P4SGD: Programmable Switch Enhanced Model-Parallel Training on Generalized Linear Models on Distributed FPGAs

Hongjing Huang, Yingtao Li, Jie Sun, Xueying Zhu, Jie Zhang, Liang Luo, Jialin Li, Zeke Wang*

Abstract—Generalized linear models (GLMs) are a widely utilized family of machine learning models in real-world applications. As data size increases, it is essential to perform efficient distributed training for these models. However, existing systems for distributed training have a high cost for communication and often use large batch sizes to balance computation and communication, which negatively affects convergence. Therefore, we argue for an efficient distributed GLM training system that strives to achieve linear scalability, while keeping batch size reasonably low. As a start, we propose P4SGD, a distributed heterogeneous training system that efficiently trains GLMs through model parallelism between distributed FPGAs and through forward-communication-backward pipeline parallelism within an FPGA. Moreover, we propose a light-weight, latency-centric in-switch aggregation protocol to minimize the latency of the AllReduce operation between distributed FPGAs, powered by a programmable switch. As such, to our knowledge, P4SGD is the first solution that achieves almost linear scalability between distributed accelerators through model parallelism. We implement P4SGD on eight Xilinx U280 FPGAs and a Tofino P4 switch. Our experiments show P4SGD converges up to 6.5X faster than the state-of-the-art GPU counterpart.

Index Terms—FPGA, P4, GLMs, distributed training system.

✦

1 INTRODUCTION

MACHINE learning (ML) is a popular approach to accurately predict outcomes without needing to be explicitly programmed to do so. While most of the previous work focuses on accelerating deep neural network model training, generalized linear models (GLMs), such as linear regression, classification, and support vector machine [1], remain one of the most widely used models in the real world [2], [3]. Because the number of training samples has grown from the previously tens of thousands to tens of millions today [4], [5] and a single machine typically does not feature enough computing power and memory capacity to allow efficient GLM training on a large number of samples. Thus, people resort to distributed GLM training. *Data-parallel* and *model-parallel* are two of the prevailing parallel paradigms for distributed training. Both approaches consist of three stages: *forward propagation*, *backward propagation*, and *communication*, as shown in Figure 1.

Data Parallelism. Data parallelism horizontally partitions the input dataset across workers, i.e., accelerators. Within each iteration, each worker uses its local copy of the entire model ($\vec{w}$) to train on its subset of the dataset, as illustrated
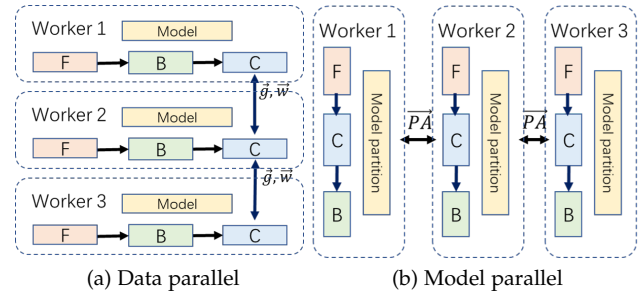

(a) Data parallel   (b) Model parallel

Fig. 1: Comparison of data-parallel and model-parallel training, F: forward propagation, B: backward propagation, C: communication, $\vec{g}$: gradient, $\vec{w}$: model, $\vec{PA}$: partial activations. Data (or model) parallelism needs to collectively communicate the whole gradient (or a mini-batch of activations) per iteration.

in Figure 1a. At end of each iteration, gradients of the model ($\vec{g}$) are averaged across all workers through `AllReduce` [6], [7] or parameter servers [8]–[10].

Model Parallelism. Model parallelism vertically partitions both the model $\vec{w}$ and the input dataset, such that each worker trains its model subset on its dataset subset, as shown in Figure 1b. Model parallelism requires synchronizing "partial activations ($\vec{PA}$)" between workers.

Both forward and backward passes in GLM training involve a low computation per weight ratio. Therefore, both parallelisms need to frequently synchronize the model (or partial activations), which significantly impacts performance when more accelerators, e.g., GPUs, are involved. When a model grows to millions of parameters, data parallelism requires transmitting millions of gradients or models

- Hongjing Huang, Yingtao Li, Jie Sun, Xueying Zhu, Jie Zhang, Zeke Wang are with Collaborative Innovation Center of Artificial Intelligence, Zhejiang University, China. E-mail: huang_hj, Li_Yingtao, jiesun, zhuxueying, carlzhang4, wangzeke@zju.edu.cn
- Liang Luo is with University of Washington. E-mail: liangluo@cs.washington.edu
- Jialin Li is with National University of Singapore, Singapore. E-mail: lijl@comp.nus.edu.sg
- Jie Sun is also with Alibaba group, China. E-mail: sunjie.sun@alibaba-inc.com

*: Corresponding author

through the network, resulting in significant communication overhead. In contrast, model parallelism avoids exchanging gradients and models and transmits only activations. This significantly reduces communication overhead and gives model parallelism higher potential than data parallelism when training a large GLM, such as a fully-connected layer, on multiple accelerators. [1] However, it still faces one severe issue.

Vanilla model-parallel training, i.e. basic model-parallel training, as shown in Figure 1b, requires an AllReduce operation to gather partial activations between forward and backward propagation. This exchange of partial activations creates a dependency that makes it impossible to overlap the forward and backward propagation. Therefore, model parallelism is more sensitive to both throughput and latency [12]–[15], different from data parallelism which is more sensitive to the aggregation throughput [16], [17]. It has become widely accepted that model parallelism can only achieve linear scalability between accelerators within a node [12]–[15] that are connected with high-bandwidth, low-latency NVLinks. Furthermore, due to limited network bandwidth and high network latency between nodes, strong scaling (linear speedup from using more accelerators to train a model under the same whole mini-batch size) with multiple accelerators in model parallelism remains challenging. In this paper, we ask:

Can we achieve strong scaling when training a large GLM through model parallelism between distributed accelerators?

To answer this question, as a start, we propose P4SGD, an efficient model-parallel training system that enables strong scaling when training a GLM.[2] P4SGD consists of a server implemented on a programmable switch and multiple workers implemented on FPGAs. P4SGD has three key innovations to address the above issue: efficient model-parallel training between distributed accelerators (**C1**), forward-communication-backward pipeline parallelism within an accelerator (**C2**), and ultra-low-latency aggregation between a P4 switch and FPGAs (**C3**).

C1: Efficient Model-parallel Training between Distributed Accelerators. Model parallelism works only well between accelerators, e.g., GPUs, within a node, which features high-bandwidth and low-latency links, e.g., NVLink, to connect these accelerators [12]–[14]. The key to scalability lies in ultra-low AllReduce latency and the overlap between the communication stage and forward/backward stages. To our knowledge, we are the first to propose P4SGD to achieve linear scale-out training on a GLM model, e.g., fully-connected layer, through model parallelism between distributed accelerators, e.g., FPGAs, which are connected via Ethernet.

C2: Forward-Communication-Backward. Pipeline parallelism within an FPGA. P4SGD exploits a new parallelism dimension, i.e., *forward-communication-backward pipeline parallelism*, to maximize compute efficiency. From a hardware

1. It is coincident with the observation from Krizhevsky [11], who proposes to train convolutional layers through data parallelism due to their high computation amount per weight, and to train fully-connected layers through model parallelism due to low computation amount per weight.

2. We will generalize P4SGD to Deep Learning (DL) model training in future work. We believe P4SGD has great potential on DL model training as well.

TABLE 1: Data parallelism (DP) vs. model parallelism (MP). D: model dimension, M: number of workers, S: number of samples, B: mini-batch size, BW: aggregation bandwidth between workers, T_{f_D}: forward propagation time of DP, T_{f_M}: forward propagation time of MP, T_l: aggregation latency, T_{b_D}: backward propagation time of DP, T_{b_M}: backward propagation time of MP, MB: micro-batch size.

	Model mem.	Dataset mem.	Network mem.	Iteration time
DP	D	$\frac{S \times D}{M}$	D	$T_{f\_D} + \frac{T_{b\_D}}{B} + \frac{D}{BW} + T_l$
Vanilla MP	$\frac{D}{M}$	$\frac{S \times D}{M}$	B	$T_{f\_M} + T_{b\_M} + \frac{B}{BW} + T_l$
P4SGD MP	$\frac{D}{M}$	$\frac{S \times D}{M}$	B	$\frac{MB}{B} * T_{f\_M} + T_{b\_M} + \frac{MB}{BW} + T_l$

perspective, P4SGD implements three distinct execution stages (i.e., forward propagation, communication, and backward propagation) with their own hardware resources, and thus organizes GLM training into three execution stages to allow pipelined execution between stages. From a software perspective, P4SGD divides each mini-batch of samples into multiple smaller *micro-batches*, which flow into three execution stages without suffering from any dependency on each other. As such, P4SGD can explore the overlap between forward/backward propagation and communication from different micro-batches, and also minimize communication overhead from model-parallel training.

C3: Latency-centric In-switch Aggregation Protocol. We design and implement a light-weight, fault-tolerant, *latency-centric* in-switch aggregation mechanism that directly interplays between a P4 switch and distributed FPGAs to minimize the latency of `AllReduce` needed by each training iteration. Such an aggregation protocol needs a careful interplay between distributed FPGAs and a P4 switch to recover from potential packet loss while achieving ultra low, stable AllReduce latency due to pure hardware implementation.

We implement the workers of P4SGD on up to eight Xilinx U280 FPGA boards [18] and the server on a Tofino P4 switch [19]. The experimental results show that 1) P4SGD under model parallelism achieves strong scaling between distributed FPGA-based accelerators; 2) P4SGD converges faster than its corresponding data-parallel counterpart; and 3) P4SGD converges up to 9.3X faster than the state-of-the-art distributed training systems on distributed GPUs that are not able to achieve linear scale-out training mainly due to its high inter-GPU communication overhead.

2 BACKGROUND: PARALLELING SGD HARDWARE

In this section, we briefly discuss the interesting properties of the hardware implementation of data-parallel and model-parallel training on GLMs. The stochastic gradient descent (SGD) hardware through either model or data parallelism consists of three stages: forward propagation, backward propagation, and communication. All three stages, which have their own hardware resources to implement, allow to explore more parallelism. Table 1 illustrates the comparison result between data parallelism ("DP") and model parallelism ("Vanilla MP").

2.1 Data-Parallel Training

Data parallelism horizontally partitions the input dataset across workers, as shown in Figure 1a. Each worker maintains a local copy of the model and trains on its own

partition of the input dataset. During each iteration, each worker goes through three steps, as shown in Figure 2a. First, it computes the dot product of the updated model and each sample in the current mini-batch (forward propagation). Second, it uses the dot products from the mini-batch to compute the gradient (backward propagation). Third, it synchronizes its gradient with other workers and computes the updated model via either collective primitive `AllReduce` or parameter server [8]–[10]. The SGD hardware has custom hardware resources for both forward and backward passes, so after we perform a forward pass on a sample, the backward pass can be started immediately without waiting for the forward results of other samples in the same mini-batch. Therefore, forward and backward passes have no dependency between samples in the same mini-batch.

Estimating Elapsed Time per Epoch. Since the three stages have their own forward-communication-backward pipeline, they can execute concurrently when there is no dependence between them. In particular, each time the forward propagation computes the loss of a sample, the backward propagation of this sample can be started immediately, so these two stages allow concurrent execution, but still have a dependency on the third stage communication. Therefore, time T_{it} per iteration is estimated to be forward propagation time of DP T_{f_D} plus backward propagation time of DP for a sample $\frac{T_{b\_D}}{B}$ plus communication time T_c, as shown in Equation 1.

$$T_{it} = T_{f\_D} + \frac{T_{b\_D}}{B} + T_c = T_{f\_D} + \frac{T_{b\_D}}{B} + \frac{D}{BW} + T_l, \quad (1)$$

where D is the model dimension, BW is the aggregation bandwidth between workers, and T_l is the aggregation latency.

2.2 Model-Parallel Training

Model parallelism (MP) vertically partitions the model across workers, as shown in Figure 1b. Each worker maintains a partition of the model and trains on its own partition of the input dataset. During each iteration, each worker goes through three steps, as shown in Figure 2b. First, it computes the partial dot product of the partial model and each partial sample in the current mini-batch (forward propagation). Second, it synchronizes its partial dot products from the mini-batch with other workers and computes the full dot product via collective communication primitive `AllReduce`. Therefore, the amount of data to be exchanged is B, which is the batch size. Third, each worker uses the full dot products to compute its gradient portion (backward propagation).

Estimating Elapsed Time per Epoch. Since three stages have the dependency regarding the model, their executions are serialized. Therefore, T_{it} is estimated to be forward propagation time of MP T_{f_M} plus backward propagation time of MP T_{b_M} plus communication time T_c, as shown in Equation 2.

$$T_{it} = T_{f\_M} + T_{b\_M} + T_c = T_{f\_M} + T_{b\_M} + \frac{B}{BW} + T_l \quad (2)$$

2.3 Data- vs. Model-Parallel Training

We compare the scale-out potential between data-parallel training with model-parallel training on M workers. From Equations 1 and 2, intuitively, we observe that data parallelism allows the overlap between forward propagation and backward propagation, but needs to communicate the whole gradient during each iteration. Model parallelism only needs to communicate B elements per iteration, but cannot overlap forward and backward propagation due to the dependency. In the following, we discuss their potential on scale-out training.

Potential of Data Parallelism on Scale-out Training. Data parallelism spans a mini-batch of samples to M workers, such that each worker only needs to work on $\frac{B}{M}$ samples, indicating that forward propagation time can be reduced by M times. However, its communication time stays the same, because the amount of data to be exchanged stays the same. However, in practice, its communication time would increase. A larger M could easily make data parallelism communication-bound, especially on GLMs that have a relatively low amount of computation per weight. Another drawback is that data parallelism fails to support a large model that exceeds the memory capacity of a worker.

Potential of Model Parallelism on Scale-out Training. Model parallelism spans the model and the dataset to M workers, such that each worker only has a partition of the model and forward and backward propagation time can be reduced by M times. Moreover, model parallelism only needs to exchange B elements with other workers, incurring negligible overhead.[3] Therefore, model parallelism can easily scale out to support M workers for training.

3 SYSTEM OVERVIEW OF P4SGD

3.1 Design Goals and Overall Architecture

When designing P4SGD, we keep three goals in mind.

G1: Allowing Efficient Scale-out Training. As the model size of GLM is ever increasing, it is not appropriate to train the model on a single worker that has limited memory and compute capacities. Therefore, it is natural to employ multiple workers to concurrently train the same model. However, it is challenging to achieve linear scalability, since GLM has low computation amount per weight, making communication overhead difficult to amortize on CPUs/GPUs.

G2: Maximizing the Overlap between Forward and Backward Propagation Computation. The SGD hardware through vanilla model parallelism allows no overlap between forward and backward propagation due to its inherent dependency, resulting in low utilization of computing resources. Therefore, maximizing the overlap between forward and backward propagation could significantly reduce computation time and then improve the efficiency of model-parallel training.

G3: Minimizing Communication Latency. Even the network traffic needed by model-parallel training is B elements per iteration, network latency T_l could still impede linear scalability, especially when M is large. This is because M workers lead to M times less computation time per iteration,

---

3. Even though the latency regarding B elements is small, the basic network latency T_l is still large.

(a) Data-parallel training

(b) Vanilla mini-batch model-parallel training

(c) Micro-batch forward-communication-backward pipeline-parallel training within a worker
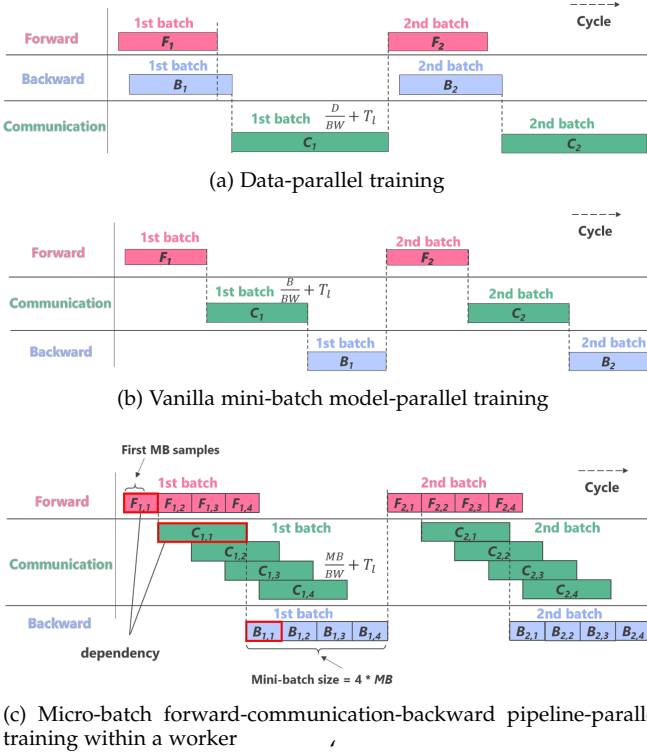
Fig. 2: Comparison of data and model parallelism for training GLMs implemented in hardware, motivating forward-communication-backward pipeline parallelism. $F_{i,j}$: forward propagation of the j-th micro-batch of the i-th mini-batch, $B_{i,j}$: backward propagation of the j-th micro-batch of the i-th mini-batch, C: communication, MB: micro-batch size, BW: network bandwidth, T_l: fixed latency of network

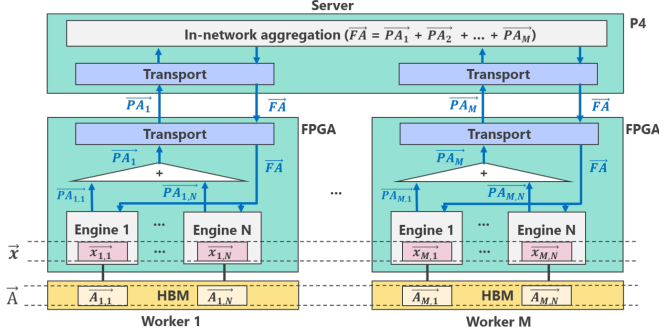Fig. 3: P4SGD consists of M workers and one server, and each worker has N engines. P4SGD realizes model parallelism between distributed workers, and forward-communication-backward pipeline parallelism within an engine in a worker. As such, P4SGD is the first to achieve linear scalability between distributed accelerators.

indicating a larger proportion of time spent on communication. Therefore, minimizing the latency of communication primitive AllReduce among workers could significantly benefit scale-out training.

Overall Architecture. To achieve the above three goals, we present P4SGD, a P4-switch-enhanced hardware training system that allows efficient scale-out GLM training through model parallelism on distributed FPGAs. In particular, P4SGD consists of M FPGA-based workers and one P4 switch enhanced server, as shown in Figure 3. All the

---

Algorithm 1: MODEL PARALLEL TRAINING

Define : E: number of epochs,
 M: number of workers,
 $\vec{A_i}[j]$: the j-th sample's partition in the i-th worker,
 PA_m: partial activation vector in the m-th worker,
 b_i: label value of the i-th sample,
 $\vec{x_i}$: the partial model vector in the i-th worker,
 γ: learning rate.

1 **P4-Switch-based Server:**
2 Issue $start$ to all workers;
3 **for** $e = 1$ **to** E **do**
4 **for** $(i = 0; i < S; i+ = MB)$ **do**
5   Pull $\overrightarrow{PA_1}, ..., \overrightarrow{PA_M}$ from all the workers;
6   $\overrightarrow{FA} = \overrightarrow{PA_1} + ... + \overrightarrow{PA_M}$;
7   Push $\overrightarrow{FA}$ to all the workers;
8 **end**
9 **end**
10 **FPGA-based worker m (1, ..., M):**
11 Load the m-th partition $\overrightarrow{a_m}$ of the training dataset $\vec{a}$;
12 Init the m-th partition $\overrightarrow{x_m}$ of the model $\vec{x}$;
13 **for** $e = 1$ **to** E **do**
14 **for** $(i = 0; i < S; i+ = B)$ **do**
15   $\overrightarrow{g_m} = 0;$ /* Zero the partial gradient */
16   **for** $(j = 0; j < B; j+ = MB)$ **do**
    /* Stage 1: forward propagation */
17   #pragma parallel in hardware
18   **for** $(k = 0; k < MB; k + +)$ **do**
19   int32 $t = i + j + k;$
20   int32 $\overrightarrow{PA_m}[k] = \overrightarrow{A_m}[t] \cdot \overrightarrow{x_m};$
21   **end**
    /* Stage 2: communication */
22   Push the partial activation vector $\overrightarrow{PA_m}$ to the server;
23   Pull the full activation vector $\overrightarrow{FA}$ from the server;
    /* Stage 3: backward propagation */
24   #pragma parallel in hardware
25   **for** $(k = 0; k < MB; k + +)$ **do**
26   int32 $t = i + j + k;$
27   int32 $\overrightarrow{scale}[k] = \gamma \times df(\overrightarrow{FA}[k], b_{i+k});$
28     $\overrightarrow{g_m} + = scale[k] \times \overrightarrow{A_m}[t];$
29   **end**
30   **end**
31   $\overrightarrow{x_m} = \overrightarrow{x_m} - \overrightarrow{g_m}/B;$
32 **end**
33 **end**

M workers go through each iteration in a lock step, to efficiently train on the same model $\vec{x}$. Each worker trains on a subset of the model over a subset of the dataset via model parallelism. During each iteration, the m-th worker that trains on the model subset $\overrightarrow{x_m}$ sends outs a network packet containing its partial aggregation $\overrightarrow{PA_m}$ to the P4-switch-based server, which aggregates them and then broadcasts a network packet containing $\overrightarrow{FA}$ to all the workers for further backward propagation computing.

One key idea of P4SGD is to divide each mini-batch of samples into multiple smaller micro-batches (**G2**), such that the SGD hardware can explore inter-micro-batch (i.e., intra-mini-batch) parallelism to maximize the overlap between forward/backward propagation and communication, while still preserving the precedence for synchronous SGD (Subsection 3.2), as shown in algorithm 1. Another key idea is light-weight, fault-tolerant, latency-centric in-switch aggregation (**G3**) that directly interplays between a P4 switch and distributed FPGAs to minimize the latency of AllReduce (Subsection 3.3). At the same time, the communication cost reduces, which can greatly increase the scalability of P4SGD (**G1**).

3.2 Architecture of an FPGA-based Worker

Algorithm 1 illustrates the detailed flow of the m-th FPGA-based worker under P4SGD. In the beginning, it loads its partition of dataset $\overrightarrow{a_m}$ (Line 11) and initiates its partition of model $\overrightarrow{x_m}$ (Line 12). The algorithm is iterated in E epochs (Line 13). In each epoch, its partition of the input dataset is scanned, one mini-batch of B samples within an iteration (Line 14). Within a mini-batch, we zero the partial gradient $\overrightarrow{g_m}$ to 0 (Line 15). Then, it keeps accumulating the gradient $\overrightarrow{g_m}$ from a micro-batch (MB) of samples, as illustrated in the following three stages.

Stage 1: Forward Propagation. The m-th worker reads MB partial samples and its partial model $\overrightarrow{x_m}$, and computes *partial activations* ($\overrightarrow{PA_m}$) of MB elements (Lines 17-21).

Stage 2: Communication. The m-th worker sends $\overrightarrow{PA_m}$ to the server for aggregation, with the payload of MB elements. Then, it waits for the corresponding *full activation* ($\overrightarrow{FA}$) for the future backward propagation (Lines 22-23).

Stage 3: Backward Propagation. The m-th worker computes the partial gradient from the micro-batch of samples and accumulated it to the partial model $\overrightarrow{x_m}$ (Lines 25-29). Then, the partial model $\overrightarrow{x_m}$ is updated with the average gradient $\overrightarrow{g_m}$.

3.2.1 Parallelism Analysis

Figure 2c illustrates the dependency of three stages in the micro-batch pipeline-parallel training within a worker. After the first micro-batch $F_{1,1}$ of the first mini-batch finishes forward propagation computation, it directly enters the communication stage by launching a network collective operation within a micro-batch, without waiting for the other micro-batches to finish. When the first batch $B_{1,1}$ receives the corresponding full activation vector from the server, it can directly enter the third stage. We observe that there is no dependency between micro-batches (e.g., $F_{1,1}$ and $F_{1,2}$) within the same mini-batch, we can overlap communication and forward/backward propagation computation within the mini-batch.

Discussion. One potential limitation is that training on a small micro-batch (e.g., 8) of samples at a time would reduce computing parallelism and then under-utilize compute power and network bandwidth on modern CPUs/GPUs, which rely on software implementations. However, P4SGD that relies on pure hardware implementation, e.g., FPGA and P4, can more tolerate small micro-batch training, without sacrificing computing throughput.

3.2.2 Estimating Time per Iteration under P4SGD

P4SGD enables the overlap between forward and backward propagation computation between micro-batches, as shown in Figure 2c, so we estimate the elapsed time T_{it} per iteration to be forward propagation time $\frac{BW}{B} \times T_{f\_M}$ of a micro-batch plus communication time T_c plus backward propagation time T_{b_M} of a mini-batch, as shown in Equation 3. Compared with the mini-batch model-parallel training shown in Figure 2b, the proposed micro-batch training is able to maximize the overlap between forward and backward propagation.

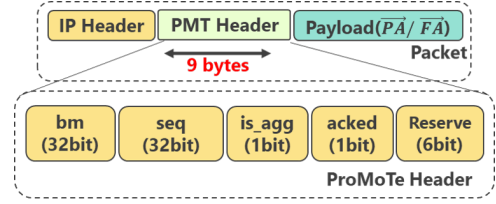$$T_{it} = \frac{MB}{B} \times T_{f\_M} + T_{b\_M} + \frac{MB}{BW} + T_l \qquad (3)$$

Fig. 4: P4SGD Packet Format

3.3 P4-Switch-FPGA Collective Operation

According to the above analysis, communication latency T_c is critical for the overall training time. Furthermore, P4SGD needs to perform an *AllReduce* operation on MB elements per iteration, where MB is typically small during our training. Therefore, P4SGD requires extremely low latency of an *AllReduce* operation on a small payload.

Aggregating partial activations in network switches offers two major latency benefits. First, as the switch locates inside the network, our approach avoids the additional network hops to and from the aggregation server, essentially reducing activation aggregation to "sub-RTT (Round Trip Time)" latency. Second, the switch data plane is designed specifically for fast and predictable packet processing. Commercial network switches, including new generation programmable switches [20]–[23], can consistently process packets under a few hundred nanoseconds – significantly lower even compared to servers with RDMA or kernel-bypassed networking.

However, traditional in-switch aggregation approaches [16], [17], need end-host servers to prepare packets before in-network aggregation, and thus, unfortunately, incur high communication latency due to unstable software processing overhead and long PCIe latency. Moreover, these approaches introduce a shadow copy mechanism to optimize for high throughput when performing AllReduce on a large message, so these approaches are not friendly to latency due to their late acknowledgement. To this end, we design and implement a light-weight, fault-tolerant, **latency-centric** in-switch aggregation that directly interplays between a P4 switch and distributed FPGAs to minimize the latency of AllReduce needed by each training iteration.

In P4SGD, we handle packet drops by an FPGA-centric retransmission mechanism. Compared to the approach taken by SwitchML [16][4], our solution eliminates the need for shadow copies of the aggregation results on the switch, and significantly reduces switch resource usage. SwitchML can support half as many outstanding aggregation operations as our approach under the same resource budget.

To address the issue of losing aggregation results due to dropped packets (which contain $\overrightarrow{FA}$), our key idea is to use a second round of communication – initiated and retransmitted by the workers – to acknowledge the reception of the full activation vector. Only when the switch receives acknowledgements from all workers for an operation, it can safely clear the aggregation result.

Specifically, as shown in algorithm 2, the switch only maintains one copy of the aggregated activations, agg. To

---

4. SwitchML is a widely-used in-switch aggregation method for distributed training. It adopts the shadow copy mechanism to optimize for throughput.

separately track the number of received activations and acknowledgement, the switch stores two sets of counters, agg_count and ack_count. Since the workers retransmit packets in the event of packet drops, two sets of bitmaps, agg_bm and ack_bm, are maintained to detect duplicates. We also augment the P4SGD packet header with additional fields as shown in Figure 4: bm is a bitmap with the source worker's index set to one; seq is the aggregation slot index in the P4 switch; is_agg indicates whether the packet is for aggregation or acknowledgement; $acked$ is a placeholder for the switch to signal that it has received all acknowledgement for the operation.

Algorithm 2: SWITCH AGGREGATION LOGIC WITH UNRELIABLE TRANSMISSION HANDLING

Initialize : N: number of aggregation slots
W: number of workers
$\overrightarrow{agg}[N] := \{\vec{0}\}$
$agg\_count[N], agg\_bm[N] := \{0\}$
$ack\_count[N], ack\_bm[N] := \{0\}$

1 **receive** $pkt(is\_agg, seq, bm, \overrightarrow{PA}, \overrightarrow{FA})$:
2   **if** is_agg **then**
3   **if** $agg\_bm[seq] \& bm = 0$ **then**
4        $agg\_count[seq] \leftarrow agg\_count[seq] + 1$;
5        $agg\_bm[seq] \leftarrow agg\_bm[seq] | bm$;
6        $\overrightarrow{agg}[seq] \leftarrow \overrightarrow{agg}[seq] + pkt.\overrightarrow{PA}$;
7   **if** $agg\_count[seq] = W$ **then**
8          $ack\_count[seq] \leftarrow 0$;
9          $ack\_bm[seq] \leftarrow 0$;
10   **end**
11   **end**
12   **if** $agg\_count[seq] = W$ **then**
13        $pkt.\overrightarrow{FA} \leftarrow \overrightarrow{agg}[seq]$;
14   forward pkt to all workers;
15   **end**
16   **end**
17   **else**
18   **if** $ack\_bm[seq] \& bm = 0$ **then**
19        $ack\_count[seq] \leftarrow ack\_count[seq] + 1$;
20        $ack\_bm[seq] \leftarrow ack\_bm[seq] | bm$;
21   **if** $ack\_count[seq] = W$ **then**
22          $agg\_count[seq] \leftarrow 0$;
23          $agg\_bm[seq] \leftarrow 0$;
24          $\overrightarrow{agg}[seq] \leftarrow \vec{0}$;
25   **end**
26   **end**
27   **if** $ack\_count[seq] = W$ **then**
28   forward pkt to all workers;
29   **end**
30   **end**
31 **end**

Algorithm 3: WORKER-SIDE AGGREGATION LOGIC WITH UNRELIABLE TRANSMISSION HANDLING

Initialize : N: number of aggregation slots
$unused[N] := \{\text{true}\}$
$seq := 0, bm := \text{WORKER\_INDEX}$

1 **send** $pa\_pkt(\overrightarrow{PA})$:
2   **if** $unused[seq]$ **then**
3      $unused[seq] \leftarrow$ false;
4      $pkt.seq \leftarrow seq; pkt.\overrightarrow{PA} \leftarrow \overrightarrow{PA}$;
5      $pkt.bm \leftarrow bm; pkt.is\_agg \leftarrow$ true;
6      $seq \leftarrow seq + 1$;
7   **if** $seq = N$ **then**
8        $seq \leftarrow 0$;
9   **end**
10   forward pkt to switch;
11   start_timer(pkt);
12   return true;
13   **end**
14   **else**
15   return false;
16   **end**
17 **end**
18 **recvive** $pkt(is\_agg, seq, bm, \overrightarrow{PA}, \overrightarrow{FA})$:
19   **if** $pkt.is\_agg$ **then**
20   cancel_timer(pkt);
21   forward $pkt.\overrightarrow{FA}$ to backward propagation;
22      $pkt.is\_agg \leftarrow$ false; $pkt.bm \leftarrow bm$;
23   forward pkt to switch;
24   start_timer(pkt);
25   **end**
26   **else**
27      $unused[pkt.seq] \leftarrow$ true;
28   cancel_timer(pkt);
29   **end**
30 **end**
31 **upon timeout** (pkt):
32   forward pkt to switch;
33   start_timer(pkt);
34 **end**

---

4 IMPLEMENTATION OF P4SGD

We implement P4SGD with one P4-switch-based server and M FPGA-based workers. In the following, we present the implementation details of an FPGA-based worker and P4-switch-based server, as shown in Figure 5.

4.1 Hardware Design of a Worker

The goal of the FPGA-based worker is three-fold. First, it maximizes the processing parallelism (**E1**), which needs not only high processing ability but also high memory bandwidth required by GLM training. Second, it enables micro-batch pipeline-parallel training to overlap forward and backward propagation computation to increase computing pipeline utilization (**E2**). Third, it needs to overlap communication and forward/backward propagation computation to amortize the negative effect of inter-FPGA communication (**E3**).

We implement each worker with verilog language on an HBM-equipped FPGA board Xilinx Alveo U280 [18]. In order to achieve **E1**, we adopt a multi-engine design to partition a subset of the model assigned to this worker uniformly to N engines, where N is parameterized at compile time. As such, all the engines train in a lock step and each engine only needs to train a small portion of the subset. For example, $\overrightarrow{x_{1,2}}$ corresponds to the model portion associated with the

We augment the worker protocol accordingly (algorithm 3). When sending partial activation to the switch, the worker adds its node bitmap to the packet, and indicates that the packet is intended for aggregation (Line 5). Once the worker receives a full activation, it sends an acknowledgement to the switch (Line 22-23). However, it only re-enables the slot for aggregation after it receives an acknowledgement confirmation from the switch (Line 26-29). To handle packet drops, the worker starts a timer after sending each packet (Line 11, 24), and retransmits the packet after the timer expires (Line 31-34). A timer is canceled once the worker receives the corresponding full activation (Line 20) or acknowledgement confirmation (Line 28).

(a) Detailed design of the n-th engine in the m-th worker

(b) Detailed design of the k-th bank in the n-th engine in the m-th worker, allowing forward-communication-backward pipeline parallelism
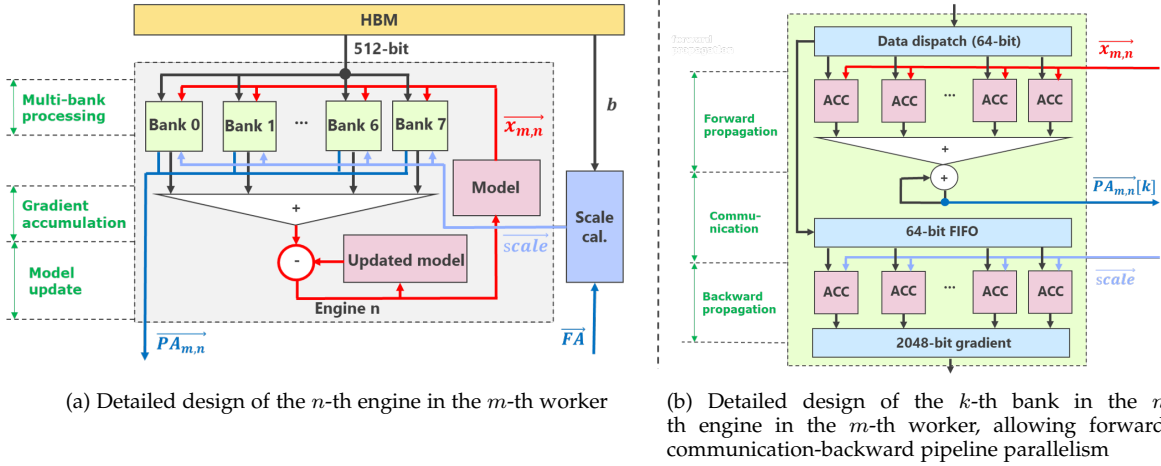
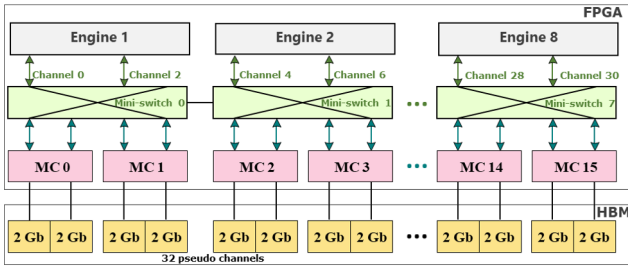Fig. 5: Each worker of P4SGD has N engines. Each engine has 8 banks to populate a micro-batch of 8 samples.

Fig. 6: Relationship between 16 HBM channels and 8 engines, each engine occupies 4 HBM pseudo channels.

second engine of the first worker.[5] Each model portion, accommodating up to 256K weights,[6] is implemented with on-chip memory, so each worker supports 2M weights. Therefore, each worker consists of an HBM subsystem (subsection 4.1.1), N engines (subsection 4.1.2, 4.1.3), as shown in Figure 3.

4.1.1 Hardware Design of Memory Subsystem

The memory subsystem employs the HBM subsystem that has 32 memory channels to support 32 independent 256-bit memory accesses. Due to FPGA resource limitations, an FPGA board can only afford to instantiate up to N=8 engines, each engine can occupy four consecutive HBM channels to provide sufficient memory bandwidth and capacity (8Gb) for its subset of the dataset. Figure 6 illustrates the relationship between 32 HBM channels and 8 engines. Since each engine is designed to process 512-bit data stream per cycle, each engine combines two 256-bit AXI interfaces to access the data in the HBM.

4.1.2 Hardware Design of an Engine

Built on the state-of-the-art GLM accelerator MLWeaving [24] that supports efficient low-precision training, each engine extends to enable GLM training on distributed FPGAs in a lock step. To achieve **E2** in section 4.1, we adopt a

5. We also vertically partition the dataset $\vec{A}$ in the same way.
6. This number is parameterizable at compile time, under the constraint of FPGA resource limitation. However, we can easily generalize P4SGD to support a large model that is stored in external memory, e.g., HBM, without affecting performance.
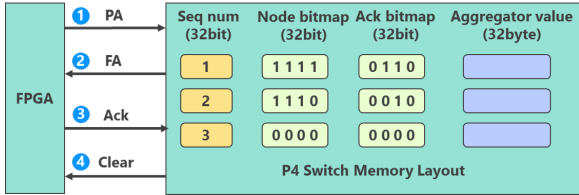
multi-bank design to allow each engine to train on a micro-batch of MB=8 samples concurrently, one bank to populate a sample. As such, MB samples flow into the computing pipeline concurrently.

The hardware design of each engine consists of three stages: "multi-bank dividing", "gradient accumulation", and "model update". In the "multi-bank dividing" stage, the 512-bit input data stream flows into 8 banks, each of which consumes 64-bit data from the same sample, one bit from one feature of a sample. We leave the detailed design of each in Subsection 4.1.3. The output of each bank is a 2048-bit gradient (i.e., 64 32-bit gradient elements from 64 features). In the "gradient accumulation" stage, we instantiate 64 8-element-wise adder trees to aggregate gradients from all 8 banks. Unlike MLWeaving, we use DSP instead of LUT to construct the adder tree. Each level of the adder tree can add three numbers, which saves resources and reduces calculating time. In the "model update" stage, the aggregated gradient from the above MB=8 samples is accumulated into the "updated model". When accumulating the gradient from the last micro-batch of each mini-batch into the "updated model", we also update the architectural model ($\vec{x}$) with the same value written into the "updated model".

4.1.3 Hardware Design of a Bank

Figure 5b shows the details of each bank. According to the distributed SGD algorithm, each bank consists of three stages. To achieve **E3** in section 4.1, we send "partial activations" of MB samples each time to support micro-batch model-parallel training on GLM in the "communication" stage.

The hardware design of each bank consists of three stages: "forward propagation", "communication", and "backward propagation". In the "forward propagation" stage, we instantiate 64 bit-serial multipliers [25], [26] to consume 64-bit data stream, a bit from each feature. At the same time, the 64-bit data stream is also fed into the 64-bit "FIFO". Each bit-serial multiplier outputs a 32-bit calculation result, which is directly fed into the full-pipelined adder tree. The output of the adder tree feeds to an accumulator, which aggregates the corresponding "partial activation". In the "communication" stage, together with "partial activations" from the other banks in the same engine, the n-th engine

Fig. 7: Interaction between P4 switch and FPGA

TABLE 2: Evaluated datasets

Dataset	Samples	Features	Classes
gisette [27]	6,000	5,000	2
real_sim[7]	72,309	20,958	2
rcv1[7]	20,242	47,236	2
amazon_fashion[8]	200,000	332,710	5
avazu[7]	40,428,967	1,000,000	2

prepares $\overrightarrow{PA_{m,n}}$ for the current micro-batch of MB samples. The m-th worker then aggregates "partial activations" from all its N engines to produce $\overrightarrow{PA_m}$, which is sent to the P4-switch-based server for in-network aggregation in Figure 3. In the "backward propagation" stage, we also instantiate 64 bit-serial multipliers, each of which reads a 1-bit feature from the 64-bit "FIFO" and multiplies it with the corresponding 32-bit element $\overrightarrow{scale}[k]$ per cycle. Therefore, 64 bit-serial multipliers is able to produce 2048-bit gradient computation results per cycle for further processing in the corresponding engine (Subsection 4.1.2).

4.2 Implementation of P4-switch-based Server

Figure 7 briefly describes the hardware implementation of reliable transport between P4 switch and an FPGA.

Our switch data plane is implemented in the P4 language. Each of the agg, agg_count, agg_bm, ack_count, and ack_bm in algorithm 2 is mapped directly to a Tofino register array, as shown in Figure 7. In our current configuration, the size of the register arrays is set to 64K (16 bits indices), permitting a maximum of 64K outstanding aggregation operations. The register arrays are distributed over 4 stages of a switch pipeline – out of the 12 total stages available. Resource consumption in any of the 4 stages is capped at 70.83% of the available SRAM. This leaves ample resources for the other bread-and-butter switching functionalities. We leverage the Tofino packet replication engine to implement multicasting to workers.

5 EVALUATION

5.1 Experimental Setting

Workloads. We conduct our experiments on five datasets with the various number of features, as shown in Table 2. All datasets used are publicly accessible.

Experimental Platform. We run our experiments on a cluster with two network switches and eight machines. One network switch is Mellanox SN2700 with 32 x QSFP28 ports, and the other one is the Wedge100BF-32X P4 reconfigurable switch which provides 32 x QSFP28 ports and a 20 MB packet buffer. Each machine configured with a 12-core/24-thread Intel Xeon(R) Silver 4214 CPU (2.2GHz), a Xilinx

7. https://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/
8. This is part of the indian news dataset, and the dataset can be downloaded from https://jmcauley.ucsd.edu/data/amazon/

TABLE 3: Resource consumption of a worker with 8 engines

Hardware modules	LUTs	REGs	RAMs	DSPs	Freq.
PCI-Express	63K	98K	4.3Mb	0	250MHz
Network transport	10K	27K	3.5Mb	0	250MHz
HBM subsystem	7K	42K	3.26Mb	0	450MHz
8 engines	188K	904K	152Mb	4096	250MHz
Total utilization	304K (23%)	1.1M (42%)	165Mb (47.5%)	4096 (45%)	

Alveo U280 FPGA [18], a 100Gb/s (RDMA-enabled) Mellanox MT27800 NIC and a NVIDIA A100 GPU (40 GB HBM2 memory, 6912 CUDA cores). We use this platform for all the experiments.

P4SGD Implementations. We implement P4SGD with a P4-switch-based server and FPGA-based workers. Table 3 shows the resource consumption of P4SGD with 8 engines on Xilinx Alveo U280. The total resource utilization is about 50%. P4SGD allows instantiating a flexible number of engines from 1 to 8. Furthermore, we also implement the data-parallel training system that also leverages a P4 switch to do in-network aggregation between distributed FPGAs. The data-parallel system aggregates gradients of length D, rather than B, after forward and backward propagation within an iteration. The adopted precision is 4 bits because 1) the execution time will decrease linearly as the precision decreases, and 2) MLWeaving [24] demonstrates that low-precision (above 3 bits) training takes a similar number of epochs to converge as that of full-precision CPU approaches. So we choose 4-bits precision to decrease the training time without affecting convergence rate.

GPU Baseline. The GPU baseline, labeled "GPUSync", is implemented on the GPUs in the cluster adopting a synchronous distributed linear model SGD. "GPUSync" leverages the state-of-the-art cuBLAS library [28] to efficiently implement forward and backward propagation. In particular, "GPUSync" uses the function *cublasSgemm*, which does auto parallelization within a single GPU. From the NVIDIA Nsight Systems, we observe that *cublasSgemm* uses at least 512 thread blocks, 128 threads per thread block, to compute on a mini-batch of samples in our experiment, indicating that "GPUSync" has already fully utilized GPU computing power within a CUDA call. Moreover, we leverage a few optimization methods on GPU to accelerate the training: CUDA Graphs [29] to reduce kernel invocation overhead, and RDMA+GPUDirect-enabled NCCL [30], [31] to reduce inter-node communication overhead. "GPUSync" adopts model-parallel training, which is obviously faster than data-parallel training (Subsection 5.3). However, "GPUSync" cannot efficiently scale out due to its CUDA kernel invocation overhead, in particular, each training iteration needs to launch three CUDA kernels: two *cublasSgemm* for forward/backward passes and one AllReduce for communication. When employing more GPUs, the GPU compute cycles per kernel are reduced and kernel invocation overhead can dominates the overall time.

Two CPU Baselines. We implement the synchronous SGD algorithm on distributed CPUs, labeled "CPUSync". We employ the following optimization methods on distributed CPUs: multi-core (12 cores), AVX2 instruction (512-bit) and RDMA-based openMPI (version 3.4.1) library. "CPUSync" adopts model-parallel training, which is obviously faster than data-parallel training in our experiment. The other

CPU baseline is "SwitchML" that adopts the same computation method as "CPUSync", but uses the communication method from SwitchML [16], rather than RDMA-based OpenMPI.

Comparison Methodology. Our evaluations mainly validate three hypotheses. First, P4-switch-FPGA in-network aggregation can significantly reduce the aggregation latency, benefiting model-parallel training on GLMs. Second, P4SGD is able to achieve an almost linear scale-out on distributed FPGAs, with the help of the P4 switch. Third, P4SGD converges faster and consumes lower than its counterparts on distributed GPUs/CPUs.

5.2 Comparison of Aggregation Latency

We measure the latency of P4-switch-FPGA in-network aggregation that serves distributed GLM training on FPGAs (Subsection 3.3). Figure 8 illustrates the latency comparison of `AllReduce` that performs an array $\overrightarrow{PA}$ of 8 32-bit elements in each of 8 workers. We have three observations.

First, P4SGD is able to reach average latency of $1.2\mu s$, which is an order of magnitude smaller than that of "CPUSync" and "GPUSync", because in-switch aggregation reduces the additional network hops, and our hardware implementation reduces software launching and synchronization overhead.

Second, the latency fluctuation of P4SGD is significantly smaller than that of "CPUSync" and "GPUSync", demonstrating one of the advantages of P4SGD in terms of offering deterministic latency. Third, SwitchML [16] introduces longer latency even than "CPUSync" and "GPUSync", because 1) SwitchML leverages the shadow copy mechanism to delay the acknowledgment of received aggregation packets for higher throughput, and 2) SwitchML uses data packets with a minimum size of 256B, while other methods adopt 64B network packets.
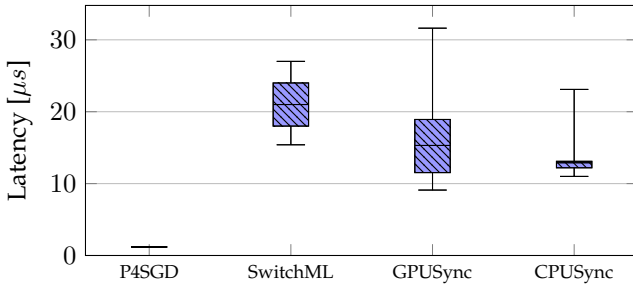

Fig. 8: Aggregation latency comparison. Whiskers show the 1st and 99th percentile.

5.3 Comparison of Data Parallelism and Model Parallelism

In this section, we compare the time per epoch of data-parallel and model-parallel approaches with other two baselines: "CPUSync" and "GPUSync". The number of workers in all experiments is 4. The number of engines of P4SGD is 8. Figure 9 illustrates the comparison results under different mini-batch sizes on the dataset $rcv1$ and $amazon\_fasion$. We have three observations. First, model parallelism has significantly smaller elapsed time per epoch than its corresponding data parallelism in most cases on

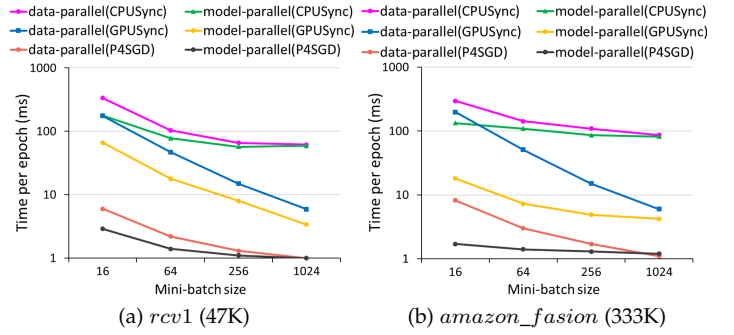
(a) $rcv1$ (47K) (b) $amazon\_fasion$ (333K)

Fig. 9: Hardware efficiency comparison between data-parallel and model-parallel, the number of workers is 4.

the same platform, because model parallelism introduces less network traffic especially when the mini-batch size is small. Although data parallelism requires larger epoch time when the mini-batch size is large, significantly more epochs are required to reach the same convergence rate, resulting in a smaller overall convergence speedup. Second, model parallelism has a higher speedup over data parallelism on the same platform when B is smaller, because P4SGD introduces hardware pipeline parallelism and P4-switch-FPGA in-switch aggregation with ultra-low latency. For example, when B is 16, model parallelism is $4.8\times$ faster than data parallelism on FPGAs under $amazon\_fasion$, while model and data parallelism has roughly the same elapsed time when B is 1024. Third, model parallelism has a higher speedup over data parallelism on the same platform when the number of features is larger. For example, when B is 16, P4SGD is $2\times$ and $4.8\times$ faster than the corresponding data-parallel implementation on FPGAs under datasets $rcv1$ and $amazon\_fasion$, respectively. In the following experiments, we always use model parallelism.

5.4 Hardware Efficiency: Throughput

We examine the hardware efficiency of P4SGD, in terms of achievable throughput. First, we examine the effect of different characteristics on P4SGD. Second, we compare P4SGD with the baselines on GPUs and CPUs. In the following experiment, by default, we instantiate 8 engines within a worker for best performance.

5.4.1 Hardware Characteristics of P4SGD

We typically run 200 epochs and get the average throughput to analyze the effect of each hardware characteristic.

Effect of Mini-Batch Size. We examine the effect of mini-batch size on P4SGD. We use the implementation of P4SGD-8-8. Figure 10 illustrates the P4SGD-8-8's speedup of various mini-batch sizes over the case with "B=16", in terms of throughput, on the different datasets. We have two observations. First, a larger mini-batch size leads to a higher speedup, since a large mini-batch size allows to overlap forward/backward propagation and communication between micro-batches that belongs to the same mini-batch. Second, a larger number of features leads to a smaller speedup when increasing the mini-batch size, because P4SGD is able to overlap communication time with computation time that occupies a higher proportion of the total computation time due to a larger number of features, even when B is small.
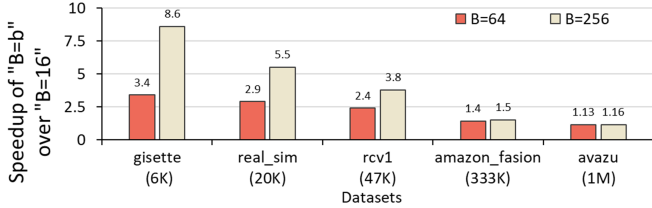
Fig. 10: Effect of mini-batch size, in term of epoch time, on different datasets under the case with 8 workers, each with 8 engines, B: mini-batch size
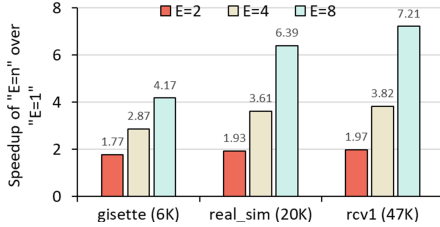

Fig. 11: Scale-up ability, in term of epoch time, under the case with 1 worker, E: number of engines, mini-batch size = 64, and precision = 4 bits

Scale-up Ability. We examine the scale-up ability of P4SGD that instantiates multiple engines for model-parallel training on GLMs. Figure 11 shows the throughput ratio of the cases with a various number of engines over the case with one engine on the datasets $gisette$, $real\_sim$, and $rcv1$. We have two observations. First, a larger number of engines leads to higher throughput due to fewer training tasks per engine within a worker. Second, a larger number of features leads to a higher throughput improvement when increasing the number of engines, because a larger number of features leads to a higher proportion of computation time to overall training time, where computation time can be linearly reduced by introducing more engines.

Scale-out Ability. We examine the scale-out ability of P4SGD that employs multiple workers for model-parallel training on GLMs. Figure 12 shows the throughput ratio of the cases with a various number of workers over the case with one worker. We have three observations. First, a larger number of workers leads to a higher throughput due to fewer training tasks per worker. Second, a larger number of features leads to a higher throughput improvement when increasing the number of workers, because a larger number of features leads to a higher proportion of computation time to overall training time, which can more easily amortize the negative effect of communication time. Third, when the number of features reaches 1 million, the throughput closely increases linearly with an increasing number of machines. It indicates that P4SGD achieves a strong scale-out ability when the number of features is large enough ($> 1M$).

5.4.2 Scalability Comparison with CPU/GPU Baselines

We compare scalability, in terms of time per epoch, of P4SGD with other three baselines: "SwitchML", "CPUSync", and "GPUSync". Figure 13 illustrates the average epoch time with the various number of workers and mini-batch size. We have four observations.

First, "P4SGD" is significantly faster than the other three counterparts and has the highest scalability. This is due to
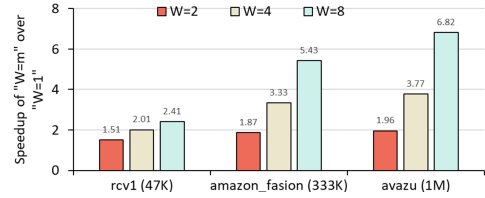

Fig. 12: Scale-out ability of P4SGD in term of epoch time with 8 engines, W: number of workers, mini-batch size = 16, and precision = 4 bits

its latency-centric in-switch aggregation protocol and hardware pipeline parallelism to maximize the overlap between communication and forward/backward propagation.

Second, "GPUSync" fails to scale out when B is relatively small, because 1) it cannot fully utilize the GPU computing power due to severe kernel invocation overhead (innovating three kernels per iteration), and 2) more distributed GPUs exacerbate kernel invocation overhead and thus amortize the benefit of reduced computation time of the CUDA kernel $cublasSgemm$, especially when the dimension is relatively small, as shown in Figure 13a. High-performance intra-node NVLink slightly relieves the communication overhead, as the communication stage only accounts for roughly 20% of the total training time when using RDMA-GPUDirect-powered NCCL.

Third, "CPUSync" can relatively easily scale out, because computation time dominates the overall training time on distributed CPUs, and communication time is negligible. Therefore, when the number of workers increases, the overall training time can drop quickly.

Forth, "SwitchML" is slower than "CPUSync", and its scale out ability is also worse than that of "CPUSync". The mainly reason for this is that "SwitchML" has the highest aggregation latency due to its shadow copy mechanism that delays the acknowledgement of received aggregation packets, as shown in Figure 8. Actually, "SwitchML" adopts the shadow copy mechanism to greatly increase the throughput of in-network aggregation, rather than decreasing latency.

5.5 Statistical Efficiency: Loss vs. Epochs

We compare the statistical efficiency of P4SGD with "CPUSync", and "GPUSync" on the datasets $rcv1$ and $avazu$. Figure 14 shows the convergence trend under an increasing number of epochs. The batch sizes for all approaches are 64. We observe that all the methods require the same number of epochs to converge to roughly the same loss, because they are all synchronous.

5.6 End-to-End Comparison: Loss vs. Time

We compare the end-to-end performance of P4SGD with "CPUSync" and "GPUSync", which use the configurations with the shortest convergence time. Figure 15 shows the end-to-end convergence comparison, in terms of loss vs. time, on the datasets $rcv1$ and $avazu$. We have two observations. First, P4SGD is able to converge up to 6.5X faster than "GPUSync", indicating great potential on in-network aggregation and micro-batch model-parallel training that enable efficient model-parallel training on distributed FPGAs. "GPUSync" converges relatively slowly, due to 1) long communication overhead per iteration, and 2) no overlap
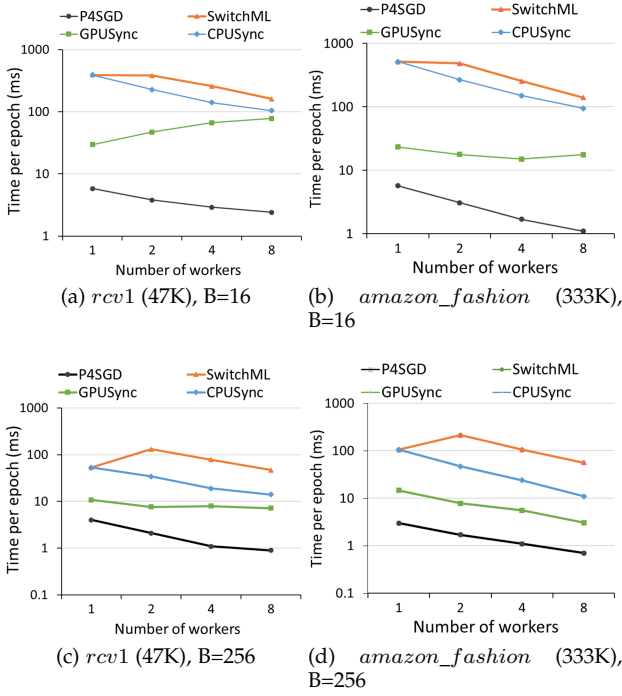
(a) $rcv1$ (47K), B=16

(b) $amazon\_fashion$ (333K), B=16

(c) $rcv1$ (47K), B=256

(d) $amazon\_fashion$ (333K), B=256

Fig. 13: Hardware efficiency comparison between P4SGD and other baselines

(a) $rcv1$ (47K)

(b) $avazu$ (1M)

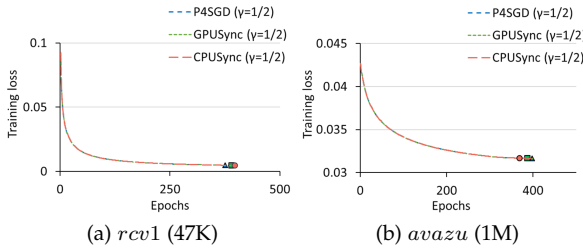Fig. 14: Statistical efficiency: training loss vs. epoch, γ: learning rate, precision = 4 bits

between forward/backward propagation and communication. Second, P4SGD can converge up to 67X faster than "CPUSync", because "CPUSync" suffers from low compute power and long collective operation latency.

5.7 Energy Consumption

In addition to hardware efficiency and performance, energy consumption has become a significant consideration in recent years. P4SGD has a significant advantage in terms of energy consumption.

To measure the power consumption of the Xilinx Alveo U280, we use the Alveo Card Management Solution Subsystem (CMS Subsystem). The CMS firmware is responsible for gathering the U280's voltage, current, power consumption, and other related information from the satellite controller device. [32] The power consumption of the "CPUSync" is measured using lm_sensors [33], an open-source application that provides tools and drivers for monitoring temperatures, voltage, and power. The power consumption of the "GPUSync" is measured using the NVIDIA System Management Interface (nvidia-smi) [34], which can manage and monitor NVIDIA GPU devices.

We evaluate the power consumption of P4SGD against "CPUSync" and "GPUSync" in the end-to-end experiments
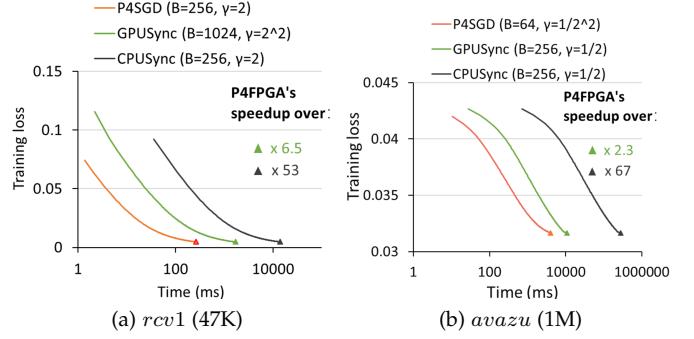

(a) $rcv1$ (47K)

(b) $avazu$ (1M)

Fig. 15: End-to-end convergence comparison: training loss vs. time, B: mini-batch size, γ: learning rate, precision = 4 bits

TABLE 4: Energe consumption, the number of workers for all methods is 8.

Method	Dataset	Time(s)	Total Power(W)	Energy(J)
P4SGD	rcv1	0.27	528	143
	avazu	4.12	528	2175
GPUSync	rcv1	1.76	920	1619
	avazu	10.9	920	10028
CPUSync	rcv1	14.4	496	7142
	avazu	128.25	496	63612

in Section 5.6. Table 4 presents the energy consumption comparison, which does not include the power consumption of the host system. The results show that P4SGD is up to 11X more energy-efficient than "GPUSync" and 50X more energy-efficient than "CPUSync", demonstrating the significant advantage of P4SGD in terms of energy consumption.

6 RELATED WORK

Parallelism in Distributed ML. ColumnSGD [35] proposes the distributed SGD training system using model parallelism on multiple CPUs. Its iteration time remains almost unchanged as the number of machines increases. Even worse, ColumnSGD increases the per-iteration time by 1.3 times, when the number of machines increases from 10 to 40. MegatronLM [12] efficiently trains multi-billion language models with model parallelism between GPUs within a node and with data parallelism between nodes. Alpa [36] automates inter- and intra-operator parallelism for training a *large* DL model that cannot fit in a GPU. Alpa recommends intra-operator parallelism, due to its high communication overhead, to accelerators that are connected with high-bandwidth communication link like NVLink, and recommends inter-operator parallelism to distributed accelerators that are connected by relatively low-bandwidth network. GSPMD [37] auto-completes the sharding on every tensor based on user annotations, the user can combine data-, model-, and pipeline- parallelism, etc. However, GSPMD does not partition individual operators or tensors in pipeline parallelism, but partitions the training graph into multiple stages that run on different devices. However, P4SGD strives to efficiently, e.g., in strong scaling, leverage intra-operator parallelism (model parallelism in our paper) to train a fully-connected layer (i.e., GLM model) between distributed accelerators such as FPGA with the help of a programmable switch.

Pipeline Parallelism in Distributed ML. Previous pipeline parallelism, e.g., GPipe [38], HetPipe [39], TeraPipe [14]

assigns different layers of a model to different GPUs to train a large model, and thus is orthogonal to P4SGD, which targets model parallelism.

FPGA-accelerated ML systems. P4SGD is closest to ML-Weaving [24], which is a one-size-fits-all system for any-precision learning. The hardware efficiency of MLWeaving is comparable to that of P4SGD using 1 engine and 1 worker. Prior works [40]–[53] exploit FPGAs to achieve high performance, but not for model-parallel training. Brainwave [45] accelerates DNN inference via model parallelism on multiple FPGAs. Previous work [53] presents a multiple-FPGA system for accelerating distributed SVM training, but all the FPGAs are on one host, where they communicate with each other through PCIe. Therefore, its scalability is limited in a single host.

In-network Aggregation. Current research ATP [17], SwitchML [16] and [54] leverages programmable network switch to perform gradient aggregation in data parallel training. Also, they adopt shadow copy mechanism to optimize for throughput. In contrast, P4SGD optimizes for ultra low in-network aggregation latency to benefit our model-parallel training between distributed FPGAs. SHARP [55] offloads collective operation processing to the InfiniBand network. SHARP approaches a hierarchical aggregation tree architecture to enable general aggregation operations on hundreds of hosts. Herring [56] is a scalable distributed data-parallel training library-based parameter server that adopts a balanced fusion buffer to solve the problem of unbalanced data sent and received by servers. Herring targets data-parallel distributed training mainly via increasing the throughput of aggregation operations, while P4SGD targets model-parallel distributed training mainly via reducing the latency of aggregation operations.

7 Conclusion

We propose P4SGD, a model-parallel distributed training system that allows strong scaling when training GLMs. P4SGD adopts micro-batch hardware pipeline-parallel training to overlap forward/backward propagation and communication within a worker. At the same time, we propose a latency-centric in-switch aggregation protocol to lower communication overhead between distributed FPGAs. We prototype P4SGD on multiple FPGAs and a P4 switch. The experimental result shows that P4SGD is able to converge up to 9.3x faster than its GPU counterpart. We will make P4SGD open-sourced to benefit the community.

Limitation and Future Work. The main limitation is that P4SGD implements the model on FPGA's on-chip memory, so the model size is limited due to the limited on-chip memory size. Our current implementation supports parameterizable model size, but up to 2M. However, we can easily generalize P4SGD to support a large model size by storing the model in external memory, e.g., HBM. As such, the implementation for a large model needs more memory bandwidth to achieve line-rate hardware processing. Fortunately, the current P4SGD only uses 25% HBM's memory bandwidth, leaving the majority of memory bandwidth for engines to access a large model in external memory. We believe P4SGD would not lose any processing speed. We leave the related implementation in future work.

Another future work is to generalize P4SGD to DNN training using an FPGA-GPU co-processing approach. In particular, we train compute-intensive layers, e.g., convolutional, on GPUs via data parallelism, since these layers have a high computation amount per weight, while we train the fully-connected layers on P4SGD via model parallelism since these layers may have a big model size but have a low computation amount per weight.

The third future work is to run the P4SGD at a frequency of 225MHz, enabling it to process 512-bit data from a single HBM channel, thereby doubling the number of engines supported by the current design.

References

[1] S. Rendle, "Factorization Machines," in *ICDM*, 2010.

[2] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed Linear Algebra for Large-Scale Machine Learning," *PVLDB*, 2016.

[3] H. Liu, K. Wang, Z. Chen, and K. E. Jordan, "Efficient Multi-Stage Preconditioners for Highly Heterogeneous Reservoir Simulations on Parallel Distributed Systems," in *SPE RSC*, 2015.

[4] T. Zheng, G. Chen, X. Wang, C. Chen, and S. Luo, "Real-Time Intelligent Big Data Processing: Technology, Platform, and Applications," *Science China Information Sciences*, 2019.

[5] P. Pop, M. L. Raagaard, S. S. Craciunas, and W. Steiner, "Design Optimisation of Cyber-Physical Distributed Systems using IEEE Time-Sensitive Networks," *IET Cyber-Physical Systems: Theory & Applications*, 2016.

[6] L. Oden, B. Klenk, and H. Fröning, "Energy-Efficient Collective Reduce and Allreduce Operations on Distributed GPUs," in *CC-GRID*, 2014.

[7] J. Lee, I. Hwang, S. Shah, and M. Cho, "FlexReduce: Fexible Allreduce for Distributed Deep Learning on Asymmetric Network Topology," in *DAC*, 2020.

[8] 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 *OSDI*, 2014.

[9] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *NIPS*, 2013.

[10] 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," in *EuroSys*, 2016.

[11] A. Krizhevsky, "One Weird Trick for Parallelizing Convolutional Neural Networks," *CoRR*, 2014.

[12] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2020.

[13] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," *arXiv preprint arXiv:1807.05358*, 2018.

[14] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models," *CoRR*, 2021.

[15] Microsoft, "ZeRO & DeepSpeed: New system optimizations enable training models with over 100 billion parameters," https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/.

[16] 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," *arXiv preprint arXiv:1903.06701*, 2019.

[17] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. M. Swift, "ATP: In-network Aggregation for Multi-tenant Learning." in *NSDI*, 2021.

[18] Xilinx, "Alveo U280 Data Center Accelerator Card Data Sheet," https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf, 2020.

[19] Edge-core, "Wedge 100BF-32X R07," https://www.edge-core.com/_upload/images/Wedge100BF-32X_65X_DS_R07_20200731.pdf, 2020.

[20] Intel, "Intel Tofino Programmable Ethernet Switch ASIC," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[21] Intel, "Intel Tofino 2 P4-programmable Ethernet switch ASIC," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[22] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading Latency for Compute in the Network," in *NAI*, 2020.

[23] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed Shared Memory with In-Network Cache Coherence," in *FAST*, 2021.

[24] Z. Wang, K. Kara, H. Zhang, G. Alonso, O. Mutlu, and C. Zhang, "Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-Precision Learning," *Proc. VLDB Endow.*, 2019.

[25] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 383–396.

[26] Y. Umuroglu, L. Rasnayake, and M. Själander, "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 307–3077.

[27] C.-C. Chang and C.-J. Lin, "LIBSVM: a Library for Support Vector Machines," *TIST*, 2011.

[28] NVIDIA, "cublas," 2010.

[29] Alan Gray, "Getting Started with CUDA Graphs," https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[30] NVIDIA, "Accelerating GPU-Storage Communication with NVIDIA Magum IO GPUDirect Storage," https://nvdam.widen.net/s/k8vrp9xkft/tech-overview-magnum-io-1790750-r5-web, 2021.

[31] NVIDIA , "NVIDIA Collective Communication Library (NCCL) Documentation," https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html, 2020.

[32] Xillinx, "Alveo Card Management Solution Subsystem Product Guide (PG348)," https://docs.xilinx.com/r/en-US/pg348-cms-subsystem/Introduction.

[33] Free Software Foundation, Inc., "lm-sensors," https://github.com/lm-sensors/lm-sensors.

[34] NVIDIA, "NVIDIA System Management Interface," https://developer.nvidia.com/nvidia-system-management-interface.

[35] Z. Zhang, W. Wu, J. Jiang, L. Yu, B. Cui, and C. Zhang, "ColumnSGD: A Column-oriented Framework for Distributed Stochastic Gradient Descent," in *ICDE*, 2020.

[36] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez *et al.*, "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning," *arXiv preprint arXiv:2201.12023*, 2022.

[37] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni *et al.*, "GSPMD: General and Scalable Parallelization for ML Computation Graphs," *arXiv preprint arXiv:2105.04663*, 2021.

[38] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *NeurIPS*, 2019.

[39] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "Hetpipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in *ATC*, 2020.

[40] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs," in *FPL*, 2018.

[41] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A Massively Parallel FPGA-based Coprocessor for Support Vector Machines," in *FCCM*, 2009.

[42] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," in *MICRO*, 2016.

[43] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim *et al.*, "A Cloud-Scale Acceleration Architecture," in *MICRO*, 2016.

[44] G. R. Chiu, A. C. Ling, D. Capalija, A. Bitar, and M. S. Abdelfattah, "Flexibility: FPGAs and CAD in Deep Learning Acceleration," in *ISPD*, 2018.

[45] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *MICRO*, 2018.

[46] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *ISCA*, 2017.

[47] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ISCA*, 2018.

[48] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "FPGA-Accelerated Dense Linear Machine Learning: A Precision-Convergence Trade-off," in *FCCM*, 2017.

[49] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian *et al.*, "E-RNN: Design Optimization for Efficient Recurrent Neural Networks in FPGAs," in *HPCA*, 2019.

[50] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh, "In-RDBMS Hardware Acceleration of Advanced Analytics," *arXiv preprint arXiv:1801.06027*, 2018.

[51] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016.

[52] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration," in *HPCA*, 2015.

[53] J. Dass, Y. Narawane, R. N. Mahapatra, and V. Sarin, "Distributed Training of Support Vector Machine on a Multiple-FPGA System," *TC*, 2020.

[54] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training," in *SoCC*, 2018.

[55] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir *et al.*, "Scalable Hierarchical Aggregation Protocol (SHArP): a Hardware Architecture for Efficient Data Reduction," in *COMHPC*, 2016.

[56] I. Thangakrishnan, D. Cavdar, C. Karakus, P. Ghai, Y. Selivonchyk, and C. Pruce, "Herring: Rethinking the Parameter Server at Scale for the Cloud," in *SC*, 2020.

Hongjing Huang is currently a Eng.D. student at Zhejiang University, China. Prior to that, he received his master's and bachelor's degree from Zhejiang University. His research interests include distributed machine learning, SmartNIC, etc.

Yingtao Li is currently a Eng.D. student at Zhejiang University, China. His research interests include in-network computation, programmable switch application, etc.
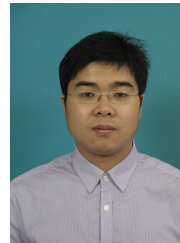
Jialin Li received his Ph.D. degree from the University of Washington in 2019. Li is currently an Assistant Professor in the School of Computing at the National University of Singapore. His research interests are in co-designing distributed systems with data center networks, data plane operating systems, and system software for programmable network hardware.

Jie Sun is currently a Ph.D. student at Zhejiang University, China. Prior to that, he received his bachelor's degree from Zhejiang University. His research interests include graph neural network, machine learning system, etc.

Xueying Zhu is currently a Ph.D. student at Zhejiang University, China. Prior to that, she received her bachelor's degree from Zhejiang University. Her research interests include in-network computation, SmartNIC, etc.

Zeke Wang received his Ph.D. degree from Zhejiang University, China in 2011. He is a Research Professor at Collaborative Innovation Center of Artificial Intelligence, Department of Computer Science, Zhejiang University, China. His current research interests mainly focus on building machine learning systems using heterogeneous devices, e.g., SmartNIC and SmartSwitch.

Jie Zhang is currently a Ph.D. student at Zhejiang University, China. Prior to that, he received his bachelor's degree from Zhejiang University. His research interests include cloud storage, SmartNIC, etc.

Liang Luo Received his Ph.D. degree from University of Washington in 2020. His research focuses on improving distributed training efficiency.