Accelerating Distributed Reinforcement Learning with In-Switch Computing

Youjie Li
UIUC
li238@illinois.edu

Iou-Jen Liu
UIUC
iliu3@illinois.edu

Yifan Yuan
UIUC
yifany3@illinois.edu

Deming Chen
UIUC
dchen@illinois.edu

Alexander Schwing
UIUC
aschwing@illinois.edu

Jian Huang
UIUC
jianh@illinois.edu

ABSTRACT

Reinforcement learning (RL) has attracted much attention recently, as new and emerging AI-based applications are demanding the capabilities to intelligently react to environment changes. Unlike distributed deep neural network (DNN) training, the distributed RL training has its unique workload characteristics – it generates orders of magnitude more iterations with much smaller sized but more frequent gradient aggregations. More specifically, our study with typical RL algorithms shows that their distributed training is latency critical and that the network communication for gradient aggregation occupies up to 83.2% of the execution time of each training iteration.

In this paper, we present iSwitch, an in-switch acceleration solution that moves the gradient aggregation from server nodes into the network switches, thus we can reduce the number of network hops for gradient aggregation. This not only reduces the end-to-end network latency for synchronous training, but also improves the convergence with faster weight updates for asynchronous training. Upon the in-switch accelerator, we further reduce the synchronization overhead by conducting on-the-fly gradient aggregation at the granularity of network packets rather than gradient vectors. Moreover, we rethink the distributed RL training algorithms and also propose a hierarchical aggregation mechanism to further increase the parallelism and scalability of the distributed RL training at rack scale.

We implement iSwitch using a real-world programmable switch NetFPGA board. We extend the control and data plane of the programmable switch to support iSwitch without affecting its regular network functions. Compared with state-of-the-art distributed training approaches, iSwitch offers a system-level speedup of up to 3.66× for synchronous distributed training and 3.71× for asynchronous distributed training, while achieving better scalability.

CCS CONCEPTS

• **Networks → In-network processing**; • **Computing methodologies → Distributed artificial intelligence**; **Reinforcement learning**; • **Hardware → Networking hardware**.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06...$15.00
https://doi.org/10.1145/3307650.3322259

KEYWORDS

distributed machine learning, reinforcement learning, in-switch accelerator, in-network computing

ACM Reference Format:
Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322259

1 INTRODUCTION

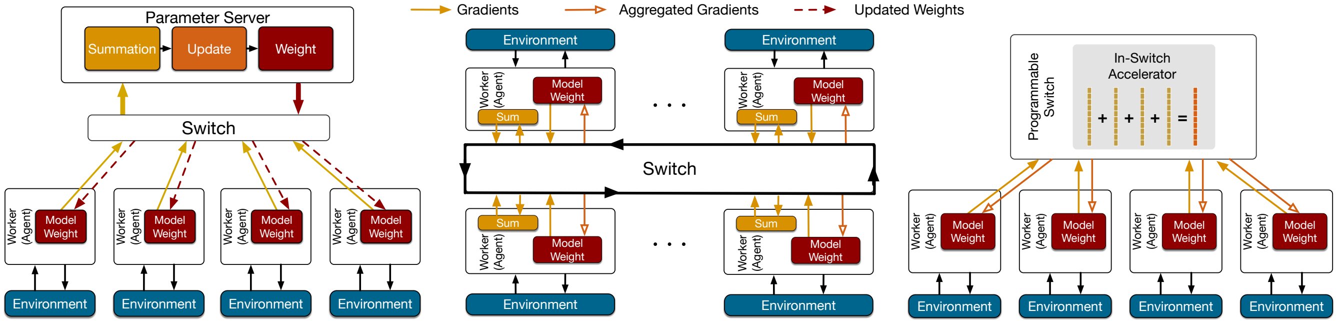
We have been seeing a disruptive trend that new and emerging AI applications are increasingly operating in dynamic environments and are taking actions to react to environment changes [11, 32, 37, 57]. These requirements of the emerging AI applications are naturally satisfied by reinforcement learning (RL). Similar to other popular machine learning techniques such as deep neural networks (DNN), RL also demands distributed training to improve their performance and training results based on the ever-growing need of analyzing larger amounts of data and training more sophisticated models.

Unlike distributed DNN training, the distributed RL training generates orders of magnitude more iterations with much smaller sized gradient aggregations. According to our study on popular RL algorithms (see Table 1), a typical RL algorithm [30] will generate millions of iterations, while its model size is much smaller than the size of a typical DNN model. Therefore, the latency of gradient communication in each iteration is a critical factor that significantly affects the performance of the distributed RL training.

To support distributed RL training, the state-of-the-art systems typically use one of two approaches. They either adopt the centralized parameter servers, in which the local gradient on each worker is aggregated to the central servers to perform weight update [7, 23, 46], or use the AllReduce based training, in which the gradient aggregation is conducted in a decentralized manner [10, 24].

In the former approach (see Figure 1a), it is well known that the centralized parameter server is the bottleneck that limits the scalability of distributed training [24, 26], as all training workers have to interact with the central server to transmit gradient or receive updated weight in each iteration. Considering that millions of iterations are involved in RL training, this bottleneck will significantly affect the training performance.

The latter approach (see Figure 1b) is proposed to address the scalability issue via performing gradient aggregation in a circular manner. However, it requires more network hops through switches



(a) Centralized parameter server (PS). (b) AllReduce-based training (AR). (c) In-switch acceleration (iSwitch).

Figure 1: Different approaches for (synchronous) distributed RL training.

Table 1: A study of popular RL algorithms.

RL Algorithm	DQN [31]	A2C [41]	PPO [48]	DDPG [27]
Environment	Atari [8]	Atari [8]	MuJoCo [52]	MuJoCo [52]
Model Size	6.41 MB	3.31 MB	40.02 KB	157.52 KB
Training Iteration	200.00M	2.00M	0.15M	2.50M

to complete aggregation on gradients of all the workers [10, 24, 51] in a cluster. As we scale the training with more computing nodes, the number of network hops required for gradient aggregations will be linearly increased.

To further understand the performance characteristics of these approaches, we quantify the overheads of the critical components in the distributed training with various RL algorithms such as DQN [30], A2C [41], PPO [48] and DDPG [27]. Our study results show that the network communication for gradient aggregation takes 49.9%–83.2% of the execution time of each iteration (see details in Figure 4).

To this end, we propose iSwitch, an in-switch acceleration approach for distributed RL training (see Figure 1c). iSwitch develops hardware accelerators for gradient aggregations within programmable switches. It is proposed as a practical and effective solution based on three observations. First, as discussed, the gradient aggregation is the major bottleneck in distributed RL training and it incurs significant network communication overhead. Moving the gradient aggregation from server nodes into network switches can significantly reduce the number of network hops required. Second, programmable switches have been widely deployed in data centers today. They provide the flexibility and basic computational capacity for developers to program the hardware, which simplifies the iSwitch implementation. Third, the switching techniques have been developed for decades with the purpose of scaling clusters. In-switch computing can scale the distributed RL training by leveraging the existing hierarchical rack-scale network architecture.

iSwitch is a generic approach that benefits both the synchronous and asynchronous distributed RL training. In synchronous training, all workers are blocked during gradient aggregation in each iteration. With in-switch accelerator, iSwitch reduces the end-to-end network communication overhead, and thus alleviates the blocking time. Moreover, since iSwitch conducts the in-switch aggregation at the granularity of network packets rather than the entire gradient vectors (consisting of numerous network packets), iSwitch further reduces the synchronization overhead caused by the aggregation.

For asynchronous distributed RL training, each worker runs independently without being blocked. However, due to the asynchrony, the removed blocking overhead is traded with staleness of local weight and gradient in training workers, which hurts the training convergence and increases the number of training iterations. iSwitch improves the convergence as the faster network communication enables workers to commit fresher gradients. Therefore, the training can converge in less number of iterations. To further increase the parallelism of the asynchronous distributed RL training, we rethink the RL training algorithms and fully pipelines the execution of local gradient computing, aggregation, and weight update.

Furthermore, iSwitch scales the distributed RL training at rack scale. It utilizes the existing rack-scale network hierarchy and integrates the in-switch accelerators into different layers of switches to conduct the hierarchical aggregation. iSwitch requires minimal hardware cost by extending the network protocols and control/data plane of programmable switches. As an extension to the programmable switch, iSwitch does not affect its regular network functions. To the best of our knowledge, this is the first work on the in-switch acceleration for distributed RL training. Overall, we make the following contributions in the paper:

- We quantify the performance characteristics of the distributed RL training with a variety of typical RL algorithms, and show that network communication for gradient aggregation is the major bottleneck in distributed RL training.
- We propose the in-switch computing paradigm, and develop the in-switch aggregation accelerator which significantly reduces the number of network hops required by gradient aggregation for each training iteration.
- We apply the in-switch acceleration to both synchronous and asynchronous distributed RL training. Upon the new distributed computing paradigm, we rethink the RL training algorithms to further improve their parallelism.
- We propose a hierarchical aggregation mechanism with exploiting the existing network architecture in rack-scale clusters to scale the distributed RL training.

We implement iSwitch with a real-world NetFPGA board. To demonstrate the efficacy of iSwitch, we train a variety of popular RL algorithms including DQN [30], A2C [41], PPO [48], and DDPG [27]. Our experimental results demonstrate that compared with state-of-the-art distributed training approaches, iSwitch offers a system-level

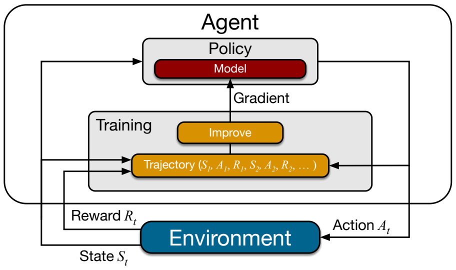


Figure 2: An example of reinforcement learning.

speedup of 1.72–3.66× for synchronous distributed training and 1.56–3.71× for asynchronous distributed training. Our evaluation also shows that iSwitch achieves better scalability for both synchronous and asynchronous distributed training in a rack-scale cluster.

2 BACKGROUND AND MOTIVATION

2.1 Distributed RL Training

The standard RL setting assumes an *agent* interacting with a given *environment* repeatedly over a large number of steps, as shown in Figure 2. At the beginning, the agent receives an initial *state* from the environment and then takes an *action* based on its *policy* (parameterized by a *model*) which maps current state to an action from a possible action set (i.e., action ← policy(state)). After the selected action takes effect in the environment, the next state will be generated and returned back to the agent along with a *reward*. This agent-environment interaction continues until the agent encounters a terminal state and the sequence of interactions between initial and terminal state forms an *episode*. Afterwards, the interaction restarts to generate a new episode. During the generation of numerous episodes, those states/actions/rewards are collected to form a *trajectory* which is then used to improve the policy by updating its model based on the computed *gradient*. The goal of the agent is to learn a policy that maximizes the reward objective – *episode reward*, i.e., the rewards accumulated over an episode.

It is well known that DNN training is time-consuming. This is also true for RL training. Different from DNN training, RL training requires a huge number of iterations, e.g., 200 million iterations to learn Atari games with DQN algorithm (see Table 1), as compared to the popular DNN, ResNet, which requires only 600K iterations [14], and thus demanding a significant amount of training time, e.g., eight days on a single GPU for DQN training [29].

To overcome this challenge, distributed RL training gains popularity recently [32, 33, 53]. Its key idea relies on multiple agents, namely *workers*, to explore the environments in parallel to earn local trajectories for model improvements, i.e., gradients. Those computed local gradients from workers can be "aggregated" (i.e., gradient aggregation) by a central node or decentralized workers to obtain a fully summed gradients for updating the model of the policy. Once the policy is improved, workers get ready for the next training iteration.

2.2 Synchronous and Asynchronous Training

The workers in distributed training can run either synchronously or asynchronously. In synchronous setting, all workers are blocked during gradient aggregation (as well as weight update and transfer) in each iteration. In asynchronous setting, all workers are allowed to run independently without blocking.

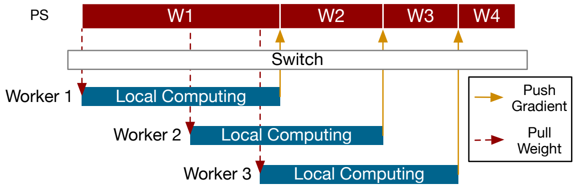


Figure 3: The asynchronous distributed RL training with the centralized parameter server.

We demonstrate an example of asynchronous training with the parameter server approach in Figure 3, where the server maintains the up-to-date weights and workers independently pull the latest weight for local computation. Once gradient is computed locally (although staled already), it is pushed to the parameter server to update current weight. Through the centralized server, all workers, although runs asynchronously, always keep up to the up-to-date weight in a certain extent. Note that the asynchronous training does not apply to the AllReduce approach (see Figure 1b), since the circular aggregation in AllReduce is a globally synchronized process [10, 25, 51].

As synchronous and asynchronous approaches offer different trade-offs, they co-exist as the two mainstream methods for distributed training. Synchronous distributed training demands synchronization among workers for gradient aggregation, and global barrier is placed for each training iteration [7, 10, 23, 56]. Such blocking aggregation stays in the critical path of the synchronous training systems and significantly affects the execution time of each iteration, especially in large-scale distributed systems [12, 15, 26].

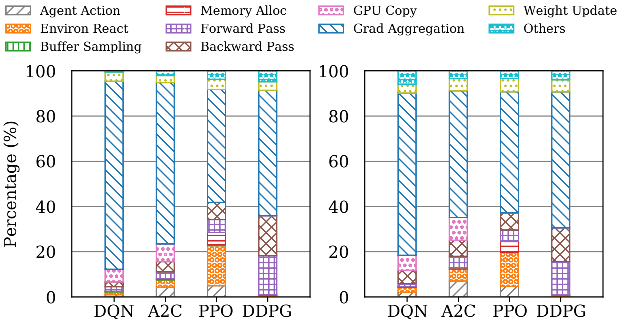
Asynchronous training [15, 21, 46] breaks the synchronous barrier among workers for minimal blocking overhead. However, the asynchrony suffers from the drawback of using stale gradient for model update, which slows down training convergence [15, 25], i.e., requiring more training iterations. By contrast, the synchronous training has no staleness issue, and thus enjoys a faster convergence, i.e., requiring minimal iterations.

Ideally, we want to have fast gradient aggregation for both synchronous and asynchronous training, such that synchronous training will pay less blocking overhead for aggregation, and asynchronous training will obtain fresher gradient for faster convergence. In this paper, we propose iSwitch that can benefit both synchronous and asynchronous RL training.

2.3 Gradient Aggregation Approaches

As discussed, there are two mainstream approaches for gradient aggregation in distributed RL training: centralized parameter server based approach (PS) [7, 23] and decentralized AllReduce based approach (AR) [10, 24].

We compare both approaches in Figure 1. We show the PS approach in Figure 1a, in which the local gradients in each worker are sent to the central server to perform summation, followed by the weight update. The updated weights are then sent back to all workers to overwrite their local copies, such that the next iteration can start. Figure 1b illustrates the Ring-AllReduce approach, in which each worker sends its local gradients to the next neighbor to perform



(a) PS. (b) AllReduce.

Figure 4: Performance breakdown of each iteration in distributed RL training using PS and AllReduce based approach, respectively. Gradient aggregation is the bottleneck.

partial summation in a circular manner until the gradients are fully aggregated. Afterwards, each work uses the aggregated gradients to perform update on local weights.

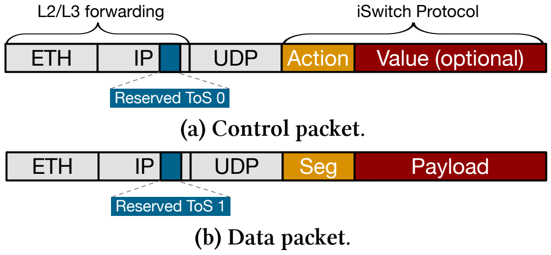
To facilitate our discussion, we assume that there are multiple workers and a central parameter server connected with a network switch. For the PS approach, each worker has to go through four network hops to complete the gradient aggregation. And the central server is the bottleneck. The AR approach avoids this central bottleneck but requires much more network hops. For the case where N workers connected to a switch, the number of network hops for the aggregation is $(4N-4)$ as discussed in [24, 51], which is linear to the number of workers.

To further understand their performance characteristics, we run the synchronous distributed RL training with both PS and AR approaches in a GPU cluster connected with 10Gb Ethernet (see the detailed experimental setup in § 5.3). We break down the training procedure of each iteration into multiple components: local gradient computing (including agent action, environment reaction, trajectory buffer sampling, memory allocation, forward pass, backward pass, and GPU memory copy), gradient aggregation, weight update, and others. We quantify their performance overheads in Figure 4. As can be seen, the gradient aggregation occupies a large portion (49.9% – 83.2%) of the execution time of each iteration for both PS and AR approaches. As the gradient aggregation involves only simple arithmetic operation (e.g., sum), its overhead mainly comes from the network communication.

2.4 Why In-Switch Computing

To this end, we propose iSwitch, an in-switch computing approach that exploits the computational capacity of programmable switches to reduce the gradient aggregation overhead. As shown in Figure 1c, iSwitch requires only two network hops (i.e., from worker node to switch, and from switch to worker node) to complete the gradient aggregation. iSwitch cuts the number of network hops by at least half, and thus offers much lower end-to-end communication time for each iteration of distributed RL training.

We utilize programmable switches to pursue the in-switch computing approach for accelerating distributed RL training for three reasons. First, programmable switches are pervasive today. In modern data centers or rack-scale clusters, programmable switches have become the backbone technology that allows developers to define their own functions for network packet processing. Second, they offer the flexibility for developers to program the hardware, which simplifies the iSwitch implementation. The programmable switch has control plane and data plane. The control plane is in charge of network management, while the data plane is responsible for data transferring (i.e., packet forwarding). We develop iSwitch by extending both the control plane and data plane without affecting the regular network functions. Third, the switch inherently enables the scalability. For instance, the switches have been widely used to scale the cluster size in data centers. In this project, we exploit the existing network architecture of a typical data center to scale distributed RL training in rack-scale clusters.



(a) Control packet.

(b) Data packet.

Figure 5: Format of the control/data packet in iSwitch.

3 DESIGN AND IMPLEMENTATION

The goal of iSwitch is to reduce the end-to-end execution time of distributed RL training by alleviating its network communication overhead and increasing its parallelism and scalability.

3.1 Challenges

As discussed, exploiting programmable switches to conduct gradient aggregation brings benefits for distributed RL training. However, we have to overcome three challenges.

- First, the programmable switch was originally designed for packet forwarding. Our in-switch computing needs to enable the point-to-point communication between the switches and worker nodes for gradient aggregation, without affecting the regular network functions (§ 3.2).
- Second, the programmable switch has limited computation logic and on-chip memory for our acceleration purpose. Therefore, our design should be simple and efficient to meet the performance requirement (§ 3.3).
- Third, as we increase the number of worker nodes and switches in a rack-scale cluster, our proposed in-switch computing should be able to scale for distributed RL training (§ 3.4).

In the following, we will address the aforementioned challenges respectively by extending the programmable switches.

3.2 Network Protocol Extension

To support in-switch computing for distributed RL training, we propose to build our own protocol and packet format based on regular network protocols. Figure 5 demonstrates the format of the control and data packets in iSwitch. We use Type of Service (ToS) field [13] in the IP header to identify packets with our specialized protocol.

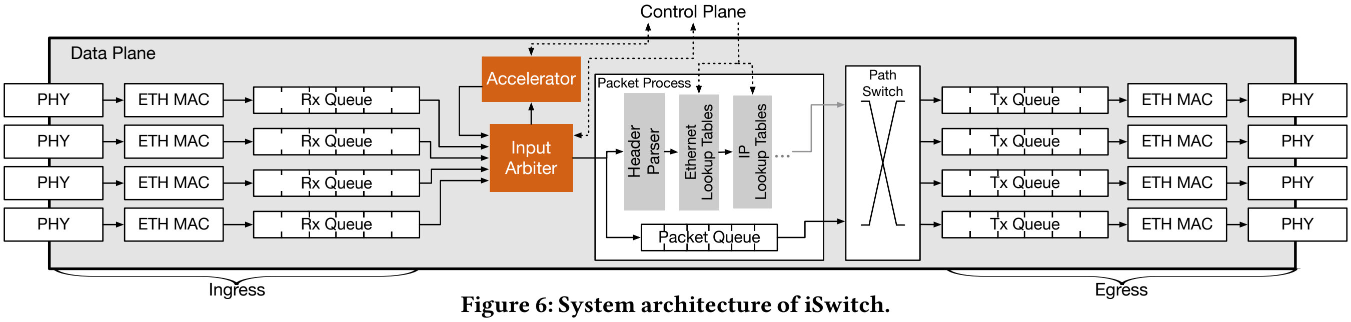


Figure 6: System architecture of iSwitch.

Table 2: Control messages in iSwitch protocol.

Name	Description
`Join`	Join the training job
`Leave`	Leave the training job
`Reset`	Clear accelerator buffers/counters on the switch
`SetH`	Set the aggregation threshold H on the switch
`FBcast`	Force broadcasting a partially aggregated segment on the switch
`Help`	Request a lost data packet for a worker
`Halt`	Suspend the training job on all workers
`Ack`	Confirm the success/failure of actions

ToS is a 1-byte field in the IP protocol header and is used to prioritize different IP flows. We tag packets that belong to the in-switch RL training with reserved ToS values. To differentiate between control and data packets in iSwitch, we use different ToS values.

Control message. As shown in Figure 5a, tagged by a reserved ToS value, the packet of control message has one 1-byte mandatory `Action` and one optional `Value` payload after the UDP header. In the `Action` field, we define multiple unique action codes for the basic operations for distributed RL training (see Table 2).

For some actions, we will use the `Value` field. To be specific, for `Join` message, the `Value` field can be used for the meta-data regarding the training model. Also, for `SetH` message, the `Value` field is used to specify how many gradient vectors (i.e., aggregation threshold H) need to be aggregated before broadcasting the results. By default, H is equal to the number of workers.

Data message. Figure 5b depicts the format of the data packet. Similar to the control packet, data packet is also tagged with a reserved ToS value. Its UDP payload begins with a 8-byte `Seg` field to indicate the indices of the transferred data packets. Each `Seg` number corresponds to a spacial offset in the gradient vector and the gradient data from the packets with the same `Seg` number will be aggregated. Besides the `Seg` field, the rest payload space (limited by the Ethernet frame size, i.e., typically 1,522 bytes) is filled with the gradient data. Furthermore, for the efficiency of data processing, all gradient data are transmitted and computed in a raw float-point format in iSwitch.

3.3 Data and Control Plane Extension

In iSwitch, we design and integrate an in-switch accelerator into the data plane as a "bump-in-the-wire" component.

Data plane extension. We present the extended data plane of an Ethernet switch in Figure 6. The incoming network packets are received by PHY Transceiver (PHY) and Ethernet Media Access Control (ETH MAC), and then stored in Rx Queues for further processing. The Input Arbiter always selects one non-empty Rx Queue to fetch a packet in a prioritized order, and feeds the chosen packet into the Packet Process module. After that, the header information of the packet is extracted, parsed, and compared with different forwarding rules in the Lookup Tables for destination identification. And then, the packets will be dispatched to their corresponding egress Tx Queues, where the packets will be finally transmitted through Ethernet MAC and PHY Transceiver.

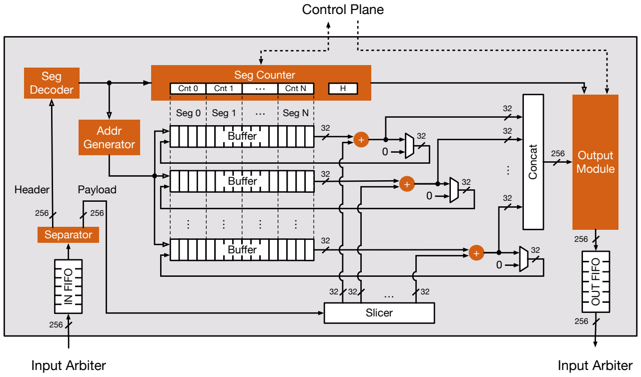
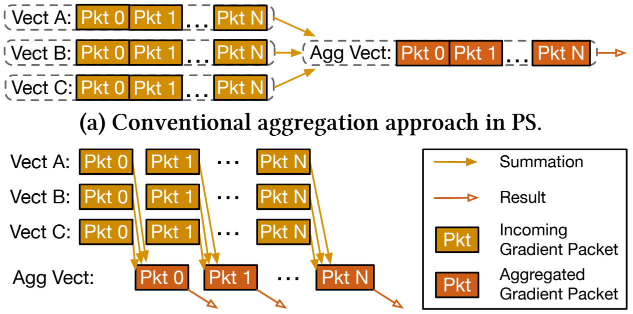


Figure 7: In-switch accelerator architecture.

To enable our in-switch acceleration, we enhance the functionality of the Input Arbiter, such that it can detect and feed tagged packets to the accelerator instead of the original Packet Process module, according to their ToS fields. And, the Input Arbiter treats the output of the in-switch accelerator as the output from an ingress Rx Queue, so that the result of gradient aggregation can be sent out to worker nodes as a regular traffic.

In-switch accelerator design. To maximize the data-level parallelism, our in-switch accelerator processes each packet at the granularity of "burst" which refers to the data that the internal bus can deliver in a single clock cycle (e.g., 256 bits). Thus, each data packet is divided into multiple bursts to be processed and computed.

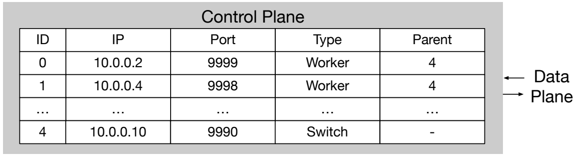
We describe the architecture of the in-switch accelerator in Figure 7. When a packet feeds in, a Separator will separate the bursts of the header from those of the payload. The header bursts, which include the Ethernet, IP, UDP, and iSwitch protocol fields, will be fed into a Seg Decoder. The payload bursts, which include a segment of the gradient vector, will be fed into the accumulation loops. The Seg Decoder will extract the Seg number, and pass it to both a Seg Counter and a Addr Generator. The Seg Counter keeps track of the aggregation


Vect A: Pkt 0 Pkt 1 ... Pkt N
Vect B: Pkt 0 Pkt 1 ... Pkt N
Vect C: Pkt 0 Pkt 1 ... Pkt N
Agg Vect: Pkt 0 Pkt 1 ... Pkt N
Vect A: Pkt 0 Pkt 1 ··· Pkt N
Vect B: Pkt 0 Pkt 1 ··· Pkt N
Vect C: Pkt 0 Pkt 1 ··· Pkt N
Agg Vect: Pkt 0 Pkt 1 ··· Pkt N
Summation
Result
Pkt Incoming Gradient Packet
Pkt Aggregated Gradient Packet

(a) Conventional aggregation approach in PS.

(b) On-the-fly aggregation approach in iSwitch.

Figure 8: Comparison of different aggregation approaches.



ID	IP	Port	Type	Parent
0	10.0.0.2	9999	Worker	4
1	10.0.0.4	9998	Worker	4
...	...	...	...	...
4	10.0.0.10	9990	Switch	-

Figure 9: Control plane and its membership table.

status of the gradient segments by assigning each segment an aggregation counter. The counter increments for each aggregated gradient segment until reaching the specified "aggregation threshold" H.

During the aggregation, each payload burst will be sliced by a Slicer into individual 32-bit floating-point elements, and fed into the adders. The adders compute in parallel, and keep summing the incoming payload bursts with accumulated segment data offered by the Buffers which store the immediate aggregated gradient data. To align the summation data for the same Seg number and burst offset, an Addr Generator is adopted to generate the Buffer addresses on the fly. This process continues until the full aggregation of a segment. And then, (1) the counter resets; (2) the Buffer is written back with zeros using the multiplexers; and (3) the Output Module is triggered for output by concatenating the results, prepending the header, and sending out a data packet containing the aggregated segment.

Beyond the fine-grained processing of each packet within the accelerator, iSwitch also conducts the gradient aggregation at the granularity of network packets. Different from conventional approaches (see Figure 8a) where they have to wait for the arrival of the entire gradient vectors before the summation operations, iSwitch starts the computation immediately as soon as the packets with the same Seg number are received. Such an on-the-fly aggregation approach hides the overhead of summation operations and data transmission, which further reduces the latency of gradient aggregation.

Control plane extension. To support distributed training with in-switch acceleration, we extend the control plane to maintain a lightweight membership table for the workers and switches involved in the current training job. As shown in Figure 9, the membership table records the ID number (an unique number for each membership entry), IP address, UDP port number, type, and the corresponding parent ID in the network typology for every involved worker/switch. The entries in membership table can be updated with the control messages, such as Join and Leave in Table 2. These information can be used by data plane for data collection, computation, forwarding, and broadcast. Besides maintaining a membership table, the control plane also manages the in-switch accelerator for its initialization, configuration, as well as resetting. This can be fulfilled through the control messages such as Reset and SetH in Table 2.

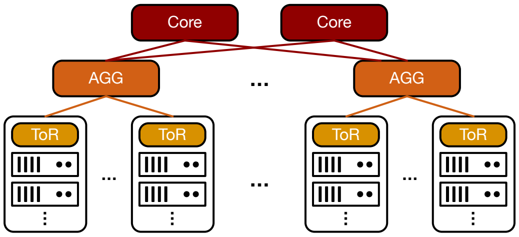


Figure 10: A typical network architecture at rack scale.

The control plane also helps handling packet lost, although it is uncommon in the cluster environment, with minimal overhead. Specifically, we offload the majority of tasks of handling lossy packet to workers, and leave only simple tasks such as accepting/forwarding control message (e.g., FBcast and Help) in the switch.

3.4 Scaling In-Switch Computing

We have discussed the in-switch acceleration for distributed RL training within a rack of worker nodes. We now discuss how to scale out the in-switch computing in a rack-scale cluster or data center. Figure 10 shows the network architecture of a typical cluster or data center [4, 47]. All the servers in the same rack are connected by a Top-of-Rack switch (ToR) with 10Gb Ethernet [1, 50]. In the higher level, there are Aggregate switches (AGG) and Core switches (Core) connected with higher network bandwidth (e.g., 40Gb to 100Gb).

To scale out distributed RL training with iSwitch in the rack-scale cluster, we develop an "hierarchical aggregation" approach. Specifically, if a switch finishes its local aggregation for a certain segment in the gradient vector stored in the Buffer, it will forward the aggregated segment to the switches in the higher level for global aggregation. If there are more than one switch in the higher level, the switch will select the one with the smallest value of IP addresses, so that all gradient data could be finally aggregated in the Core switch. Then the globally aggregated gradient will be broadcasted to the lower-level switches for further distribution. Such a design leverages the existing rack-scale network architecture and does not introduce additional hardware or network topology changes.

3.5 Implementation

We implement iSwitch with a real-world NetFPGA-SUME board [34]. NetFPGA-SUME has an ×8 Gen3 PCIe adapter card incorporating Xilinx Virtex-7 FPGA and four 10Gbp Ethernet ports. We use the reference switch design provided by NetFPGA community [35] for further development. To fully utilize the bit-width of its internal AXI4-Stream bus (i.e., 256 bits/cycle), we employ eight 32-bit floating-point adders for parallel gradient aggregation. Our in-switch accelerator is integrated into this reference switch design and interacts with other components using standard 256-bit AXI4-Stream bus at the frequency of 200MHz. In terms of the on-chip resource utilization, iSwitch accelerator consumes extra 18.6% of Lookup Table (LUT), 17.3% of Flip-Flop (FF), 44.5% of Block RAM (BRAM), and 17 DSP slices, compared with the unmodified reference design. Note that

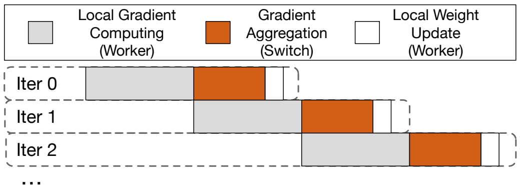

Local Gradient Computing (Worker)
Gradient Aggregation (Switch)
Local Weight Update (Worker)
Iter 0
Iter 1
Iter 2
...

Figure 11: The three-stage pipeline in the optimized asynchronous distributed RL training with iSwitch.

the implementation of iSwitch hardware and network protocols are generic to both synchronous and asynchronous distributed training.

4 IMPLICATIONS ON RL TRAINING

In the section, we discuss how we exploit the in-switch computing paradigm to facilitate our hardware/algorithm co-design, and further improve the performance of both synchronous and asynchronous distributed RL training.

As discussed in § 2.2, for synchronous training, we can directly apply iSwitch to reduce the end-to-end execution time of gradient aggregation by replacing the aggregation operation, such as the AllReduce operation, with our in-switch aggregation. We will not discuss the details (see § 6.1 for performance benefits). For asynchronous training, iSwitch offers new optimization space to improve the training parallelism with the in-switch computing paradigm, which demonstrates an useful case of iSwitch's implications on distributed RL training. We will discuss it in details in this section.

4.1 Rethink Asynchronous RL Training

Conventional approach for asynchronous distributed training (see Figure 3) relies on a central parameter server to maintain the up-to-date weights, where each worker interacts with the server to keep up with the latest weights such that the training can converge. To gain the benefits from iSwitch, a straightforward approach is to shift the functions of parameter server to the network switch. However, this will significantly increase the hardware cost, because the tasks running on parameter servers demand not only intensive computation resource, but also large memory space for storing weights and historical updates. With the in-switch aggregation, we rethink the asynchronous distributed training, and propose two optimization techniques to further decentralize the training and increase its parallelism.

Decentralized weight storage. Instead of pushing gradient to the central server, we aggregate gradients from asynchronous workers via switch and then broadcast the summed gradients to each worker for weight update in every iteration. Since we initialize the same model weights among all workers, and also broadcast the same aggregated gradients, the decentralized storage of weights are always agreed over iterations in spite of asynchronous training.

Three-stage pipeline. We decouple the three key stages within a training iteration: (1) Local Gradient Computing (LGC), (2) Gradient Aggregation (GA), and (3) Local Weight Update (LWU). The first stage is on the workers, which performs environment interactions, trajectory collection, and gradient generation with uploading. The second is in the switch, which conducts the gradient gathering, summing, and broadcasting. The third is on the workers for weight updates.

Algorithm 1 Asynchronous distributed training algorithm with in-switch acceleration (logical view)

In the switch

```
Initialize gradient buffer $g_{\text{sum}}$, the number of gradient vectors to aggregate $H$ (= the number of child nodes, by default).
while true do
    wait until $H$ gradient vectors received
    sum-reduce the $H$ gradient vectors into $g_{\text{sum}}$
    broadcast back summed gradients $g_{\text{sum}}$
end while
```

On "Local Weight Update (LWU)" thread of each worker

```
Initialize iteration index $t_s$, total number of iterations $T$, the same initial weight $w_s$ over workers, and learning rate $\gamma$.
for $t_s = 0 \ldots T$ do
    wait until $g_{\text{sum}}$ received
    update weight: $w_s \leftarrow w_s - \gamma \cdot g_{\text{sum}} / H$
end for
```

On "Local Gradient Computing (LGC)" thread of each worker

```
Initialize copy of iteration index $t_w$, copy of weight $w_w$, and staleness bound $S$. ($t_s$, $T$, $w_s$ are in shared/global memory.)
while $t_s < T$ do
    copy iteration index: $t_w \leftarrow t_s$
    copy updated weight: $w_w \leftarrow w_s$
    interact $environment$ and collect $trajectory$ based on $w_w$
    compute gradient $g_w$ based on collected $trajectory$ and $w_w$
    // check staleness of local gradient before commit
    if $(t_s - t_w \leq S)$ then
        nonblocking send $g_w$ to switch
    else
        continue
    end if
end while
```

For the three stages in a training iteration, we can pipeline them to increase the parallelism of distributed training, as illustrated in Figure 11. At the LGC stage, each worker runs independently without synchronizing with other workers or the switch, and keeps uploading computed gradients to the switch. At the GA stage, the switch aggregates gradients in an asynchronous manner, and keeps aggregating the incoming gradients. Once sufficient gradient vectors are received, the aggregated gradients are broadcasted back to workers, so that the LWU stage can start. Such an approach encourages faster workers to contribute more to the aggregation, while slower workers commit less without blocking the training.

Inevitably, due to the asynchrony, staleness of weights and gradients could occur, which would slow down the training convergence. In this work, we propose to bound the staleness of gradient explicitly. Specifically, we check the staleness of local gradient on each worker and commit only lightly staled gradients within a bound to the switch. We show the detailed procedure in Algorithm 1. We prove the convergence of our proposed asynchronous training with both empirical evaluations (see § 6.2) and theoretical derivations as below.

4.2 Proof of Convergence

To prove the convergence of asynchronous iSwitch, we convert it into the classical parameter-server based asynchronous training [15, 21]. By showing that the former is mathematically equivalent to the latter, we reach the same conclusion as in [15, 21] but constants change.

To be specific, we assume there is a virtual parameter server in our asynchronous iSwitch (see Algorithm 1), which stores the up-to-date weights and also performs weight updates as in the classical design. Such a parameter server is equivalent to the LWU thread on each worker node. As discussed, all workers perform identical weight updates over iterations, and thus the decentralized agreed weights can be regarded as being stored on a single centralized server. Consequently, gradient pushing, aggregation, and broadcasting can be reduced to the upstream communication to the parameter server, while weight copying in the LGC thread on each worker node can be reduced to the downstream communication from the parameter server. All workers run in parallel asynchronously to push gradients (through the switch) to the parameter server to perform updates, and then the updated weights will be used in a new iteration. The minor difference between our approach and [15, 21] lies in the aggregation of gradient vectors. This can be reduced to the usage of a larger batch-size for training, which does not change the convergence rate. Therefore, our proposed asynchronous training can be reduced to the the conventional approaches [15, 21], and offers a convergence rate of $O(T^{-0.5})$ for convex objectives via stochastic gradient descent, where T is the number of training iterations.

5 EVALUATION METHODOLOGY

5.1 Benchmarks for Distributed RL Training

To evaluate the training performance of iSwitch, we use four popular RL algorithms [27, 31, 41, 48] as our benchmarks. Based on their single-node training code [9, 16, 17], we develop three reference designs for each benchmark by following the state-of-the-art distributed training approaches: synchronous and asynchronous parameter-server based training (Sync/Async PS) [7, 15, 23, 46], and AllReduce based training (AR) [10, 24]. Our reference designs are highly optimized, and show around 10% better performance with higher training rewards than the OpenAI-Baseline [8] with MPI (a popular baseline used in the community [17, 32, 41]). We list these RL algorithms as follows:

- **DQN** [31] is one of the most popular RL algorithms for arcade game playing. Its model size is 6.4 MB [9] when applied to the task of playing Atari game set [3], from which we choose the classical game, “Pong”, as used in [9, 29, 31].
- **A2C** [41] is another popular RL algorithm for game playing. Its model size is 3.3 MB [17] when applied to the Atari game set [3], from which we choose a different yet classical game “Qbert”.
- **PPO** [48] is a more recent algorithm mainly for simulated robotic locomotion. Its model size is 40 KB [17] when applied to the robotic control in simulation environment set MuJoCo [52], from which we choose a classical environment, “Hopper”, as used in [17, 48].
- **DDPG** [27] is yet another algorithm for continuous control. The dual model size of DDPG is 157.5 KB in total [16] when applied to the task of robotic control in MuJoCo [52], from which we choose another classical environment, “HalfCheetah”.

We implement all reference designs using the state-of-the-art libraries: PyTorch 1.0 [45], CUDA 9.2 [38], CuDNN 7.2.1 [39], GYM [42], and OpenMPI 3.1.4 [43]. For iSwitch design, we use the same code and libraries from the reference design but with a different gradient aggregation method, i.e., in-switch aggregation, as well as a dual-thread training in asynchronous iSwitch (see Algorithm 1).

5.2 Metrics and Their Definitions

We use multiple training approaches for each benchmark: synchronous parameter server (PS), AllReduce (AR), iSwitch (iSW), as well as asynchronous parameter server (Async PS), iSwitch (Async iSW). We evaluate all approaches using the following metrics:

- **Final Average Reward**: the *episode reward* averaged over the last 10 *episodes* (see § 2.1), which is a standard metric used in the RL training evaluation.
- **Number of Iterations**: the number of training iterations required to complete the end-to-end training. For synchronous training approaches, it can be measured at any of worker nodes. For asynchronous training approaches, it can be measured precisely at the parameter server of PS or the LWU thread of iSW by counting the number of weight updates.
- **Per-Iteration Time**: the average time interval between two consecutive iterations. For synchronous approaches, it is the latency of one training iteration. For asynchronous approaches, it can be measured precisely by the time interval between two consecutive weight-update operations at the parameter server of PS or the LWU thread of iSW.
- **End-to-End Training Time**: the total training time required to achieve the same level of “Final Average Reward” for each benchmark with different approaches.

5.3 Experimental Setup

Main cluster setup. To measure the training performance in actual wall-clock time, we setup a main cluster consisting of four nodes. Each node has a NVIDIA Titan RTX GPU [40] and an Intel Xeon CPU E5-2687W@3*GHz* [19]. We use this four-node cluster for evaluating AR and iSW approaches. To also support the PS approach, we use an additional node as the parameter server. All nodes are connected to a Netgear 10Gb Ethernet switch [36] via Intel X540T2 10Gb Ethernet NICs [18]. Consider the small size of transferred gradients of RL models, e.g., 40KB for PPO, we do not consider supporting larger network connections (i.e., 40∼100Gbps) in our experiments. As for iSW approach, we replace the network switch with a NetFPGA-SUME board [34], and fully use the four Ethernet ports on the NetFPGA-SUME board to connect the worker nodes.

Scalability experiment setup. For the scalability experiments, we emulate the training performance of all the approaches with more worker nodes in a cluster consisting of two-layer regular switches as in Figure 10. Specifically, the cluster has a root switch connecting to multiple “racks” and each rack contains three worker nodes (due to the port limitation of NetFPGA boards). We emulate the hierarchical aggregation of iSwitch in the cluster. We develop the emulation with three goals: the emulated aggregation must have (1) the exact number of network hops, (2) the same amount of traffic in the network links as possible, and (3) accurate accelerator overhead. We achieve these goals by transferring synthetic gradient data from each worker node to its third next neighbor worker node, such that each gradient message always traverses through the hierarchy of switches. After that, a barrier is set among workers to capture the slowest gradient transfer such that the aggregation can be deemed as completed. This emulation approach matches the real aggregation

Table 3: Summary of performance speedups in "End-to-End Training Time" for different training approaches. Speedups are based on the baseline PS for each benchmark.

System-Level Speedup in End-to-End Training Time		DQN	A2C	PPO	DDPG
Sync	PS	1.00×	1.00×	1.00×	1.00×
	AR	1.97×	1.62×	0.91×	0.90×
	iSW	3.66×	2.55×	1.72×	1.83×
Async	PS	1.00×	1.00×	1.00×	1.00×
	iSW	3.71×	3.14×	1.92×	1.56×

for (1) and (2), although with minor amplification on the network traffic between switches. To achieve the goal (3), we measure the hardware accelerator overhead and add it to the aggregation time. For emulation of the local computation, we use the same trace from the PS/AR approaches, and apply it to the iSwitch for fair comparison. Besides, we also obtain the "Number of Iterations" required for iSwitch. For synchronous training, iSwitch shares the same number of iterations as PS/AR, due to their mathematical equivalence in distributed training (see Table 4). For asynchronous training, the iterations required by iSwitch can be emulated by controlling the usage of staled gradient in synchronous training approach, where the staleness of iSwitch can be calculated by the measured time ratio of the three stages (see Figure 11) in each training iteration. Thus, we believe the emulation platform can reflect the scalability of a real-world rack-scale cluster with in-switching computing enabled.

6 EVALUATION

We evaluate the training performance of the four benchmarks (see § 5.1) using the main cluster. We measure the "End-to-End Training Time", and summarize the performance speedups in Table 3. In synchronous training setting, iSwitch approach (iSW) prevails with a great margin compared to other approaches, and offers a performance speedup of 1.72–3.66×, compared with the baseline design (PS). Although AR approach also provides improvement on DQN and A2C, the performances on PPO and DDPG are actually slightly worse than the PS. As for the asynchronous training setting, the advantage of iSwitch still holds, and offers a performance speedup of 1.56–3.71×, compared to the baseline PS. Note that we evaluate the performance of synchronous (§ 6.1) and asynchronous (§ 6.2) distributed training approaches separately, as the main objective of this work is to accelerate and to support both types of approaches, instead of comparing them, as discussed in § 2.2.

6.1 Benefits with Synchronous iSwitch

To understand the performance improvement resulting from iSwitch under synchronous training setting, we compare the "Per-Iteration Time" of iSwitch with the PS and AR over four benchmarks in Figure 12. We also provide detailed timing breakdown of the "Per-Iteration Time" for different approaches. This result shows that compared with the PS, iSW offers 41.9%–72.7% shorter "Per-Iteration Time" because of the 81.6%–85.8% reduction in gradient aggregation time for the four benchmarks.

iSwitch provides substantial acceleration in gradient aggregation for three reasons. First, the aggregation process in iSwitch requires only half number of network hops (two hops) compared with the PS design (four hops), which achieves halved end-to-end communication latency. Second, iSwitch possesses the unique feature of aggregation on-the-fly (as shown in Figure 8), which reduces the aggregation granularity from the gradient vector size, i.e., the model size in baseline design, to the network packet size. Instead of waiting for the arrival of all gradient vectors before starting computation, iSwitch conducts aggregation immediately once packets of the same index arrive (see Figure 8), which reduces the synchronization overhead caused by gradient aggregation. Third, iSwitch offers balanced communication by assigning a dedicated network link to each of worker node, which removes the bottleneck caused by the central link in PS design.

In addition to the comparison with the baseline design (PS), we also compare iSwitch with another mainstream approach – AllReduce based training (AR) [10, 24] which offers balanced communication. The result in Figure 12 shows that iSwitch still outperforms AR over all four benchmarks, i.e., 36.7%–48.9% reduction in "Per-Iteration Time." These improvements are still attributed to the accelerated gradient aggregation of iSwitch, i.e., 63.4%–87.9% reduction in aggregation time for iSW, in comparison with AR. As discussed in § 2.3, there is a performance trade-off between PS and AR. The AR approach suffers from more network hops than PS, but it removes the bottleneck caused by the central parameter server. On the meanwhile, the

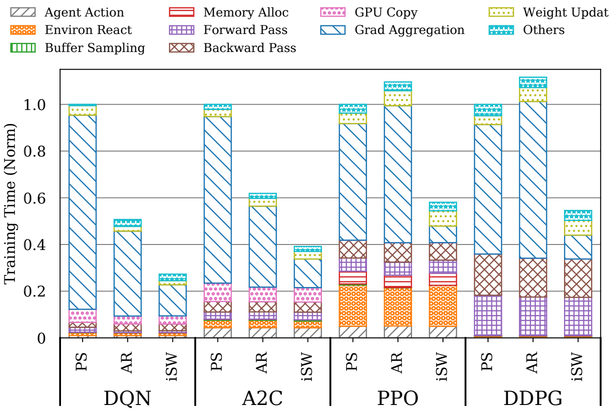


Figure 12: Comparison of "Per-Iteration Time" among different synchronous distributed training approaches along with detailed breakdown. All approaches are normalized against their baseline approach PS for each benchmark.

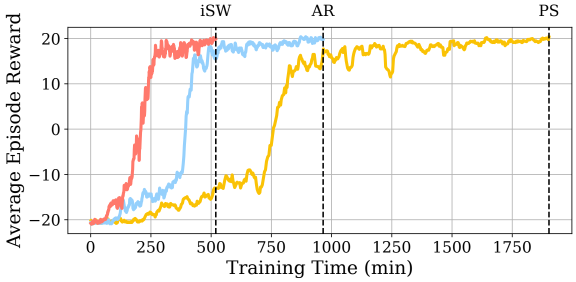


Figure 13: Comparison of training curves of DQN using different synchronous approaches. The curves are measured in training rewards vs. wall-clock time. iSW achieves the same level of rewards as other approaches in a much shorter time.

Table 4: Performance comparison of different synchronous distributed training approaches.

	DQN			A2C			PPO			DDPG		
	PS	AR	iSW	PS	AR	iSW	PS	AR	iSW	PS	AR	iSW
Number of Iterations	1.40E+06			2.00E+05			8.00E+04			7.50E+05		
End-to-End Training Time (hrs)	31.72	16.08	8.66	2.87	1.78	1.12	0.39	0.42	0.22	8.07	9.01	4.40
Final Average Reward	20.00	19.94	20.00	13491.73	13478.39	13489.22	3090.24	3093.18	3091.61	2476.75	2487.43	2479.62

Table 5: Performance comparison of different asynchronous distributed training approaches.

	DQN		A2C		PPO		DDPG	
	Async PS	Async iSW	Async PS	Async iSW	Async PS	Async iSW	Async PS	Async iSW
Number of Iterations	6.30E+06	3.50E+06	1.20E+06	4.00E+05	5.40E+05	1.20E+05	3.00E+06	1.50E+06
Per-Iteration Time (milli-secs)	24.88	12.07	13.13	12.53	3.40	7.99	11.58	14.89
End-to-End Training Time (hrs)	43.54	11.74	4.38	1.39	0.51	0.27	9.65	6.20
Final Average Reward	19.10	19.82	13402.83	13505.46	3083.67	3084.23	2421.89	2485.35

benchmarks demand different communication/computation loads due to their model sizes (see § 5.1). As a result, compared with PS, AR performs better for DQN and A2C but worse for PPO and DDPG. iSwitch runs faster than both PS and AR because of the reduced end-to-end network latency as well as the on-the-fly aggregation.

Furthermore, we show the detailed results including the number of iterations, absolute training time, and achieved training rewards, in Table 4. We observe that all synchronous approaches train the same "Number of Iterations" to reach the same level "Final Average Rewards" for each benchmark.

To demonstrate the synergy of acceleration and training rewards of all synchronous approaches, we evaluate the actual training curves in wall-clock time for all benchmarks, and demonstrate a case study of DQN in Figure 13.

6.2 Benefits with Asynchronous iSwitch

We now compare iSwitch with the asynchronous baseline (Async PS) for all benchmarks. To show a fair comparison, we gives the same staleness bound ($S = 3$) for both approaches, although the conventional Async PS approach does not involve staleness control mechanisms, such that the staleness of gradient ranges from 0 to 3 iterations.

We summarize the training performance of the two approaches in Table 5. We observe that iSwitch (Async iSW) offers faster convergence, i.e., 44.4%–77.8% reduction in the "Number of Iterations", compared with the baseline (Async PS). This is due to the smaller staleness of gradient on average [15, 25] in iSwitch, although both approaches are bounded by the same maximal staleness. The alleviated staleness of gradients can be attributed to the advantage of accelerated gradient aggregation in iSwitch, because the faster gradient aggregation results in earlier/in-time weight update, and thus offers fresher weight and gradient for next iteration. On the other hand, Async PS suffers from doubled end-to-end communication latency (as discussed in § 6.1), as well as the burdened central network link, and thus increases the gradient/weight communication time. As a result, the staleness of gradient becomes larger, causing an increased number of training iterations [15, 25, 26].

From Table 5, we also observe that iSwitch demonstrates 4.6%–51.5% shorter "Per-Iteration Time" for DQN and A2C, compared with the baseline. This is because asynchronous iSwitch not only enjoys the benefit of acceleration on gradient aggregation, but also employs the pipelined training to hide part of the execution time (see Figure 11), especially the accelerated gradient aggregation and weight update. By contrast, the Async PS still pays for the long communication latency, thus increasing the time interval between two consecutive weight updates, i.e., larger "Per-Iteration Time".

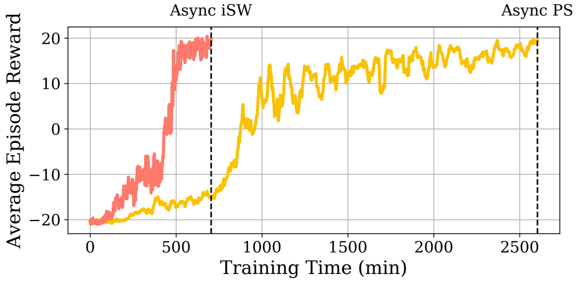


Figure 14: Comparison of training curves of DQN using different asynchronous approaches. The curves are measured in training rewards vs. wall-clock time. Our solution Async iSW achieves the same level of rewards as the other approach in a much shorter time.

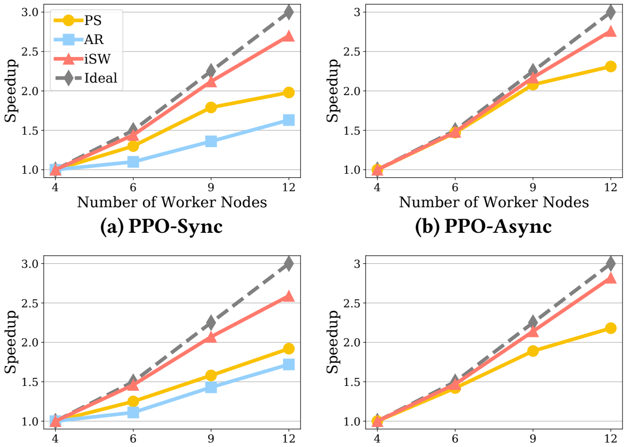
Note that for PPO and DDPG, iSwitch does not show improvement in "Per-Iteration Time". This is mainly due to the relatively smaller ratios of gradient aggregation time in PPO and DDPG. Therefore, even with pipeline, the hidden time of gradient aggregation only offers a slight reduction in "Per-Iteration Time", the limited benefit of which cannot outperform the Async PS. However, the accelerated gradient aggregation of iSwitch reduces the staleness of gradients, and reduces the number of training iterations.

To combine the effectiveness of iSwitch approach in both reduced "Number of Iterations" and improved "Per-iteration Time", we show the "End-to-End Training Time" in Table 5. Asynchronous iSwitch offers 35.7%–73.0% reduction in "End-to-End Training Time", compared with the baseline Async PS.

Moreover, to demonstrate the synergy of acceleration and training rewards of both asynchronous approaches, we evaluate the actual training curves in wall-clock time for all benchmarks, and demonstrate the an example of DQN in Figure 14.

6.3 Scalability of iSwitch

To evaluate the scalability, we measure and compare the speedups of the end-to-end training for all the training approaches, following the scalability experiment setup in § 5.3. We show the case study on the scalability of training PPO and DDPG with 4, 6, 9 and 12 worker nodes in Figure 15. For synchronous distributed training, as shown in



(a) PPO-Sync **(b) PPO-Async**

(c) DDPG-Sync **(d) DDPG-Async**

Figure 15: Scalability comparison of all training approaches. The speedups are based on the "End-to-End Training Time" and normalized against the 4-node case of each approach.

Figure 15a and 15c, we observe that the AR approach offers the worst speedups as the cluster scales. This is because its number of network hops for gradient aggregation is linear in cluster size, as discussed in § 2.3. The PS approach shows the second best scalability. However, it suffers from the central bottleneck in both communication and computation , and this drawback worsens as we increase the number of worker nodes. iSwitch outperforms both AR and PS with a great margin because of three major reasons: (1) the minimal number of network hops required, (2) balanced and reduced traffic load in hierarchical aggregation, and (3) the in-switch accelerator of iSwitch.

For asynchronous distributed training (see Figure 15b and 15d), we observe that asynchronous PS approach cannot outperform asynchronous iSwitch approach, since Async PS still requires more network hops, although the asynchronous mechanism alleviates the central bottleneck to some extent. By contrast, Async iSwitch holds the best scalability (i.e., almost linear speedups), since it enjoys not only the aforementioned advantages enabled by in-switch computing, but also the benefit of three-stage pipeline as well as the alleviated staleness from the accelerated aggregation.

7 RELATED WORK

Distributed RL training. Recent research in the machine learning community proposes several distributed RL algorithms, such as A2C [41] and A3C [29]. These works, however, mainly focus on the single-node solution. In this paper, we target distributed RL training that involves multiple nodes at rack scale. The system community also shows interests in developing distributed RL systems, such as Gorila [33], Ray [32], and Horizon [11]. Our contributions on the in-switch acceleration are orthogonal to these works. iSwitch could be integrated into these systems to further improve their performance.

More recently, researchers propose to reduce the communication overhead of distributed training system by adopting the AllReduce technique [10, 24, 56], because AllReduce, especially Ring-AllReduce, is a bandwidth-optimal communication method which suits for bandwidth demanding workloads, such as distributed DNN training. Distributed RL training, however, is more sensitive to latency. Moreover, due to the global synchronization nature of AllReduce, it only supports synchronous training. iSwitch supports both synchronous and asynchronous training.

In-network computing. Researchers have proposed several solutions related to in-network computing in the high-performance computing (HPC) community, such as BlueGene Network [6], PERCS Interconnect [2], and CM-5 [22]. However, these works were developed using customized interconnect architecture for HPC applications. By contrast, iSwitch is designed with commodity programmable switches that have been widely adopted in data centers today [1, 4, 47, 50].

Recent research in networked systems has worked on the software-defined networking (SDN) with programmable switches, such as P4 [5] and PISCES [49]. SDN enables platform operators to implement their own networking policies in programmable switches to facilitate network management. However, none of previous works investigates the in-switch accelerator techniques for distributed machine learning workloads. iSwitch leverages the programmability of programmable switches to build an accelerator inside the switch, which conducts computation on packet payloads to facilitate the gradient aggregation for distributed RL training.

Hardware accelerators for machine learning. Recent advances in both academia and industry offer various specialized hardware accelerators for machine learning [20, 28, 44, 54, 55, 58, 59]. For instance, TABLA [28] uses FPGA accelerators to improve the performance of training workloads. Google presented the TPU [20] which is an ASIC accelerator with a systolic array architecture. CosMIC [44] proposes a distributed machine learning training system using multiple FPGA and ASIC accelerators. Most of these works, however, focus only on training or inference of DNNs without considering the RL. Importantly, these works accelerate only the local computation of machine learning algorithms, while leaving the network bottleneck in distributed training systems untouched. iSwitch sets out to reduce the network overhead with the proposed in-switch computing.

8 CONCLUSION

In this paper, we take an initial effort in quantifying the performance overhead in distributed RL training, and propose an in-switch computing paradigm, iSwitch, to remove the network bottleneck by providing a full set of solutions: (1) an in-switch aggregation accelerator to reduce the end-to-end communication overhead; (2) an acceleration support for both synchronous and asynchronous distributed RL training with improved parallelism; and (3) a hierarchical design for rack-scale clusters to scale the distributed RL training. The experiments with various RL workloads demonstrate that iSwitch offers a system-level speedup of up to 3.66× for synchronous distributed training, and 3.71× for asynchronous distributed training, compared with state-of-the-art approaches.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizon Network, and a gift fund from Intel Labs.

REFERENCES

[1] Alexey Andreyev. 2014. Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-\facebook-data-center-network/.

[2] L. Baba Arimilli, Ravi Arimilli, Vicente Chung, Scott Clark, Wolfgang E. Denzel, Ben C. Drerup, Torsten Hoefler, Jody B. Joyner, Jerry Lewis, Jian Li, Nan Ni, and Ramakrishnan Rajamony. 2010. The PERCS High-Performance Interconnect. In *Proceedings of the 18th IEEE Symposium on High Performance Interconnects (HOTI'10)*. Santa Clara, CA.

[3] Atari. 1972. Atari Games, https://www.atari.com/.

[4] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. Melbourne, Australia.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programmming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (July 2014).

[6] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. 2011. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. Seattle, WA.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. Lake Tahoe, NV.

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. [n.d.]. OpenAI Baselines. https://github.com/openai/baselines.

[9] Dulat Yerzat. 2018. DQN Adventure, https://github.com/higgsfield/RL-Adventure.

[10] Facebook. 2018. Writing Distributed Applications with PyTorch, https://pytorch.org/tutorials/intermediate/dist_tuto.html.

[11] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, and Xiaohui Ye. 2018. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *arXiv* arXiv/1811.00260 (2018). https://arxiv.org/abs/1811.00260

[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv* arXiv/1706.02677 (2017). https://arxiv.org/abs/1706.02677

[13] Network Working Group. 2001. Requirement for Comments: 3168, https://tools.ietf.org/html/rfc3168.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. Las Vegas, NV.

[15] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via A Stale Synchronous Parallel Parameter Server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*.

[16] Ilya Kostrikov. 2018. DDPG and NAF, https://github.com/ikostrikov/pytorch-ddpg-naf.

[17] Ilya Kostrikov. 2018. Pytorch-A2C-PPO-Acktr, https://github.com/ikostrikov/pytorch-a2c-ppo-acktr.

[18] Intel Corporation. 2017. Intel X540, https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-x540-t2-brief.html.

[19] Intel Corporation. 2017. Xeon CPU E5, https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors.html.

[20] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*. Toronto, Canada.

[21] J. Langford, A. J. Smola, and M. Zinkevich. 2009. Slow Learners are Fast. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems (NIPS'09)*. Vancouver, Canada.

[22] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W.Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. 1996. The Network Architecture of the Connection Machine CM-5. *J. Parallel and Distrib. Comput.* 33 (1996). http://www.sciencedirect.com/science/article/pii/S0743731596900337

[23] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*. Montreal, Canada.

[24] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Gerhard Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. 2018. A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. Fukuoka City, Japan.

[25] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NIPS'18)*. Montreal, Canada.

[26] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2017. Asynchronous Decentralized Parallel Stochastic Gradient Descent. *arXiv* arXiv/1710.06952v3 (2017). http://arxiv.org/abs/1710.06952v3

[27] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous Control with Deep Reinforcement Learning. *arXiv* abs/1509.02971 (2015). http://arxiv.org/abs/1509.02971

[28] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning. In *Proceedings of the 2016 IEEE International Symposium on High-Performance Computer Architecture (HPCA'16)*. Barcelona, Spain.

[29] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv* arXiv/1602.01783 (2016). http://arxiv.org/abs/1602.01783

[30] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXIv* arXiv/1312.5602 (2013). http://arxiv.org/abs/1312.5602

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. In *Nature*.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA.

[33] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv* arXiv/1507.04296 (2015). http://arxiv.org/abs/1507.04296

[34] NetFPGA-SUME. 2014. https://netfpga.org/site/#/systems/1netfpga-sume/details/.

[35] NetFPGA SUME Team. 2019. NetFPGA-SUME-Wiki, https://github.com/NetFPGA/NetFPGA-SUME-public.

[36] NETGEAR Corporation. 2017. ProSafe XS712T SWITCH, https://www.netgear.com/support/product/xs712t.aspx.

[37] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. 2017. Real-Time Machine Learning: The Missing Pieces. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*. Whistler, Canada.

[38] NVIDIA Corporation. 2018. NVIDIA CUDA C Programming Guide.

[39] NVIDIA Corporation. 2018. NVIDIA CuDNN https://developer.nvidia.com/cudnn.

[40] NVIDIA Corporation. 2019. NVIDIA TITAN RTX, https://www.nvidia.com/en-us/titan/titan-rtx/.

[41] OpenAI. 2017. OpenAI Baselines: ACKTR & A2C. https://blog.openai.com/baselines-acktr-a2c/

[42] OpenAI. 2018. OpenAI: GYM, https://gym.openai.com/.

[43] OpenMPI Community. 2017. OpenMPI: A High Performance Message Passing Library, https://www.open-mpi.org/.

[44] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-Out Acceleration for Machine Learning. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. Boston, MA.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'17)*.

[46] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'11)*.

[47] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*. London, UK.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv* arXiv/1707.06347 (2017). http://arxiv.org/abs/1707.06347

[49] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*. Florianopolis, Brazil.

[50] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*. London, UK.

[51] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *The international Journal of High Performance Computing Applications* 19 (2005).

[52] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2015. MuJoCo: A Physics Engine for Model-Based Control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vilamoura, Portugal.

[53] Paulina Varshavskaya, Leslie Pack Kaelbling, and Daniela Rus. 2008. Efficient Distributed Reinforcement Learning Through Agreement. In *Proceedings of the 9th International Symposium on Distributed Autonomous Robotic Systems (DARS)*. Tsukuba, Japan.

[54] Qian Wang, Youjie Li, and Peng Li. 2016. Liquid State Machine based Pattern Recognition on FPGA with Firing-Activity Dependent Power Gating and Approximate Computing. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'16)*. Montreal, Canada.

[55] Qian Wang, Youjie Li, Botang Shao, Siddhartha Dey, and Peng Li. 2017. Energy Efficient Parallel Neuromorphic Architectures with Approximate Arithmetic on FPGA. *Neurocomputing* 221 (2017). http://www.sciencedirect.com/science/article/pii/S0925231216311213

[56] Mingchao Yu, Zhifeng Lin, Krishna Giri Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander Schwing, Murali Annavaram, and Salman Avestimehr. 2018. GradiVeQ: Vector Quantization for Bandwidth-Efficient Gradient Aggregation in Distributed CNN Training. In *Proceedings of the 32nd Conference on Neural Information Processing Systems (NIPS'18)*. Montreal, Canada.

[57] Yuan Yu, Martin Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zhang. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the 13rd EuroSys Conference (EuroSys'18)*. Porto, Portugal.

[58] Xiaofan Zhang, Xinheng Liu, Anand Ramachandran, Chuanhao Zhuge, Shibin Tang, Peng Ouyang, Zuofu Cheng, Kyle Rupnow, and Deming Chen. 2017. High-Performance Video Content Recognition with Long-Term Recurrent Convolutional Network for FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. Ghent, Belgium.

[59] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*. Marrakech, Morocco.