# Accelerating Deep Learning using Multiple GPUs and FPGA-Based 10GbE Switch

Tomoya Itsubo*, Michihiro Koibuchi†, Hideharu Amano*, and Hiroki Matsutani*
*Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
Email: {itsubo@arc,hunga@am,matutani@arc}.ics.keio.ac.jp
†National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan 101-8430
Email: koibuchi@nii.ac.jp

***Abstract*—A back-propagation algorithm following a gradient descent approach is used for training deep neural networks. Since it iteratively performs a large number of matrix operations to compute the gradients, GPUs (Graphics Processing Units) are efficient especially for the training phase. Thus, a cluster of computers each of which equips multiple GPUs can significantly accelerate the training phase. Although the gradient computation is still a major bottleneck of the training, gradient aggregation and parameter optimization impose both communication and computation overheads, which should also be reduced for further shortening the training time. To address this issue, in this paper, multiple GPUs are interconnected with a PCI Express (PCIe) over 10Gbit Ethernet (10GbE) technology. Since these remote GPUs are interconnected via network switches, gradient aggregation and optimizers (e.g., SGD, Adagrad, Adam, and SMORMS3) are offloaded to an FPGA-based network switch between a host machine and remote GPUs; thus, the gradient aggregation and optimization are completed in the network. Evaluation results using four remote GPUs connected via the FPGA-based 10GbE switch that implements the four optimizers demonstrate that these optimization algorithms are accelerated by up to 3.0x and 1.25x compared to CPU and GPU implementations, respectively. Also, the gradient aggregation throughput by the FPGA-based switch achieves 98.3% of the 10GbE line rate.**

## I. Introduction

A gradient descent optimization algorithm with a back-propagation algorithm is used for training deep neural networks. It iteratively computes gradients of a loss function and optimizes weight parameters of the neural networks. The training phase performs a large number of matrix operations, and thus the computation cost is extremely high. GPUs (Graphics Processing Units) are typically used for accelerating the gradient computation. Actually, a cluster of computers each of which equips multiple GPUs has been used for reducing the training time [1][2][3].

In the parallel and distributed deep learning [4], in addition to the gradient computation parallelized by multiple GPUs, their gradients are aggregated and the weight parameters are then optimized based on the gradients by using a parameter optimization algorithm. Although the gradient computation is still a major computation bottleneck of the training, the gradient aggregation and parameter optimization impose both communication and computation overheads, which should also be reduced for further shortening the training time.

In this paper, we focus on the gradient aggregation and parameter optimization and accelerate them by using FPGA-based 10Gbit Ethernet (10GbE) switches. More specifically, multiple GPUs are interconnected with a PCI Express (PCIe) over 10GbE technology [5]. Since these remote GPUs are interconnected via network switches, the gradient aggregation and optimization algorithms are offloaded to the FPGA-based 10GbE switches between these remote GPUs. In this case, the gradient aggregation and parameter optimization are completed in the network. We implement the gradient aggregation and four optimization algorithms (i.e., SGD, Adagrad, Adam, and SMORMS3) on NetFPGA-SUME card [6] that has four 10GbE interfaces. The proposed FPGA-based switch is evaluated in terms of the gradient aggregation performance, parameter optimization performance, and FPGA resource utilization.

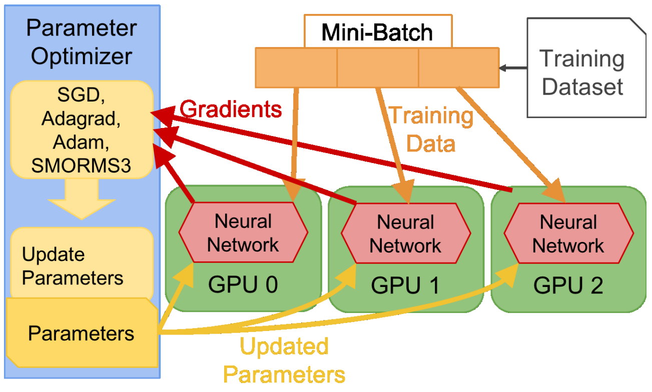

Fig. 1. Training with synchronous data parallel model

The rest of this paper is organized as follows. Section II introduces the parameter optimization algorithms, parallel and distributed deep learning approaches, and remote GPU technologies over Ethernet. Section III proposes the network switch that connects remote GPUs and performs the gradient aggregation and parameter optimization, and Section IV describes the implementation. Section V shows the evaluation results of performance and resource utilization. Section VI concludes this paper.

## II. Related Work

### A. Parallel and Distributed Training Models

As a distributed deep learning approach, this paper employs a synchronous data parallel model [4] that combines data parallel training and synchronous parameter optimization. In the data

**Algorithm 1** Algorithm of SGD

```
Require: $lr$: Learning rate
Require: $\theta_0$: Initial parameter
Require: $f(\theta)$: Loss function with parameter $\theta$
Require: $g(\theta)$: Gradient of loss function with parameter $\theta$
$t \leftarrow 0$
while Exit condition is not satisfied do
    $t \leftarrow t + 1$
    $g(\theta_t) \leftarrow \nabla_{\theta_t} f(\theta_t)$
    $\theta_t \leftarrow \theta_{t-1} - lr \cdot g(\theta_t)$
end while
```

**Algorithm 2** Algorithm of Adagrad

```
Require: $lr$: Initial learning rate
Require: $\theta_0$: Initial parameter
Require: $f(\theta)$: Loss function with parameter $\theta$
Require: $g(\theta)$: Gradient of loss function with parameter $\theta$
Require: $\delta$: Small constant
$t \leftarrow 0$
$h \leftarrow 0$
while Exit condition is not satisfied do
    $t \leftarrow t + 1$
    $g(\theta_t) \leftarrow \nabla_{\theta_t} f(\theta_t)$
    $h_t \leftarrow h_{t-1} + g^2(\theta_t)$
    $lr_t \leftarrow \frac{lr}{\sqrt{h_t} + \delta}$
    $\theta_t \leftarrow \theta_{t-1} - lr_t \cdot g(\theta_t)$
end while
```

parallel model, training data is divided and assigned to GPU workers, so that the training phase is performed in parallel by sharing intermediate training results. Then, the gradients separately computed by GPU workers are combined so as to optimize weight parameters of a model. Since a synchronous model is assumed, all the GPU workers are synchronized when they finish the gradient computation of errors for their assigned mini-batch. Then, the gradients are aggregated and weight parameters are optimized.

Figure 1 illustrates an outline of a training phase of the synchronous data parallel model. Below is the procedure using multiple GPU workers.

1) A certain amount of training data is picked up as a mini-batch.
2) The mini-batch is assigned to each GPU worker so that it processes the assigned training data with a neural network.
3) Each GPU worker computes error gradients, and they are aggregated in a host machine.
4) Weight parameters are optimized with the aggregated gradients, and then the optimized parameters are distributed to all the GPU workers.

### *B. Parameter Optimization Algorithms*

A gradient method is used for the parameter optimization of neural networks so that an error calculated by a loss function is minimized. There are various algorithms to compute new

**Algorithm 3** Algorithm of Adam

```
Require: $lr$: Learning rate
Require: $\theta_0$: Initial parameter
Require: $f(\theta)$: Loss function with parameter $\theta$
Require: $g(\theta)$: Gradient of loss function with parameter $\theta$
Require: $\delta$: Small constant
Require: $\beta_1, \beta_2 \in [0, 1)$ :
$t \leftarrow 0$
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
while Exit condition is not satisfied do
    $t \leftarrow t + 1$
    $g(\theta_t) \leftarrow \nabla_{\theta_t} f(\theta_t)$
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g(\theta_t)$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g^2(\theta_t)$
    $\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$
    $\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$
    $\theta_t \leftarrow \theta_{t-1} - lr \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \delta}$
end while
```

**Algorithm 4** Algorithm of SMORMS3

```
Require: $lr$: Learning rate
Require: $\theta_0$: Initial parameter
Require: $f(\theta)$: Loss function with parameter $\theta$
Require: $g(\theta)$: Gradient of loss function with parameter $\theta$
Require: $\delta$: Small constant
$t \leftarrow 0$
$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$s_0 \leftarrow 1$
while Exit condition is not satisfied do
    $t \leftarrow t + 1$
    $g(\theta_t) \leftarrow \nabla_{\theta_t} f(\theta_t)$
    $s_t \leftarrow 1 + (1 - x_{t-1} \cdot s_{t-1})$
    $\rho_t \leftarrow \frac{1}{s_t + 1}$
    $m_t \leftarrow \rho_t \cdot m_{t-1} + (1 - \rho_t) \cdot g(\theta_t)$
    $v_t \leftarrow \rho_t \cdot v_{t-1} + (1 - \rho_t) \cdot g^2(\theta_t)$
    $x_t \leftarrow fracm_t^2 v_t + \delta$
    $\theta_t \leftarrow \theta_{t-1} - \frac{\min\{lr, x_t\}}{\sqrt{v_t} + \delta} \cdot g(\theta_t)$
end while
```

weight parameters based on gradients of the loss function and current parameters. A well-known algorithm is SGD (Stochastic Gradient Descent) listed in Algorithm 1. In SGD, gradients multiplied by a learning rate are subtracted from current weight parameters to compute new parameters. Although SGD is simple, it may converge to a local solution with low accuracy or require a number of iterations to convergence, depending on a selected learning rate. A careful tuning is thus required for selecting the hyperparameters.

To improve SGD in terms of accuracy and the number of iterations to convergence, variants of SGD have been invented. Typical examples of such algorithms are Adagrad [7], Adam [8], and SMORMS3 [9]. They are listed in Algorithms 2, 3,

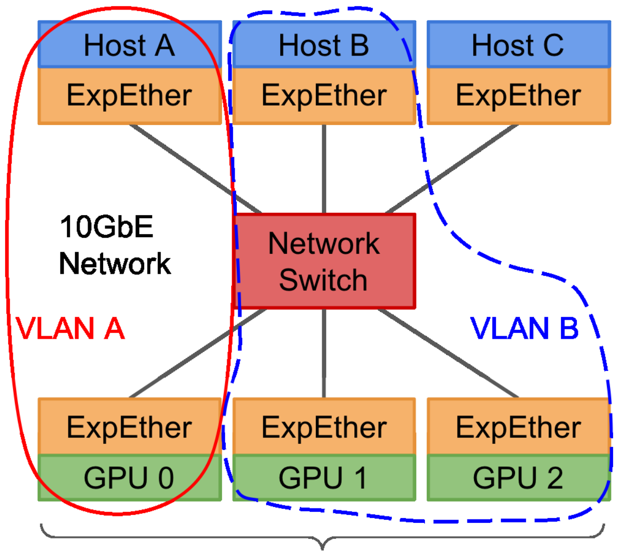

Fig. 2. Connection between hosts and GPUs using ExpEther [5]

and 4. They are widely used to optimize a model with higher accuracy and smaller number of iterations to convergence. Since computation costs for these algorithms are higher than that of SGD, although they can reduce the number of iterations, they incur a longer computation time for each iteration.

### C. Distributed Deep Learning Using GPUs

A large-scale distributed deep learning has been efficiently executed on GPU clusters. In a GPU cluster, nodes consist of a host machine with several GPUs and they are interconnected with a high-speed network, such as 10GbE. In [1], a GPU cluster consisting of 128 nodes with 1,024 GPUs completed a training phase of ImageNet in 15 minutes. In [2] and [3], GPU clusters using 4,352 GPUs and 2,048 GPUs completed the training phase of ImageNet in 122 seconds and 74.7 seconds, respectively. They use MPI AllReduce for communication between the nodes. Although the communication is efficiently done by using Allreduce, the communication overhead increases as the number of nodes increases, and the execution time per iteration increases. In [1], approximately 20% of execution time is spent for communication at 128 nodes.

In [10], it is reported that a significant portion of training time is spent for the communication. To reduce the communication overhead, nodes in a cluster circulate their gradients to aggregate them within a cluster without a specific aggregation node. Also, since the gradients are more tolerant of accuracy loss than weight parameters, a compression technique is applied to the gradients to reduce the communication overhead.

### D. Network-Attached GPUs

Since the number of PCIe (PCI Express) slots in a single machine is limited, the number of required host machines is increased as the number of GPUs increases, resulting in a higher cost and power consumption. To mitigate this limitation, GPUs can be connected to network switches and accessed via a high-speed network remotely by using PCIe over Ethernet technology.

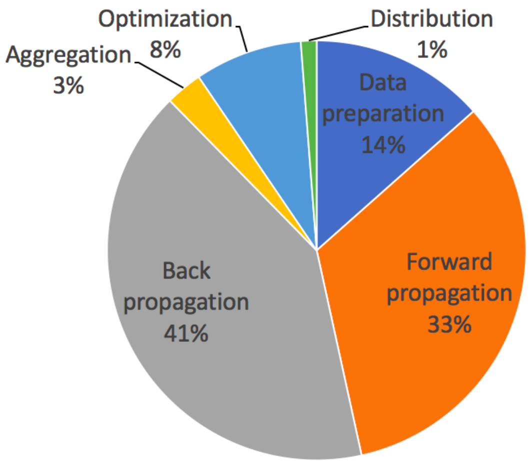

Fig. 3. Breakdown of execution times in training phase

TABLE I
PRELIMINARY EVALUATION ENVIRONMENT

| | |
|---|---|
| CPU | Intel Core i7-6850K @3.6GHz |
| Memory | 32GB |
| GPU | NVIDIA Geforce GTX 1080 (8GB RAM) x4 |
| CUDA | version 10.0 |
| Chainer | version 6.2.0 |

As a PCIe over 10GbE (10Gbit Ethernet) technology, in this paper we use ExpEther 10G [5] that can extend PCIe over 10GbE. PCIe devices in a 10GbE network can be assigned to the host machines as shown in Figure 2. Such network-attached GPUs have been used to accelerate a large-scale graph processing [11]. In this paper, we employ network-attached GPUs for accelerating the gradient computation, while the gradient aggregation and parameter optimization are accelerated by using network-attached FPGA.

## III. FPGA-BASED SWITCH DESIGN

### A. Preliminary Evaluations

This section proposes an FPGA-based acceleration of the gradient aggregation and parameter optimization in distributed deep learning. First, a preliminary evaluation is conducted to show execution times for the aggregation and parameter optimization in an overall training phase. To measure the execution time, Chainer [12] is used as a deep learning framework. Table I shows the preliminary evaluation environment. GoogleNet is used as the DNN model, and Adam is used as a parameter optimization algorithm. A synchronous data parallel model is used in this evaluation.

Figure 3 shows a breakdown of the execution time. The execution time for each iteration is divided into six parts: data preparation, forward propagation, back propagation, gradient aggregation, parameter optimization, and parameter distribution. Gradient computation (i.e., forward propagation and back propagation) are executed by four GPUs, while parameter optimization using Adam is done by a single GPU. The other parts, such as gradient aggregation, are done by a host CPU.

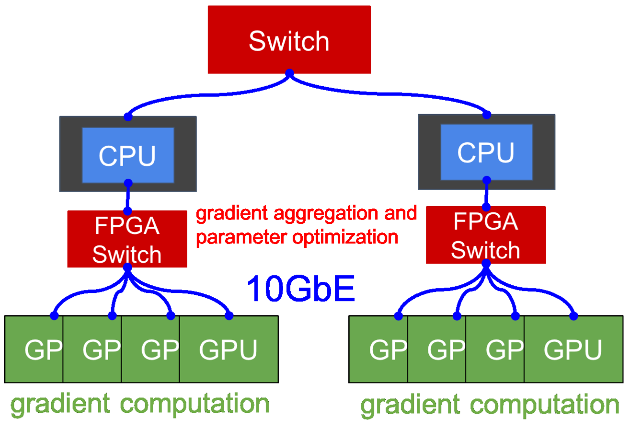

Fig. 4. Proposed system (gradient computation is done by "GPUs" and gradient aggregation and parameter optimization are done by "Switches")

As shown in Figure 3, the execution times for the compute-intensive parts, such as forward propagation and back propagation, are large. Those for the gradient aggregation and parameter optimization account for approximately 10% even with four GPUs. Although they are not a major bottleneck in the case of four GPUs, their execution times would be increased as the number of GPUs increases. Thus, their execution times should be reduced for further shortening the training time.

## B. System Overview

To further reduce the execution times of distributed deep learning, in this paper we propose to use network-attached GPUs for the gradient computation, and offload the gradient aggregation and parameter optimization to network-attached FPGA in a network.

When a host machine accesses a remote GPU, input data and output data to/from the GPU go through one or more network switches. A high speed data processing can be performed during a communication between the host machine and the GPU by incorporating in the network switch. In this paper, we propose to offload the gradient aggregation and parameter optimization to an FPGA-based 10GbE switch. Conventionally, this gradient aggregation and parameter optimization are processed by CPU. Figure 4 illustrates the proposed system, where remote GPUs using PCIe over 10GbE are interconnected via an FPGA-based network switch. The gradient aggregation and parameter optimization are thus completed in the middle of communication with low overheads.

Since the 10GbE bandwidth is narrower than PCIe Gen3 x16, data transfer time may increase in the proposed system. This issue can be mitigated by using recent ExpEther 40G product. Actually, offloading the gradient aggregation and parameter optimization to a network switch can compensate for the lower bandwidth and efficiently improve their execution times, as shown in Section V.

## C. Gradient Aggregation Function

Here, data flow of the gradient aggregation in the proposed system is described below. As shown in Figure 4, GPU workers compute gradients and then send them to a host machine (denoted as "CPU") via the FPGA-based network switch (denoted as "Switch"). The FPGA-based switch extracts the gradients from ExpEther packets sent from GPUs to a host. Then the gradients are aggregated and sent to the host. The host machine thus receives already-aggregated gradients from the switch.

In large-scale distributed deep learning, a single host machine will be a bottleneck, and thus multiple host machines should be used as shown in Figure 4. In this case, these nodes perform AllReduce to exchange their already-aggregated gradients. Even with such multiple host cases, gradient aggregation overheads are greatly reduced by the proposed network switch.

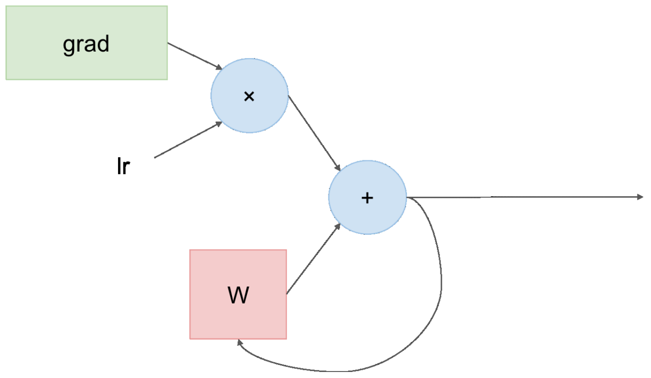

Fig. 5. Computation graph of SGD

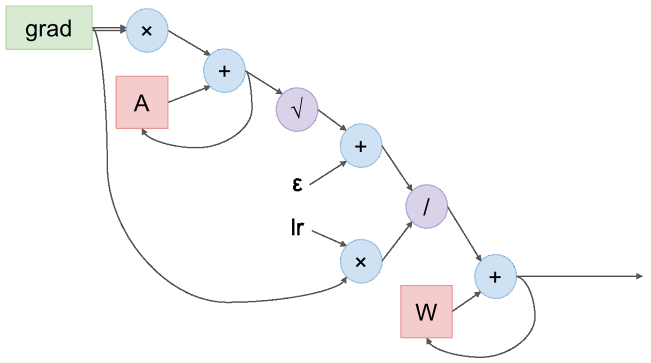

Fig. 6. Computation graph of Adagrad

## D. Parameter Optimization Function

The parameter optimization is performed in the proposed switch after all the gradients are aggregated in the network. As parameter optimization algorithms, SGD, Adagrad, Adam, and SMORMS3 are implemented on the FPGA-based switch. These algorithms are selected so as to cover both the simple algorithm (i.e., SGD) and sophisticated algorithm that requires relatively higher computation cost (i.e., SMORMS3).

Figures 5, 6, 7, and 8 show their computation graphs, respectively. In the computation graphs, circle symbols represent computational operations, such as addition, subtraction, multiplication, division, square root, and minimum. Square symbols represent storage elements, each of which is corresponding to a floating point number. *grad* represents input gradients and W represents weight parameters before optimization. The

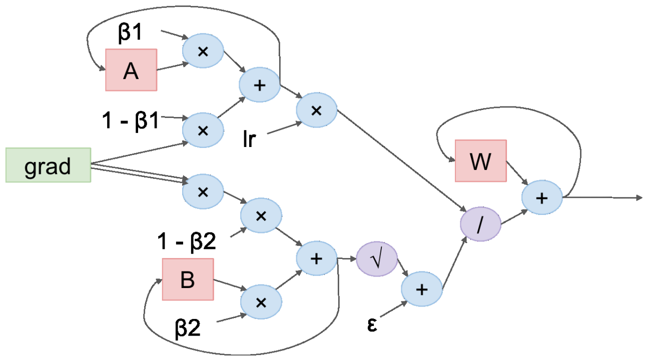

Fig. 7. Computation graph of Adam

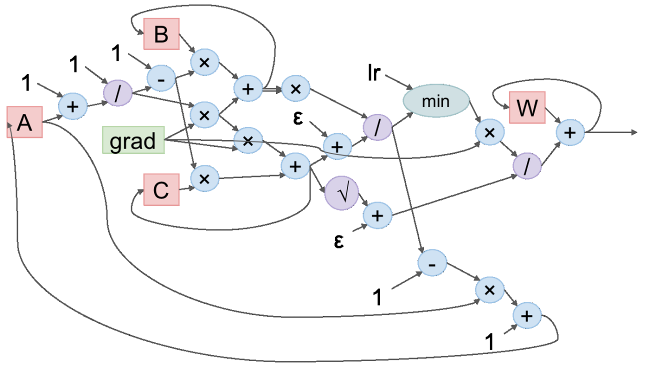

Fig. 8. Computation graph of SMORMS3

other symbols, such as A, B, and C, are algorithm-specific parameters.

## IV. Implementation

### A. GPU Packet Trace

Prior to the implementation, this section describes packet traces of remote GPUs connected via a PCIe over 10GbE technology. The packet traces were analyzed so that we can extract the gradients from ExpEther packets and write back optimized weight parameters to the packets. The packets were also used for the evaluations of the proposed FPGA-based network switch.

Since NVIDIA's GPUs are used for the gradient computation, CUDA (Compute Unified Device Architecture) [13] is used as an integrated development environment for the remote GPUs. In a CUDA program, a cudaMemcpy function copies data from host main memory to GPU device memory. The instruction and data are transferred as PCIe over 10GbE packets and go through the proposed FPGA-based network switch. We collected the PCIe over 10GbE packet traces between a host machine and remote GPU devices.

Figure 9 shows the packet capture environment. As shown, there are three machines: a host machine, a switch machine, and a capture machine. The host machine is equipped with an ExpEther host adapter. It executes a CUDA program for a remote GPU connected via the proposed network switch. The switch machine is equipped with a NetFPGA-SUME card that has a Xilinx Virtex-7 XC7VX690T FPGA and four SPF+ connectors for 10GbE interfaces. This FPGA card is used as a

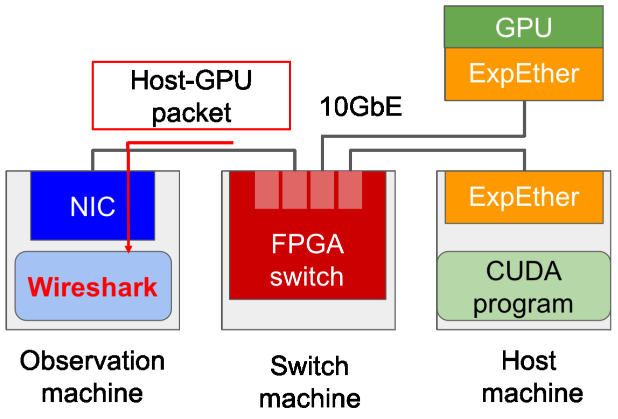

Fig. 9. Packet capture environment

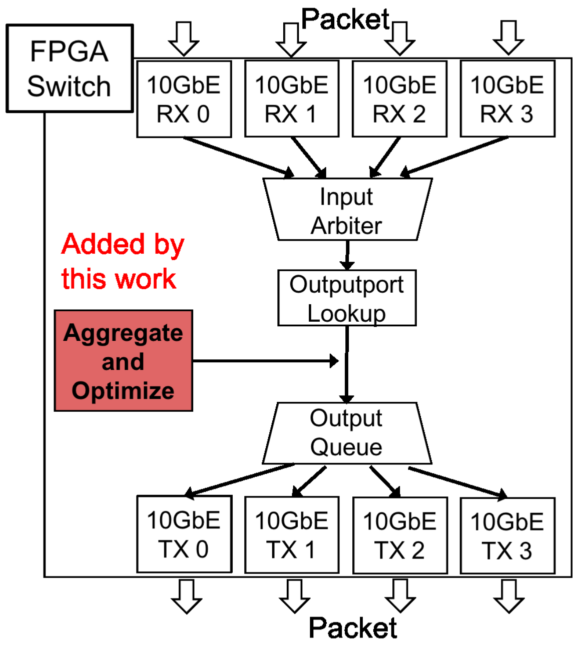

Fig. 10. Block diagram of FPGA-based network switch

4-port 10GbE switch. The packet capture machine is equipped with a 10GbE network interface card connected to the network switch. Wireshark is used as a packet capture software at the capture machine. In this environment, when a CUDA program is executed on the host machine for the remote GPU, PCIe over 10GbE packets are transferred between the host machine and the remote GPU. These packets go through the network switch and are captured by the capture machine.

### B. Baseline Network Switch

Reference Switch Lite design provided by NetFPGA team [6] is used as a baseline 10GbE switch implemented on NetFPGA-SUME card. The gradient aggregation and parameter optimization modules are inserted to this baseline switch. Figure 10 shows a block diagram of the FPGA-based network switch. When packets are injected to the network switch, they are passed from one of four 10GbE interfaces (i.e., RX0 to

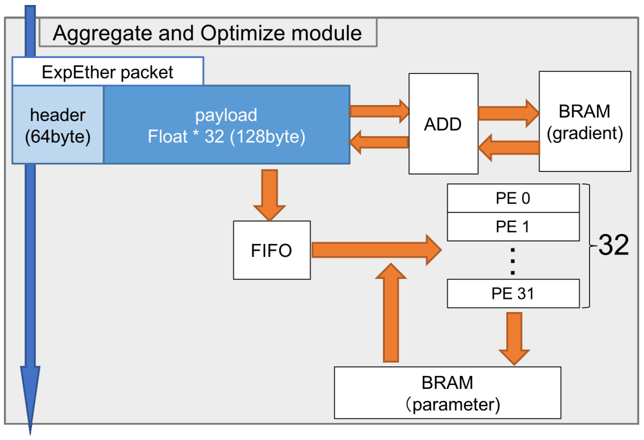

Fig. 11. Gradient aggregation and parameter optimization modules

RX3). Input Arbiter module receives packets one by one in a round robin manner from these interfaces. Next, Outputport Lookup module performs a routing function based on a packet header to determine a destination port of the switch. The gradient aggregation and parameter optimization modules judge if incoming packets are subject to the gradient aggregation, parameter optimization, or none of them. The gradient aggregation and parameter optimization modules are explained in the next subsections. Finally, Output Queue module distributes packets to one of the four 10GbE ports according to the routing result by Outputport Lookup module.

## C. Gradient Aggregation and Parameter Optimization Modules

Figure 11 illustrates the gradient aggregation and parameter optimizations modules proposed in this paper. In these modules, first, based on the ExpEther packet format captured in Section IV-A, incoming packets are analyzed to see if they are subject to the gradient aggregation or parameter optimization modules. If the packet is not related to the aggregation nor parameter optimization, the packet simply skips these modules.

If the packet is subject to the gradient aggregation, the sequence number of the gradient field in the packet is read, and the corresponding gradient is retrieved from a BRAM that stores the aggregated gradients. Then, the gradients of the packet and those retrieved from the BRAM are added. After the addition, the result is written back to both the packet and BRAM, and then the packet is sent to the next module.

If the packet is subject to the parameter optimization, in the same way, the sequence number of the gradient field in the packet is read, and the corresponding parameter is retrieved from a BRAM that stores the parameter. The gradients extracted from the packet and the parameters extracted from the BRAM are fed to PEs, and the parameter optimization is performed. After the parameter update is completed, new parameters are stored in the BRAM.

## D. Parameter Optimization Algorithms

The parameter optimizer in the 10GbE switch is implemented on NetFPGA-SUME card. The target FPGA device is Xilinx Virtex-7 XC7VX690T. Xilinx Vivado v2016.4 is used for logic synthesis and implementation. The target operating frequency is 200MHz. As arithmetic IP cores, Floating-Point Operator v7.0 provided by Xilinx is used for these algorithms. These IP cores include addition, subtraction, multiplication, division, size comparison, and square root operations. They use 32-bit single-precision floating-point numbers. The four optimization algorithms (i.e., SGD, Adagrad, Adam, and SMORMS3) are implemented on NetFPGA-SUME card. These algorithms are implemented by combining and/or cascading the above-mentioned arithmetic IP cores as shown in Figures 5 to 8.

TABLE II
LATENCIES OF PARAMETER OPTIMIZATION ALGORITHMS

| Optimizer | # of cycles |
|---|---|
| SGD | 21 |
| Adagrad | 99 |
| Adam | 137 |
| SMORMS3 | 164 |

TABLE III
FPGA RESOURCE UTILIZATIONS OF OPTIMIZATION ALGORITHMS

| Optimizer | LUTs | FFs | DSPs |
|---|---|---|---|
| SGD | 535 (0.12%) | 953 (0.11%) | 3 (0.08%) |
| Adagrad | 2,695 (0.62%) | 4,880 (0.56%) | 9 (0.25%) |
| Adam | 5,409 (1.25%) | 10,278 (1.19%) | 36 (1.00%) |
| SMORMS3 | 6,100 (1.41%) | 11,510 (1.33%) | 40 (1.11%) |

The latencies (the number of clock cycles) to complete these algorithms are listed in Table II. Here, the latency is a duration between when a single-precision floating-point input data is injected and when the corresponding computation result is generated.

Please note that the above-mentioned parameter optimization cores are in charge of single input data only. To accelerate the parameter optimization algorithms, multiple instances or PEs (processing elements) of these optimizer cores are implemented, as shown in Figure 12. Floating-point numbers of input data are distributed to these PEs in a round-robin manner and processed in parallel. Throughput of the parameter optimization increases as the number of the optimizer PEs is increased, as long as the parallelism of input data can be exploited; thus there is a trade-off between the throughput and the resource utilizations.

The four optimization algorithms are evaluated in terms of the FPGA resource utilizations of LUTs (Look Up Tables), FFs (Flip Flops), and DSP (Digital Signal Processing) slices. Table III shows the result. In this implementation, 32 gradients are included in a single ExpEther packet, and thus 32 PEs should be implemented to fully exploit the parallelism of input data. As shown in Table III, the PE sizes are quite small, and thus we implemented 32 PEs for each algorithm on the FPGA-based network switch.

## V. EVALUATIONS

## A. Gradient Aggregation Throughput

First, the proposed FPGA-based network switch is evaluated in terms of the gradient aggregation throughput. Test packets including gradients in 10GbE line rate are generated by using

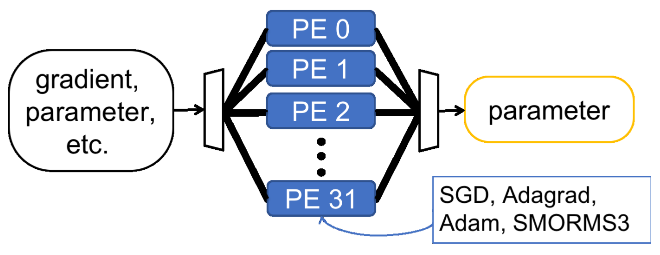

Fig. 12. Multi-PE implementation of optimization algorithms

TABLE IV
FPGA RESOURCE UTILIZATION OF PROPOSED SWITCH

| Optimizer | LUTs | FFs | DSPs |
|---|---|---|---|
| SGD | 66,775 (15.36%) | 104,582 (11.68%) | 96 (2.67%) |
| Adagrad | 135,895 (31.32%) | 230,246 (26.19%) | 288 (8.00%) |
| Adam | 222,743 (51.37%) | 402,982 (46.13%) | 1,152 (32.00%) |
| SMORMS3 | 244,855 (56.47%) | 442,406 (50.68%) | 1,280 (35.56%) |

Open Source Network Tester [14] and sent to the proposed network switch. The gradients are represented as an array of 32-bit single-precision floating-point numbers. In this evaluation, each packet contains 32 gradients and thus the packet length is 192 bytes including a packet header and the payload. Open Source Network Tester is directly connected to the proposed network switch with a 10GbE SFP+ cable, and the gradient aggregation is executed on the switch. The aggregation throughput is measured by counting the number of packets processed by the proposed switch.

The measurements are performed ten times and the average throughput is 8.92Gbps. Assuming the packet length is 192 bytes, the 10GbE line rate in our environment is 9.07Gbps when considering the Ethernet preamble and interframe gap inserted for each packet. In this case, the measured throughput of the gradient aggregation is corresponding to 98.3% of the 10GbE line rate in our environment, and thus almost the line rate is achieved.

### B. *Resource Utilization*

The proposed network switch including the gradient aggregation and parameter optimization modules is evaluated in terms of FPGA resource utilizations of LUTs, FFs, and DSP slices. Table IV shows the resource utilizations of an entire switch in the cases of the four parameter optimization algorithms. The resource utilization of SGD version is the lowest. It consumes approximately 15% of LUTs. Adam and SMORMS3 versions consume more resources. Especially, SMORMS3 version consumes approximately 56% of LUTs, but even with Reference Switch modules, their resource utilizations still have room to add more PEs.

### C. *Parameter Optimization Latency*

In general, the parameter optimization is done by a host CPU or GPU after the workers compute gradients. By introducing the proposed network switch, the parameter optimization is completed in the middle of communication path between host CPU and remote GPUs. In this section, the proposed network switch is evaluated in terms of the execution time of the parameter optimization.

TABLE V
CPU- AND GPU-BASED EXECUTION ENVIRONMENT

| | |
|---|---|
| OS | Ubuntu 16.04 |
| CPU | Intel Core i7-6800K @3.4GHz |
| Memory | 32GB |
| GPU | NVIDIA Geforce GTX 1080Ti (11GB RAM) x4 |
| CUDA | version 9.0 |
| Chainer | version 5.2.0 |

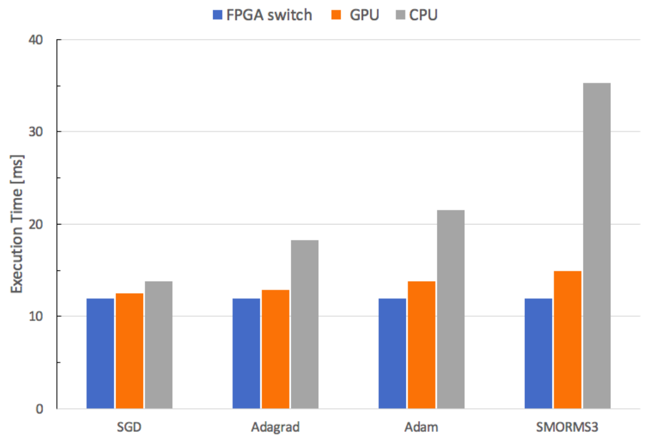

Fig. 13. Execution time of parameter optimization

The proposed network switch is compared with the following CPU and GPU-based approaches in terms of the parameter optimization.

1) CPU-based approach: Assuming a host CPU has already-aggregated gradients, the execution time for the parameter optimization by the CPU is measured.
2) GPU-based approach: After a GPU device receives already-aggregated gradients from a host machine, the execution time for the parameter optimization by the GPU is measured.
3) Proposed FPGA-based network switch: Assuming the proposed network switch is placed in a communication path between a host CPU and a remote GPU device, the number of cycles for the parameter optimizations by the network switch is measured.

The CPU- and GPU-based approaches are implemented as a software program in Chainer [12]. Their evaluation environment is listed in Table V [1]. In this evaluation, 800,000 parameters are assumed, so that the latest DNN models, such as DenseNet-BN [15], can be supported. SGD, Adagrad, Adam, and SMORMS3 algorithms are executed for 800,000 gradients using these three approaches.

Figure 13 shows the execution times of the parameter optimization of the four algorithms with the three approaches: CPU-based, GPU-based, and the proposed FPGA-based network switch approaches. In this graph, X-axis represents the algorithms and Y-axis represents their execution times. In the case of SGD, differences between the three approaches are small. However, the differences become large as the algorithm

[1]The machine used slightly differs from that used in the preliminary evaluation in Section III-A due to availability of the machine.

becomes complicated; in the case of SMORMS3, the differences are the largest. As a result, the proposed network switch approach outperforms the CPU- and GPU-based approaches by 1.2-3.0x and 1.05-1.25x, respectively.

In the proposed network switch approach, the differences between the four optimization algorithms are also quite small. This is because the serialization of input gradients is a major bottleneck in the proposed network switch regardless of the optimization algorithm selected. Among the four optimization algorithms, it turns out that the use of sophisticated algorithms (e.g., SMORMS3) is beneficial in the proposed network switch approach compared to the CPU- and GPU-based approaches.

### D. Discussions

Although a major bottleneck of the training phase is still the gradient computation by GPU workers, here we estimate how much an entire training phase can be accelerated by introducing the proposed network switch in the case of the preliminary evaluation in Section III-A. Using the proposed network switch, the gradient aggregation and parameter optimization are performed during a communication between a host machine and remote GPU devices, so "Aggregation and Optimization" part of Figure 3 is accelerated. The proposed network switch performs the gradient aggregation in 98.5% of the 10GbE line rate, and it accelerates the parameter optimization of Adam algorithm by approximately 1.2x compared to the GPU-based approach. Based on these results, an entire training phase is shortened by approximately 5% in the case of the preliminary evaluation in Section III-A. In addition to this speedup, since the gradient aggregation and parameter optimization are offloaded to the network switch, the saved CPU and GPU resources can be used for the other tasks.

In this paper, only four GPUs are used for the evaluations, but more GPUs are typically used in distributed deep learning. As the number of GPUs increases, execution times for the forward propagation and back propagation decrease, but that for the gradient aggregation increases. In [1], it is reported that when 1,024 GPUs are used for large-scale distributed deep learning, the communication part consumes approximately 15% of an entire training time. Furthermore, since the execution time for the parameter optimization cannot be accelerated as the number of GPUs is increased, the proposed network switch can accelerate especially such large-scale distributed deep learning with many GPUs.

## VI. Summary

In this paper, the gradient aggregation and parameter optimization were accelerated by an FPGA-based network switch for distributed deep learning using remote GPUs via PCIe over 10GbE. In distributed deep learning, the number of GPUs is typically increased in order to increase the degree of parallelism and shorten the training time. However, communication overheads including gradient aggregation cannot be ignored in large-scale distributed deep learning. In this paper, we thus introduced remote GPUs via PCIe over 10GbE and reduced the overhead by offloading the gradient aggregation and parameter optimization to the FPGA-based network switch placed in the middle of communication. There are several parameter optimization algorithms with different characteristics, such as computation cost and the number of iterations to convergence. Four parameter optimization algorithms including SGD, Adagrad, Adam, and SMORMS3 were implemented in the proposed network switch.

Evaluation results of the proposed FPGA-based network switch demonstrated that the gradient aggregation achieved 98.3% of the 10GbE line rate. As for the parameter optimization, the proposed network switch outperformed CPU- and GPU-based approaches by approximately 1.2-3.0x and 1.05-1.25x, respectively. Also, we estimated that the overall training phase would be accelerated by approximately 5% by introducing the proposed network switch. As a future work, we are planning to demonstrate the performance improvement of overall training phase in a real environment that consists of a host machine, four remote GPUs via PCIe over 10GbE, and the proposed FPGA-based network switch.

**Acknowledgements** This work was partially supported by JSPS KAKENHI Grant Number JP19H01106 and JST CREST Grant Number JPMJCR1785, Japan.

## References

[1] T. Akiba, S. Suzuki, and K. Fukuda, "Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes," *CoRR*, vol. abs/1711.04325, Nov 2017.

[2] H. Mikami, H. Suganuma, P. U-chupala, Y. Tanaka, and Y. Kageyama, "Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash," *CoRR*, vol. abs/1811.05233, Nov 2018.

[3] M. Yamazaki, A. Kasagi, A. Tabuchi, T. Honda, M. Miwa, N. Fukumoto, T. Tabaru, A. Ike, and K. Nakashima, "Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds," *CoRR*, vol. abs/1903.12650, Mar 2019.

[4] M. Abadi *et al.*, "Tensorow: A system for large-scale machine learning," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Nov 2016, pp. 265–283.

[5] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata, "ExpressEther - Ethernet-Based Virtualization Technology for Reconfigurable Hardware Platform," in *Proceedings of the IEEE Symposium on High-Performance Interconnects (HOTI'06)*, Aug 2006, pp. 45–51.

[6] "The NetFPGA Project," https://netfpga.org.

[7] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul 2011.

[8] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, vol. abs/1412.6980, Dec 2014.

[9] "RMSprop loses to SMORMS3," https://sifter.org/~simon/journal/20150420.html.

[10] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A Network-Centric Hardware/Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks," in *Proceedings of the International Symposium on Microarchitecture (MICRO'18)*, Oct 2018, pp. 175–188.

[11] S. Morishima and H. Matsutani, "High-Performance with an In-GPU Graph Database Cache," *IEEE IT Professional*, vol. 19, no. 6, pp. 58–64, Nov 2017.

[12] "Chainer: A Powerful, Flexible, and Intuitive Framework for Neural Networks," https://chainer.org.

[13] "NVIDIA CUDA," https://developer.nvidia.com/cuda-zone.

[14] "OSNT 10G Home," https://github.com/NetFPGA/OSNT-Public/wiki/OSNT-10G-Home.

[15] G. Huang, Z. Liu, L. van der Matten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*, Jul 2017, pp. 2261–2269.