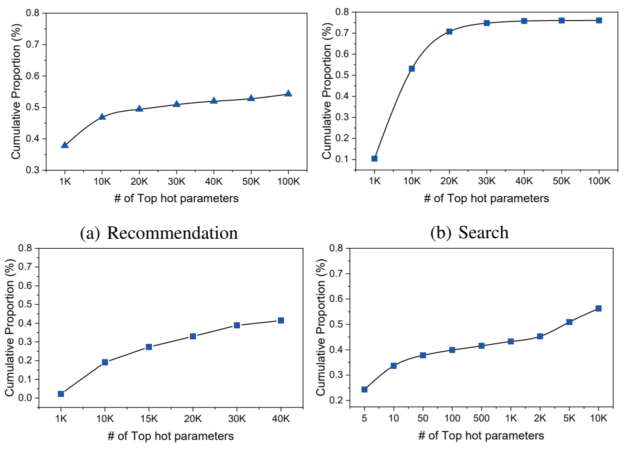
This section presents the design of Zebra that is able to accelerate distributed **sparse** DL training. We begin with the key observation that encourages the design of Zebra, and then detail Zebra.

A. Characterizing "hot-cold" Phenomenon

We first present our observation on the highly skewed update frequency of parameters in distributed sparse DL training. This observation is derived from our (industrial) training tasks of two typical sparse deep learning models: search engine and online advertising recommendation. Without loss of generality, we also study two other popular open-source sparse DL training models, DeepLight [27] and NCF [3].

Specifically, we use a testbed with 16 workers and 1 PS servers to train DDL models for online advertising recommendation (Task 1), search engine (Task 2), DeepLight (Task 3) and NCF (Task 4). The model of Task 1 consists of 150 million parameters, while Task 2 contains about 9 million model parameters. The number of parameters in Task 3 and Task 4 are relatively small, 0.32 and 0.07 million receptively. During training, we collect logs from the PS server, and count the update frequency of each parameter.

We sort the model parameters based on their update frequency, and present the cumulative proportion of update frequency for the top-k parameters in Figure 4. Note that we report the cumulative proportion for the top-100K model parameters in Task 1 and Task 2, while presenting the results for the top-40K parameters in Task 3 and the top-10K parameters in Task 4 because of their small model sizes. The most surprising finding is that they contain very "hot" parameters who contribute the bulk of the update. For example, across the 150 million parameters, the top 30,000 parameters of Task 1 constitute over half of all updates. Likewise, for Task 2, its top 30,000 parameters even account for about 70% of updates. Similar results hold for the two open-source sparse models: the top 30,000 parameters of Task 3 contribute 40% updates, and the top 5,000 parameters of Task 4 account for over half of all updates (the top 100 parameters already contribute about 40%). We believe this stems from the internal structure of sparse deep models as other research [9] also reported this phenomenon. For example, the feature of the commodity ID in online recommendation is always activated, while only some of the parameters that capture various user interests may be activated. Due to the high update frequency of hot parameters, the overlap of the non-zero element indices across different workers is high.



(a) Recommendation

(b) Search

(c) DeepLight

(d) NCF

Figure 4. Cumulative distribution of the parameter update frequency for four sparse models.

The above analysis reveals that high-dimensional sparse deep learning tasks exhibit a *hot-cold phenomenon*, where the hot parameters contribute most communication traffic. Putting this in the context of in-network gradient aggregation, we envision a system that aggregates the gradients only for the "hot" parameters as they contribute most of the updates. By doing so, we can accelerate the distributed sparse training with a limited storage requirement on programmable switches.

B. Design overview

To restate, Zebra uses programmable switches for in-network sparse gradient aggregation. Figure 5 shows its two components on the switch, namely $Zebra\_p$ and $Zebra\_s$. $Zebra\_p$ is deployed on the switch pipeline to implement gradient aggregation, while $Zebra\_s$ runs on the local CPUs of the switch to facilitate reliable transmission (see Section III-F). Given the "hot-cold" phenomenon we observed, Zebra adopts a hot-cold parameter separation mechanism. Specifically, it selects those "hot" model parameters that are frequently updated for aggregation on switches, while the remaining "cold" parameters are handed over to the parameter servers without in-network aggregation. For one sparse DL model, the hot parameters are relatively fixed; Zebra reserves limited switch registers to cache the top hot parameters and aggregate the majority of update traffic. Before the next iteration, Zebra will clear these reserved registers.

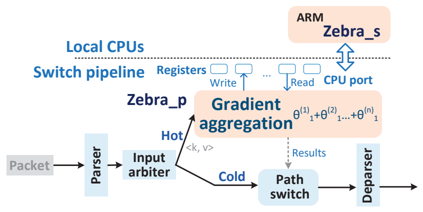


Figure 5. Structure overview of Zebra.

C. Hot Parameter Identification

Without running the training work, the "hot" parameters are unknown. In fact, the hot parameters cannot be accurately predicted because the update frequency of parameters do not follow a pre-defined pattern and thus may vary from a sparse DL task to another (see Figure 4). To address this problem, we propose two potential solutions as follows.

Sampling-based offline method. Intuitively, one straightforward yet efficient way is to utilize a sampling-based mechanism to approximately capture the update frequency distribution of DL model parameters. Specifically, we first extract a small training dataset by randomly sampling the whole dataset. And then, we train the DL model with the sampled dataset and record the update frequency of each model parameter. Finally, we sort the parameters based on their update frequency, and label the top k parameters as hot ones together with their ranks. As we will show in §V-C, by running the training with only 4∼8% of the whole dataset as input, we can identify hot parameters with an accuracy as high as 90%.

Iteration-based online approach. Besides sampling, we also propose an iteration-based approach to dynamically identify hot model parameters. Specifically, we train the DL model with the whole raw datasets as normal. For T training iterations, the scheduler gathers the update frequency of DL model parameters from programmable switches and PS servers[2], and re-calculates the top k parameters and their ranks. Finally, if the newly computed hot parameter set and ranks differ from the currently used ones significantly[3], the computed set will be dispatched to workers and programmable switches for update. The scheduler is in charge of the hot parameter set update progress in workers and switches. This approach works because the frequently activated model parameters (a.k.a non-zero gradients) in individual iterations are often similar when training sparse DL models. To confirm it, we evaluate the efficiency of the approach in §V-C.

Overall, the sampling-based approach is relatively simple, but it requires a pre-training course with sampled datasets, and introduces extra overhead. For a better accuracy, it often needs higher sampling rates. That said, it is suitable for not too large training tasks. On the contrary, the iteration-based solution can avoid the course and capture the dynamics of the hot parameters, but also at the cost of the increase in the system complexity. Therefore, for a specific training task, we can select one appropriate method between them to identify hot parameters.

Indeed, with the above methods, we are able to identify the top k parameters as hot ones, but how to determine the value of k remains a challenge. The key reason is that it actually depends on the available resources on switches. To this end, we further propose Principle 1 to calculate the value of k.

Principle 1: (Hot parameter Identification). Consider a list of parameters θ={θ_1, θ_2, ..., θ_n} that are ranked in descending order by the update frequency. We use UF={uf_1, uf_2, ..., uf_n} to refer to their corresponding update frequency. We also assume that one programmable switch is equipped with m MB on-chip memory, and storing a model parameter in a switch consumes 4 bytes of memory. We say that the top-k parameters ({θ_1, ..., θ_k}) are hot if they satisfy the following two conditions:

$$\begin{cases} T_k/T_n \geq p \\ 4B \times k \leq c \times m \quad MB \end{cases}$$

where T_k=$\sum_{i=1}^{k} uf_i$, T_n=$\sum_{i=1}^{n} uf_i$, $p \in (0,1)$ and $c \in (0,1)$ are two design parameters. The on-chip memory in our Tofino switch is 20MB, *i.e.,* $m = 20$.

The parameter c is the fraction of on-chip memory that we would like to use for gradient aggregation. In practice, c should be small ($0.05 \sim 0.1$) because occupying too much memory would affect conventional functions of switches (*e.g.,* packet forwarding). The parameter p is the expected proportion of traffic that will be intercepted and processed by the switch. While a larger p is preferred (so as k), more memory would be required. Nevertheless, as showed in Figure 4, the marginal benefit in terms of traffic saving (*i.e.,* the growth of p) when increasing k beyond some points becomes very small. Let us take Figure 4(b) as an example; increasing k beyond $30K$ will bring very limited benefit in terms of traffic saving. In this example, k will be set as 30,000 in our implementation.

[2]To reduce the overhead, only the model parameters whose update frequency is non-zero are transmitted.

[3]Actually, we use the popular Kendall tau distance [28] to quantify their difference.

D. Parameter Orchestration

Parameter layout in switch registers. The on-chip memory in a switch is physically organized as registers. A register is similar to an array, which consists of multiple register slots.One hot parameter takes one register slot to cache its gradient; a new gradient will be added to the cached gradient to get the final summation of all the gradients of this parameter. A practical restriction on switches is that one register can be operated only once in one pipeline. That said, if a register cached gradients of two parameters whose updates are carried in one packet, they would not be able to be aggregated in one pipeline. While we could use the *recirculation* operation in programmable switches to recirculate the packet back to the pipeline, recirculations can degrade the performance. As such, we need to carefully assign hot parameters to registers so that the chances that the updates of two parameters assigned in one register are carried in the same packet is low.

Before delving into the detail of parameter layout, we first describe the mapping of parameter indices. Gradients in distributed deep sparse training are transmitted in the form of $\langle key, value \rangle$ pairs, where key is the index of a parameter. Because we will use the index to directly locate their corresponding register slots, we need to map the indices obtained from the models into the range of $[0, M-1]$, where M is the total number of register slots used for gradient aggregation[4]. Suppose we rank the parameters in descending order based on the update frequency, then a parameter's index after mapping is its rank (from 0 to $M-1$). Workers store the mapping information locally, which is used to restore the indices of the aggregation results.

Next we describe the parameter layout in registers using a heat-based parameter placement mechanism. Specifically, let us consider a list of hot parameters {θ_1, θ_2, ..., θ_n} ranked in descending order of their update frequency (*i.e., heat*). Our method places the n parameters into m registers as follows. The i-th register stores the $(i+m*j)$-th parameters, where $0 \leq j \leq \lfloor n/m \rfloor$. Our intuition is that the "heat" distribution of the hot parameters is non-uniform (see Figure 4), so the heat of θ_i would be much higher than that of θ_{i+m}. Therefore, the probability that the gradients of θ_i and θ_{i+m} are encapsulated into one packet is low. That said, we can use one register to store θ_i and θ_{i+m}. With this basis, we deduce the above conclusion that θ_{i+m*j} ($0 \leq j \leq \lfloor n/m \rfloor$) are assigned to one register. We will show the effectiveness of this layout in §V-D.

Packaging gradients at workers. At the worker side, we adopt a parameter layout aware gradient packaging mechanism for encapsulate gradients into packets (see Algorithm 1). The core idea is to encapsulate parameter gradients into packets in order to achieve two goals: (*i*) reducing the likelihood that the parameters encapsulated to one packet are cached in one

[4]Suppose m registers are used and each has r register slots, then $M = m*r$.

Algorithm 1: PARAMETER_ORCHESTRATING(*G, m*)

```
Input: $G$, a batch of gradients to be transmitted.
Input: $m$, the number of switch registers.
Output: return some packets that carry the gradients.
1  $P \leftarrow$ Estimate_packets(G);
2  $G^{'} \leftarrow \emptyset$; $P_1 \leftarrow P$;
   // P is a list of packets.
   // Packaging G to $n$ packets
3  for $\theta \in G$ do
4      $k \leftarrow \theta.id\%m$; // Get register ID
5      $P_2 \leftarrow P_1$; // $P_2$ carries candidates for $\theta$
6      $is\_inserted \leftarrow false$;
       // Find out pkts having already
          include the parameters sharing the
          same register with $\theta$
7      $res \leftarrow$ same_reg_pkt_find(k);
8      if $res \neq NULL$ then
9          $P_2$.erase($res$.begin(), $res$.end());
10     $pkt \leftarrow P_2$.first();
11     if $pkt \neq NULL$ then
           // $pkt$ will carry $\theta$.
           // $P_1$ tracks the full state of pkts.
12         Update($P_1$, pkt, $\theta$);
           // Final results.
13         Update(P, pkt, $\theta$);
14         $is\_inserted \leftarrow true$;
15         if is_full(pkt) == True then
16             $P_1$.erase(pkt);
17     if $is\_inserted \neq true$ then
18         $G^{'}$.append($\theta$);
19 if $G^{'} \neq \emptyset$ then
20     P.insert(create_pkt_padding($G^{'}$));
21 return $P$;
```

register; (*ii*) using as few packets as possible. We assume that workers know the number of registers m. When a batch of gradients for the hot parameters need to be transmitted, we use Algorithm 1 for packaging gradients into packets. The algorithm first estimates how many packets (denoted by P) are needed to carry these parameter gradients (line 2), and then process each parameter θ as follows, where θ is the (rank) index of the parameter.

- First, it uses the index of θ to locate the register ID (k) that caches θ (line 4) at switches;
- Second, it filters out the packets that have carried at least one parameter that also belongs to the k-th register (line 7 to 9);
- Third, it appends θ in a candidate packet and updates corresponding states. (line 10 to 16).
- Forth, if there is no such packet, θ will be recorded to a set ($G^{'}$) for further processing (line 17 to 18).

Finally, if $G^{'}$ is not empty, the worker will encapsulate the parameter gradients in $G^{'}$ into a number of packets without considering the parameter layout in switch registers (line 19-20). The reason why we do not use the packets in P to carry the parameters in $G^{'}$ is as follows. Let us assume that the number of parameters in $G^{'}$ is n. If we used the packets in P to carry $G^{'}$, it would lead to n times recirculation operations. On the contrary, if we encapsulate them into new packets directly, the usage of recirculation operations can be significantly saved, because the parameters in $G^{'}$ are unlikely to belong to one register.

E. Gradient aggregation on switches

Switches parse the received gradient packets using their programmable parser. By doing so, the switch will get a list of $\langle key, value \rangle$ pairs, each of which is the gradient (value) of a parameter with *key* as the index. Specifically, for an update of the parameter θ, the switch locates the register as well as the slot in the register that caches θ. More specifically, we use a hash table to map the index of one parameter into the position of the registers. Then the update (*i.e.,* gradient) is added to the cached value.

Floating-point summation on switches. Gradients are typically represented as 32-bit floats. Nevertheless, the pipeline in programmable switches does not support floating-point summation. Previous studies [14], [15] use a floating-to-integer method, which coverts floats to integers using a scaling factor at the worker side before sending the gradients. This approach may require the negotiation of the scaling factor among workers for several times, and may also introduce accuracy loss [16]. Instead, we adopt the table-lookup method in [16] that implements on-the-fly floating-point summation.

Local model update. Switches receive gradients of hot parameters from all workers[5] and aggregate them. On completion of aggregation of parameters, the switches send the aggregated results to the PS which broadcasts the all updated parameters back to each worker for updating their local models.

F. System Reliability

The system reliability is important in production environments. We consider the packet loss failure that would undermine Zebra.

Packet Retransmission. While packet loss happens much less frequently in data center networks ($< 10^{-3}$ [29]), it indeed may happen. A packet loss would result in the loss of all the gradients the packet carries and thus degrading the training performance. To this end, we leverage the per-packet ACK mechanism for loss detection and retransmit the lost packets. Specifically, the receiver of a packet will immediately return an ACK to the sender. That said, switches will ack the gradient packets for hot parameters sent from workers; PS servers will ack the packets for cold parameters and aggregated results of hot parameters; and workers ack the aggregation packets that are sent from PS servers. The sender of a packet will mark

[5]If one worker does not contain hot parameter gradients in this iteration, it will send an "empty" packet to switches.

the packet as loss if it does not receive the ack before the timer expires; the retransmitted packet has the same sequence number as the original one. One challenge is how the switches retransmit these aggregated packets of hot parameters. To this end, Zebra uses the switch CPUs to reserve one copy of the aggregated packets and retransmit them after timeout.

To implement the above mechanism, switches need to keep the unacked aggregation packets locally for possible packet retransmissions. To achieve it, one potential solution is to follow the synchronization pattern like adopted in [14], [15]. Specifically, switches begin to transmit the next aggregation packet only when an ack packet for the current one has been received. That said, the switch pipeline that contain limited on-chip memory does not reserve many unacked aggregation packets. However, this comes at the cost of low communication performance due to its ping-pong transmission.

Instead, our Zebra adopts another pathway that utilizes the switch control plane to cache those unacked aggregation packets and take in charge of reliable transmission. That said, when sending out aggregation packets to workers, switches also forward one copy to their local CPUs ($Zebra\_s$) for possible retransmissions. Someone might worry that the communication between the switch control plane (local CPUs) and data plane (switch pipeline) will become a new bottleneck. But in fact, each iteration just generates about tens-of-megabytes aggregation packets at most, since programmable switches contain only 20 MB registers for hot parameter aggregation. This is indeed much smaller than the bandwidth between the switch control plane and data plane, which achieves up to about 100Gbps [18], and switch local CPUs are able to process the aggregation packets. Furthermore, our experiments also show that the extra latency introduced by this approach is also very low (§ V-E).

IV. Implementation

We implement a prototype of Zebra on commodity servers and programmable switches. Specifically, data plane components at switches are implemented with $P4_{16}$ language (1400+ LoC) and compiled with the Intel Capilano software suite; they are deployed to 3.2 Tbps Intel Tofino switches[6]. Zebra host stack is customised on lwIP [30], a lightweight user-level TCP/IP stack. Finally, we integrate Zebra into PS-lite [17], an open-source parameter server platform that is the basis of many popular learning systems (*e.g.,* MXNext).

Data plane component. Programmable switches like the Intel Tofino switch that we use have some restrictions on the access of registers and tables. A particular restriction that we have to deal with is that a register can only be read and written once per packet processing. In case we need to read and write a register more than once, we adopt the *recirculation* operation that recirculate the packet back to the pipeline. In our implementation, a packet is at most recirculated once, *i.e.,* it goes through the pipeline twice at most.

[6]Each switch consists of two pipelines (*i.e.,* ingress and egress pipelines), which contains 12 stages in total.

Host stack. lwIP is a lightweight user-level TCP/IP stack; the current versions running on Linux servers only provide TAP/TUN virtual adapters. TAP/TUN adapters incur multiple packet copy operations and context switches between the kernel space and user space, degrading the whole performance. To this end, we turn to DPDK (Data Plane Development Kit) [31]. Specifically, we use DPDK APIs to implement a network device driver based on the *template* provided by lwIP; by doing so, lwIP can send/receive packets at line speed.

System integration. PS-lite utilizes ZeroMQ [32] (a.k.a zmq), which is a high-performance asynchronous messaging library, to enable high-performance communication between workers and PS servers. However, the vanilla ZeroMQ is based on the kernel stack. Thus, we use lwIP to replace its kernel stack via replacing the interfaces invoked in ZeroMQ.

V. Evaluation

A. Methodology

Testbed Setup. Our testbed includes 8 physical machines connecting to an Intel Tofino programming switch (3.2T/s) that is equipped with two ARM cards (ARMv8.1 SoC, 24 cores). Each physical machine is equipped with 2 32-core Intel®Xeon®4214R CPU, 128GB memory, a Mellanox CX-5 dual-port 100G NIC and an NVIDIA RTX3090 GPU; the OS is Ubuntu 16.04.

Benchmarks. Our benchmark models include two types of sources: two industrial sparse DL applications and several real-world sparse models. The two industrial applications are search engine (SE) and on-line advertising (OA) from a large Internet enterprise (see Figure 4 for their characteristics). The real-world sparse models include DeepLight [27], LSTM [33] and NCF [3], which are trained by a number of open-source datasets [25], [34].

Baselines. We compare Zebra with SwitchML[7] [13], a state-of-the-art in-network aggregation solution, and the popular PS-lite framework[8] which is widely used to train sparse DL models in the community. To enable SwitchML to support sparse DL training, the gradients of the entire sparse DL model are transmitted in each iteration, rather than only the non-zero gradients. It is noteworthy that the default available memory of programmable switches for aggregation is limited to 1MB (5%*20MB) for both Zebra and SwitchML to avoid impacting the conventional network functions of switches (e.g., routing, firewalling and load balancing) that also consume switch memory. But we relax the limitation to 2MB when dissect the impact of memory cap. Furthermore, each gradient packet size is about 192 bytes. Due to the limited space, we only show the performance of Zebra that is equipped with the sampling-based offline mechanism (§III-C) to identify hot model parameters. But we also evaluate the efficiency of the both two identification methods.

[7]Comparing with SwitchML, ATP supports multi-tenant learning, which is not concerned by this paper. Thus, we choose SwitchML rather than ATP as one baseline.

[8]We choose PS-lite as one baseline because it is the basis of many PS-based learning systems (*e.g.,* MXNet).

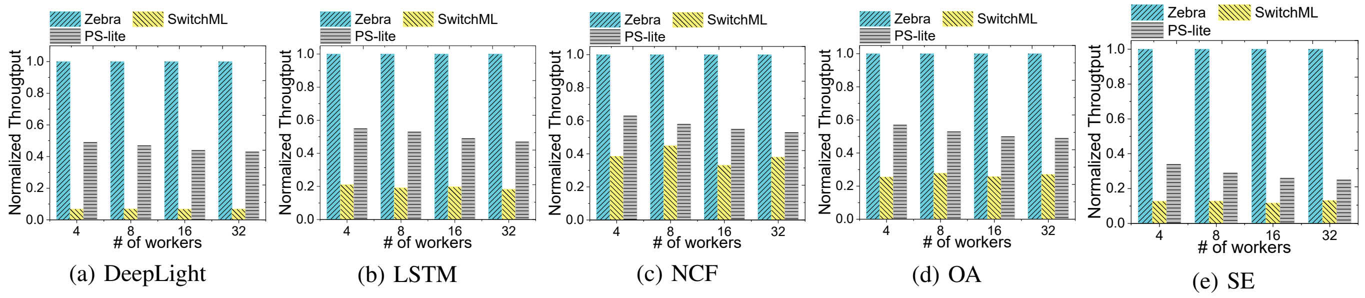


Figure 6. Aggregation throughput normalized by that of Zebra.

Overall, we evaluate Zebra from five aspects: (*i*) the effectiveness of sparse gradient aggregation; (*ii*) the feasibility of our proposed methods for identifying hot model parameters; (*iii*) the benefit of the hot parameter layout and orchestration; (*iv*) the introduced mechanisms for reliability, (*v*) the resource consumption on the switch data plane, and (*vi*) the end-to-end performance improvement.

B. Aggregation Throughput

We first evaluate the aggregation throughput of Zebra, and compare it with the two baselines. To this end, we use the tensors generated by training the benchmark models to test the aggregation throughput, where each worker transmits the tensors it generated when training individual models[9]. It is noteworthy that in SwitchML, each worker has to transmit all gradients in each iteration, regardless whether the gradients are zero or not. Besides, we selected the top 30,000 parameters with the most update frequency as hot parameters for OA and SE (see Figure 4), top 10,000 for NCF, and top 40,000 for LSTM and DeepLight.

Figure 6 shows the aggregation throughput of 5 models. Zebra consistently achieves higher aggregation throughput than that of the baselines, because of its sparsity-awareness to aggregate only the hot parameters on programmable switches. Indeed, SwitchML falls short in terms of aggregation throughput in supporting these deep sparse training, because of its streaming-based aggregation that has to aggregate all gradients for all parameters on switches (See §II-C). Because of the limited memory in the data plane for gradient aggregation (1MB), the benefit of in-network aggregation in SwitchML may even be out-weighted by its extra overhead, leading to even poorer performance than the PS-lite, especially for the models with high sparsity (*e.g.,* DeepLight). Note, the PS-lite follows the typical deep sparse training where only the non-zero gradients are transmitted. We also find that the advantage of Zebra becomes more significant as the number of the workers gets larger, because more workers means more opportunities of in-network aggregation. Specifically, with 32 workers, the aggregation throughput can be improved by 1.5∼4× in comparison with the PS-lite.

[9]As in [14], this section does not directly compare the end-to-end training performance in order to eliminate the impacts of computation capacities on workers on training performance. We leave the end-to-end evaluation in subsequent sections.

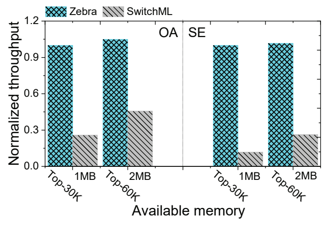


Figure 7. Aggregation throughput with different memory caps, normalized by that running in the default configuration.

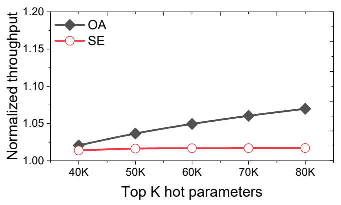


Figure 8. Zebra's throughput with different hot parameters, normalized by that running in the default configuration.

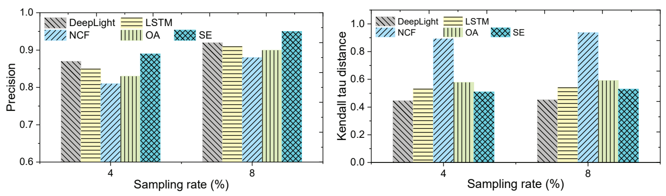
To show the impact of memory cap in the data plane on throughput, we increase the memory cap for in-network aggregation from 1MB to 2MB for SwitchML. We correspondingly double the number of hot parameters from 30K to 60K for Zebra. We use 16 workers for this set of experiments.

Figure 7 shows the results. Both Zebra and SwitchML benefit from increasing the switch memory cap. However, for Zebra, the improvement is limited: although the offloaded hot parameters are doubled, the improvement is 7% for OA and 1.7% for SE. The reason is that, as shown in Figure 8, the extra offloaded parameters contribute to a limited amount of updates that can be aggregated. SwitchML, on the other hand, is much sensitive to the memory cap: the throughput also doubles. This is because for SwitchML, a larger amount of on-chip memory lead to more parameters to be aggregated. Nevertheless, even with doubled memory usage, the aggregation throughput of SwitchML is only 45.6% of the throughput of Zebra with default configuration. These results show the importance of sparsity-awareness when aggregating gradients for distributed deep sparse training.

C. Precision of Hot Parameter Identification

For the offline identification mechanism, we run the training with a sampled small dataset (§III-C), and evaluate the precision in identifying the hot parameters. To this end, we first train the sparse models on our testbed with the whole datasets, and record the update frequency of each model parameter at the PS server side. By doing so, we get the global hot parameters. Specifically, we sort the model parameters in descending order according to the update frequency, and add each time 1,000 parameters to the hot parameter list

H_g from the highest rank to the lowest, till the increase of the cumulative update frequency falls below a pre-defined threshold (1% in our experiment). We then train the sparse model with the sampled datasets, and use the same method to get the hot parameter list H_s. Finally, we use $|H_g \cap H_s|/|H_g|$ to evaluate the precision. In addition, we use the popular Kendall tau distance [28] to further evaluate the rank precision of hot parameters.



(a) Precision of Hot parameter identification (b) Rank precision of hot parameters

Figure 9. Evaluation of the sampling-based method varying sampling rates.

Figure 9 shows the results. For the considered benchmark models, the precision exceeds 80% even with a sampling rate as low as 4%. Increasing the sampling rate to 8% improves the precision to over 90%. The Kendall tau distance between H_g and H_s is ∼0.5, and it is as high as 0.9 for NCF. These results demonstrate the high accuracy of identifying hot parameters using the sampling-based method.

Besides the sampling-based approach, we also propose the iteration-based online identification method that can capture the dynamics of the hot parameter set. However, if we frequently invoke this method to improve the precision of the hot parameter set, it will lead to significant performance overhead. Once invocation needs to sort the parameters gathered from the PS servers and switches, and notify them the results. This is a time-consuming task (e.g., *minutes*). Thus, an ideal case is to invoke this method only once, but still provide enough precision for improving the training performance.

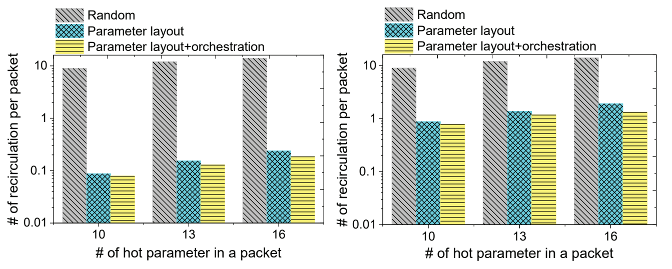
To confirm it, we train the sparse models on our testbed with the whole datasets, obtain hot parameter set after 10 iterations and compare it with that after finishing the whole training process (about 200 iterations). Table II shows the result. We can see the high accuracy relying only the first 5% iterations; invoking this method once can achieve more 69.1% precision. That said, the training in the remaining 95% iterations can benefit from the hot parameter set with high identification precision. Indeed, the identification precision will improve as the number of iterations increases, but the training in few iterations can be accelerated.

Table II
THE PRECISION OF ONLINE IDENTIFICATION OF HOT PARAMETERS AFTER TRAINING INDIVIDUAL MODELS FOR 10 ITERATIONS

DL Model	DeepLight	LSTM	NCF	OA	SE
Precision (%)	71.4	72.3	99.1	69.1	82.1

D. Evaluation of Hot Parameter Layout and Orchestration

We next evaluate the benefit of the parameter orchestration (see §III-D) in terms of the reduction of recirculations. In this set of experiments, we use the two benchmark models (OA and SE) from industry. We first design the parameter layout in registers and generate the packets accordingly at workers. We then count the number of recirculation operations needed on the switch and report the average number of recirculations for each packet. For the comparative purpose, we take a random parameter placement as the baseline.



(a) SE (b) OA

Figure 10. The number of recirculations per packet for the two benchmark models under different parameter orchestrations. The y-axis is in log scale.

Figure 10 shows the results, which demonstrate the effectiveness of our algorithm. In comparison, the random layout requires 10 recirculations per packet. The results also show the additional benefits of our gradient packaging method. We also notice that more recirculations are required in the OA model than the SE model. This is because the heat (*i.e.,* update frequency) distribution of hot parameters in OA is less biased than SE (see Figure 4).

E. Evaluation of Packet Loss Recovery

To enable packet loss recover, switch local CPUs are interleaved in the gradient packet processing (See §III-F). This may introduce extra delay. To evaluate this overhead, we trained the OA model in our testbed with packet loss rate varying from 0.01% to 0.1% as in [29]. Figure 11 shows performance loss with different packet loss rates, where the performance loss is measured by the increase of training time. We find that our packet loss recovery mechanism is practical, as even with 0.1% packet loss rate, the performance loss is under 3%.

We next perform some micro-benchmarks to evaluate the overhead introduced by the local CPUs for packet retransmissions. Specifically, we use the switch pipeline to label a specific aggregation packet, record its timestamp and forward it to the local CPUs. When received the packet, the local CPUs directly forward it back to the switch pipeline. Finally, the switch pipeline uses the packet arrival time to minus the prior recorded timestamp to calculate the additional latency. The experimental results show that the average latency is about 4.45*ms*, which is relatively low.

F. Resource consumption on Switches

Finally, we evaluate the resource consumption on the switch data plane. Zebra uses extra switch on-chip memory for gradient caching and float-pointing operation, and consumes the pipeline stages on switches. We deploy the OA benchmark model in Zebra, and use P4i, a visualization tool offered by Intel, to observe the resource consumption.

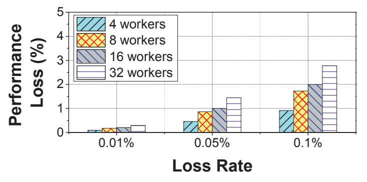
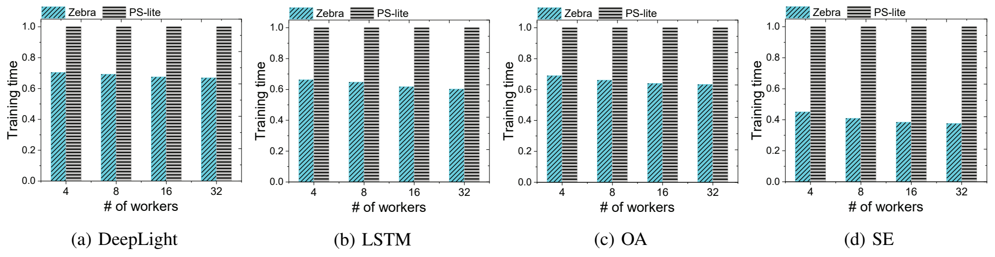


Figure 11. Performance loss under different packet loss rates.



(a) DeepLight (b) LSTM (c) OA (d) SE

Figure 12. Training time normalized by that of PS-lite.

Zebra consumes 9 pipeline stages in total. In case that Zebra affects the basic functionality of a switch (*e.g.,* packet forwarding), we can utilize one recirculation to halve the number of pipeline stages in use. Zebra also requires about 118KB for 30,000 hot parameters, 408.5KB for floating-point calculation, and 130KB for logic control. In total, Zebra uses 656.5KB on-chip memory—only 3.21% of the 20MB on-chip memory. Other types of resources include VLIW instructions (37 out of 384) and hash dist units (16 out of 32).

G. End-to-end Performance

Finally, we evaluate the end-to-end performance of Zebra in comparison with the PS-lite framework since SwitchML indeed does not work in sparse model training (see Section V-B). Specifically, we train sparse models on our testbed, and record the training time (time to reach a threshold of accuracy) by Zebra and PS-lite respectively. Figure 12 shows the results of 4 models. We can see that the end-to-end performance can be improved by 1.4$\sim$2.6$\times$ in comparison with the PS-lite.

VI. Related Work

In-network computation. With the rise of programmable hardwares, in-network computation emerges [35] for speed up data transmission. For example, NetCache [36] implements a key-value cache on programmable switches to process more than 2B queries/second; Jaqen [37] designs a switch-native approach for volumetric DDoS defense that can handle large-scale hybrid and dynamic attacks within seconds. IPSA [38] presents a new in-situ programmable switch architecture to facilitate in-network computing. ASK [39] enables in-network aggregation for key-value streams while NetReduce [40] presents transport-transparent primitives for in-network aggregation. Besides, SwitchML [13] uses programmable switches to aggregate gradient for accelerating DDL training while ATP [15] further supports multi-tenant training. However, the above approaches target dense models and they fall short when training sparse models. Instead, our Zebra target is to accelerate sparse model training.

Acceleration of distributed sparse DL training. NCCL [42], MPI [43] and Gloo [44] design high-performance collective communication libraries to accelerate distributed DL training; RDMA [45] is adopted to accelerate data transmission with an extra in-network support (*e.g.,* infiniband network). Other solutions reduce data transmission volume via gradient quantization or parameter synchronization. TernGrad [46] quantizes floating-point gradients into three numerical levels {-1,0,1}; Google proposes to shorten the width of each gradient to a 4-bit vector [47]. The DL model training can also be accelerated either through flow scheduling to minimize flow completion time [48], or through communication scheduling to decouple the dependence between gradients and change their transmission order [12] [8]. These works accelerate the training at the end host side, and are complementary to Zebra. Recently, Omnireduce [26] employs data sparsity to improve effective bandwidth utilization for sparse DL training; however, it does not consider the highly skewed distribution of update frequency of individual parameters. Nevertheless, incorporating Zebra with Omnireduce is worth further investigation.

VII. Discussion

Zebra scope. Actually, Zebra focuses on widespread high-dimensional sparse deep learning applications, such as online advertising and recommendation systems. For these applications, their training data and models are both sparse so that workers only synchronize non-zero updates during training.

Model parallelism. Indeed, Zebra focuses on data parallelism, and it may do not work in model parallelism because the model parameters in different workers are non-overlapped. That said, programmable switches cannot achieve in-network aggregation. However, we can utilize programmable switches to schedule training traffic for high efficiency.

VIII. Conclusion

This paper presents Zebra that performs in-network gradient aggregation in order to accelerate distributed sparse DL training. Overall, Zebra is motivated by the key observation on the highly skewed update frequencies of parameters from industrial sparse DL models—the *hot-cold* phenomenon. Based on this observation, Zebra offloads the aggregation of hot parameters from PS servers to programmable switches. To this end, we carefully design two methods for hot parameter identification, the parameter orchestration at both switches and workers, and also the reliability enhancement. We implemented Zebra on Intel Tofino switches and integrated it with PS-lite. Extensive experiments with different sparse models have shown the superior performance of Zebra. We believe Zebra paves the way towards the high-throughput distributed sparse DL system.

Acknowledgments

We thank our shepherd Wenfei Wu and the anonymous reviewers for their insightful feedback. This work is supported by National Key R&D Program of China (Grant No. 2022YFB2901300). Zhenyu Li is the corresponding author.

REFERENCES

[1] B. Jiang *et al.*, "Xdl: an industrial deep learning framework for high-dimensional sparse data," in *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019, pp. 1–9.

[2] S. Li and T. Hoefler, "Near-optimal sparse allreduce for distributed deep learning," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, p. 135–149.

[3] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.

[4] P.-S. Huang *et al.*, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, 2013, pp. 2333–2338.

[5] C. Zeng, X. Cheng, H. Tian, H. Wang, and K. Chen, "Herald: An embedding scheduler for distributed embedding model training," in *Proceedings of the 6th Asia-Pacific Workshop on Networking*, 2022, pp. 50–56.

[6] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *ACM/SIGDA FPGA 2015*, 2015, pp. 161–170.

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

[8] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," 2019.

[9] H. Pan, Z. Li, J. Dong, Z. Cao, T. Lan, D. Zhang, G. Tyson, and G. Xie, "Dissecting the communication latency in distributed deep sparse learning," in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20, 2020, p. 528–534.

[10] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," *Advances in Neural Information Processing Systems*, vol. 30, pp. 1709–1720, 2017.

[11] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

[12] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[13] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *NSDI*, 2021.

[14] Y. Li, I. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *ISCA 2019*, 2019.

[15] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift, "{ATP}: In-network aggregation for multi-tenant learning," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 741–761.

[16] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie, "Netfc: enabling accurate floating-point arithmetic on programmable switches," in *ICNP*, 2021.

[17] "Ps-lite platform," https://github.com/dmlc/ps-lite, 2021.

[18] "Tofino switch," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2021.

[19] "Zebra implementation," https://github.com/netsys-project/zebra, 2023.

[20] "Microsoft multiverso," https://github.com/Microsoft/multiverso/wiki, 2015.

[21] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: scalable deep learning on distributed gpus with a gpu-specialized parameter server," 2016.

[22] J. Xia *et al.*, "Rethinking transport layer design for distributed machine learning," in *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, 2019, pp. 22–28.

[23] T. Ge *et al.*, "Image matters: Jointly train advertising ctr model with image representation of ad and user behavior," *arXiv preprint arXiv:1711.06505*, 2017.

[24] G. Zhou *et al.*, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1059–1068.

[25] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *Acm transactions on interactive intelligent systems (tiis)*, vol. 5, no. 4, pp. 1–19, 2015.

[26] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 676–691.

[27] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin, "Deeplight: Deep lightweight feature interactions for accelerating ctr predictions in ad serving," in *Proceedings of the 14th ACM international conference on Web search and data mining*, 2021, pp. 922–930.

[28] "Kendall tau distance," https://www.wikiwand.com/en/Kendall_tau_distance, 2022.

[29] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *ACM SIGCOMM*, 2015.

[30] "lwip stack," http://savannah.nongnu.org/projects/lwip, 2018.

[31] "Data plane development kit," https://github.com/DPDK/, 2021.

[32] "ZeroMQ library," https://zeromq.org, 2021.

[33] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[34] "Click prediction dataset," shorturl.at/giOR9, 2023.

[35] Y. Tokusashi *et al.*, "The case for in-network computing on demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[37] Z. Liu *et al.*, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches," in $USENIX$ *Security 21*, 2021.

[38] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu, "Enabling in-situ programmability in network data plane: From architecture to language," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 635–649.

[39] Y. He, W. Wu, Y. Le, M. Liu, and C. Lao, "A generic service to provide in-network aggregation for key-value streams," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 33–47.

[40] S. Liu, Q. Wang, J. Zhang, W. Wu, Q. Lin, Y. Liu, M. Xu, M. Canini, R. C. Cheung, and J. He, "In-network aggregation with transport transparency for distributed training," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 376–391.

[41] N. Gebara, M. Ghobadi, and P. Costa, "In-network aggregation for shared machine learning clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 829–844, 2021.

[42] "Nvidia collective communications library." https://developer.nvidia.com/nccl, 2021.

[43] E. Gabriel *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[44] "Facebook, gloo." https://github.com/facebookincubator/gloo, 2021.

[45] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[46] W. Wen *et al.*, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in neural information processing systems*, 2017, pp. 1509–1519.

[47] A. T. Suresh *et al.*, "Distributed mean estimation with limited communication," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3329–3337.

[48] L. Mai, C. Hong, and P. Costa, "Optimizing network performance in distributed machine learning," in *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.