electronics

MDPI

Article

MemBox: Shared Memory Device for Memory-Centric Computing Applicable to Deep Learning Problems

Yongseok Choi [1], Eunji Lim [1], Jaekwon Shin [2] and Cheol-Hoon Lee [3,*]

[1] Artificial Intelligence Research Laboratory, ETRI, Daejeon 34129, Korea; shine24@etri.re.kr (Y.C.); ejlim@etri.re.kr (E.L.)
[2] Aviation Drone Laboratory, LIG Nex1, Yongin 16961, Korea; jaekwon.shin@lignex1.com
[3] Department of Computer Engineering, Chungnam National University, Daejeon 34134, Korea
* Correspondence: clee@cnu.ac.kr

Abstract: Large-scale computational problems that need to be addressed in modern computers, such as deep learning or big data analysis, cannot be solved in a single computer, but can be solved with distributed computer systems. Since most distributed computing systems, consisting of a large number of networked computers, should propagate their computational results to each other, they can suffer the problem of an increasing overhead, resulting in lower computational efficiencies. To solve these problems, we proposed an architecture of a distributed system that used a shared memory that is simultaneously accessible by multiple computers. Our architecture aimed to be implemented in FPGA or ASIC. Using an FPGA board that implemented our architecture, we configured the actual distributed system and showed the feasibility of our system. We compared the results of the deep learning application test using our architecture with that using Google Tensorflow's parameter server mechanism. We showed improvements in our architecture beyond Google Tensorflow's parameter server mechanism and we determined the future direction of research by deriving the expected problems.

Keywords: distributed system; shared memory; deep learning; big data; FPGA; ASIC

Citation: Choi, Y.; Lim, E.; Shin, J.; Lee, C.-H. MemBox: Shared Memory Device for Memory-Centric Computing Applicable to Deep Learning Problems. *Electronics* **2021**, *10*, 2720. https://doi.org/10.3390/electronics10212720

Academic Editors: Andrea Prati, Carlos A. Iglesias, Luis Javier García Villalba and Vincent A. Cicirello

Received: 29 September 2021
Accepted: 6 November 2021
Published: 8 November 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Distributed systems have components that are located on different computers that are networked with each other to communicate and coordinate actions by passing messages to one another [1]. The computational problems that cannot be processed in time using a single computer are reconstructed into a distributed model and processed using a distributed system, which employs the processing power of multiple interconnected computers.

Distributed systems collect and use independent resources, such as CPUs, memories, and storage, for computing and data processing and they work continuously despite some performance degradation that occurs with failure in some computing resources.

Since ARPANet has appeared, the earliest distributed system was made using Ethernet interconnection between computers. By enhancing the interconnection performance using InfiniBand, this system is adopted by data centers and other collective computers.

In the Ethernet- or InfiniBand-based distributed systems, data exchange between computers uses an MPI (message passing interface). An MPI solves problems by processing data at each computer and sending the results to a computer that aggregates the data. It assumes that locality is guaranteed, thereby each computer handles data without regard to their correlation. However, as computational problems (for example, deep learning or big data analysis) increase data dependencies and reduce the localities of data, a mechanism is required that allows all computers to have free access to huge amounts of data. However, MPI is still commonly used in distributed systems, though incorporating an MPI introduces inefficiency in a complex computational environment. To overcome this inefficiency, Google Tensorflow incorporated the Parameter Server mechanism, which has

a dedicated parameter server in cluster systems that are interconnected by Ethernet or InfiniBand and, using a parameter server, parameters are centralized in the distributed system. However, in the case of a large DNN, many workers on many nodes are required, and thus communication to the parameter server increases, which creates a large amount of network traffic overhead, resulting in overall performance degradation.

We proposed a new type of distributed system using our proprietary hardware-based shared memory (we called it Membox). It replaces a parameter server that uses a legacy network and software stack to construct a logically common data storage and allows for access to common data using memory read/write or DMA without requiring an unnecessary server software stack.

The MemBox system provides a common memory that can be accessed by all compute nodes and avoid the waste of data storage space that needs to be maintained by each node in a network-based system. By providing a dedicated port to access common data for each node, all compute nodes can access common data without significant performance degradation.

This paper begins by discussing the state-of-the-art technologies and their limitations for making the distributed system (Section 2); Section 3 discusses requirements for the distributed system and describes related research to the core technology of the constructed distributed system.

Section 4 describes the architecture of the proposed system and the components and Section 5 describes the implementation of our system.

Section 6 shows the experimental results of the deep learning application tests for our system compared with the existing system.

Finally, Section 7 concludes our work and outlines a possible future direction of the research.

2. State of the Art in Distributed Systems

A distributed system using MapReduce maps data to each computer, solves problems on each compute node, and then reduces them to find solutions to the whole problem [2], leveraging an MPI between each computer. However, a major problem is that frequent data exchanges occur, making it difficult to solve one huge problem because it breaks the relevance between data when it is decomposed and mapped to each computer. Therefore, it is not suitable for tasks such as those that combine multiple inputs into one dataset or where repetitive processing of the entire dataset is needed [3].

However, there is a trend of solving huge tasks by providing shared memory that is accessible to all nodes. The most popular is the nonuniform memory access (NUMA) architecture. The NUMA architecture allows each computer to share its memory while providing a distributed shared memory, allowing each computer to use the whole as a single pool of memory for solving one huge computational problem [4].

However, since message passing for each distributed memory is performed through a central processing unit, each computer using NUMA has an overhead for data exchange added to its computations and memory access is uneven and self-timed, leading to the poor execution of tasks that should be processed in real time [5].

Additionally, as Moore's Law has reached its limit, it has become widespread to solve complex computational problems by employing computational accelerators (such as a graphics processing unit (GPU)/field-programmable gate array (FPGA)/application specific integrated circuit (ASIC)), rather than only central processing units [6]. Accelerators greatly improve the computational power of a computer, enabling it to handle tasks that could only be processed by a distributed system before. When using GPU accelerators, it is possible to mount multiple accelerators on a single computer, which increases the computational performance.

Data that are processed by the accelerator are moved from the host memory to the accelerator, and data exchange between accelerators also goes through the host memory. It can cause bottlenecks in the system interconnection, such as the host PCIe interface. To minimize such bottlenecks, GPUDirect technology was developed to enable peer-to-peer

communication on PCIe switches between GPU accelerators. Data exchange between other interconnected computers occurs between the GPU and InfiniBand host adapter using a similar method [7,8].

A computational accelerator that is installed in each computer that comprises a distributed system uses network devices like InfiniBand to communicate between nodes. InfiniBand provides high performance and high bandwidth data exchange. However, a distributed system that incorporates InfiniBand for data exchange exhibits weak computational efficiency for each computer that is used to perform common tasks because InfiniBand also uses an MPI.

From a programming perspective, an overhead can occur when message passing rather than a consistent data processing approach is used in the shared memory. It makes the program handling of data complex. To overcome this problem, the unified memory management technique was introduced to map the accelerator memory and the host memory address area into a single address space to maintain consistency [9]. The unified memory management technique has contributed to simplified programming but since complex data exchanges remain and internal interconnections, such as PCIe, are crowded, it is not hardware-efficient when exchanging data [10].

Systems that use multiple computational accelerators can treat common data more efficiently if the computational accelerators on each node have an interface with each other. Accelerators can directly communicate with each other instead of uploading data to the host memory and propagating it over the network to other nodes.

Thus, a technique that leverages multiple GPUs within a single computer but uses a dedicated path between the GPUs was developed. It takes advantage of unified memory management and increases efficiency and maximizes performance within a computer, enabling the processing of increasingly correlated data [11,12].

However, this technique is not cost-efficient because all existing computer systems must be replaced. Furthermore, it is not suitable for users who will still want to use their own distributed systems [13].

In addition to computing using accelerators, memory-centric computing is emerging, which overcomes the limitations of Moore's Law. The traditional processor-centric architecture uses processors at the center of its calculation, focuses on all the capabilities of each computer, allows processors to handle all the memory data, and requires network access, such as Ethernet or InfiniBand, to access memory from other computers. However, in memory-centric computing, memory is at the center of the computing architecture, and all calculations are performed in the central memory. With these goals, each consortium competes against each other to create better products [14]. The leading vendor for this architecture is Hewlett Packard Enterprise. They are developing their protocol under the name "The Machine," and announced that the connection between nodes would be based on silicon photonics. They also develop related operating systems [15]. However, this development is not cost-effective because it should also replace all existing distributed systems, and existing systems cannot use these functions via a simple upgrade.

This study proposed solutions that can be applied to current systems without major changes. The architecture and implementation solutions that apply to a large-scale distributed system are presented, and experimental results that show the feasibility of these architectures and implementation solutions are provided.

3. Requirements for Distributed System and Related Previous Works

A distributed system that is composed of multiple computers can function as one system from a logical perspective.

The requirement for configuring a distributed system includes various communication paths between independent resources so that users treat them as a single system without suffering any degradation in the functionality.

Additionally, a distributed system should provide a consistent communication interface between users and systems, allowing users not to care about the locations for

distributed computing resources. In other words, an efficient distributed system ensures transparency for users [1]. We explore the requirements for a distributed system in terms of memory. If the communication mechanism with memory is improved and computing resources access it consistently, users will not suffer limitations when using a single program on multiple computers. The memory requirements to ensure the transparency of the system can be divided into seven categories:

1. Access: When accessing the memory, each computer should be able to use the same address.
2. Location: The mechanisms that each computer approaches should not vary depending on the type of memory.
3. Migration: Any computer should be able to access data in the memory when data movement or data processing occurs.
4. Relocation: All computers should be able to change the location of the data in the memory.
5. Replication: Every computer should have a copy of its data.
6. Concurrency: Data in the memory should be accessible from each computer with equal permission.
7. Persistence: It should be possible to consider the use of a backup mechanism or permanent memory for the memory that is accessed by each computer.

We considered how to make each computing resource accessible with minimal data movement, particularly focusing on the memory, which is a major independent resource. Each computer's I/O interface provides access to devices that use memory as their own, providing each computer with consistent and even access to data. In order to apply the memory characteristics to the distributed system, the shared memory system that is used in the distributed system is the most suitable one for this study. This study stands out especially in the studies related to semiconductor chips and software architectures. We refer to five representative studies.

The first study involved implementing a 16-core processor that enables both shared memory and message passing as a method of communication [16]. The proposed 16-core processor for embedded applications required hybrid inter-core communication, which supported message-passing communication using 2D Mesh networks-on-chip and shared memory communication using a memory core. The 16 cores smoothly communicated with each other through benchmarks, such as a 3780-point fast Fourier transform (FFT), H.264 decoder, low-density parity-check decoder, and long-term evaluation channel estimator.

The second study showed that a configurable single instruction multiple threads multi-core processor using a shared memory can improve performance with shared memory [17].

The third study involved a similar architecture to our memory-centric architecture, which attempted to speed up the deep learning model. In a simulation, the memory-centric system architecture doubled the speed of the device-centric deep learning architecture for training the deep neural network [18].

The fourth study explored software-implemented shared memory using InfiniBand, instead of the hardware-implemented shared memory that was used in our study. It used up to eight computation nodes and tested the system using shared memory. We used the same programming model in our study, except we used hardware shared memory [19].

The fifth was a study of data migration between dynamic random access memory (DRAM) and nonvolatile random access memory (NVRAM) for the in-memory processing environment [20]. It ensured efficient data migration by using predictions of data usage. The shared memory used in this study was based on the expectation that our memory's efficient use was possible.

In our study, we extended the first and second studies' semiconductor chip design structure to the external interface, formed the system used by the third study, and experimented to prove its validity, based on the contents of the fourth study. We expect our approach can also be used in a big data analysis by considering the fifth study's results.

4. Membox Architecture and Operation Mechanism

We began our study with the idea that each node can access the shared memory called a MemBox and implement it to meet the distributed system requirements that are described in Section 3. In other studies, shared memory was either implemented inside a compute node or inside a chip for communication between cores.

Otherwise, it is a software concept that existed in the form of a parameter server (e.g., Google Tensorflow's parameter server mechanism), which is an independent memory server. We proposed a shared memory system architecture that is to be used as a parameter server by replacing the logical shared memory with physical shared memory.

For comparison with our work, in Figure 1, we show a distributed system with each node connected using an InfiniBand or Ethernet. Each node is connected to an InfiniBand or Ethernet switch using an InfiniBand or Ethernet adapter at each node, and the commonly used data can be exchanged by the switch.

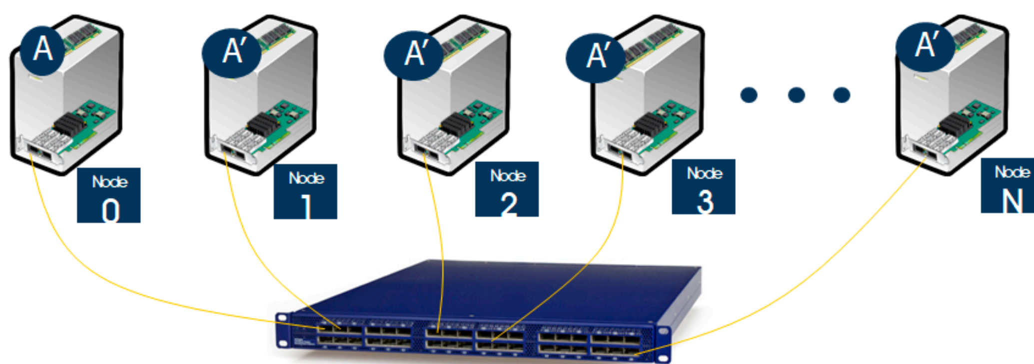

Figure 1. Network-based distributed system.

For commonly used data named A, the operating mechanism in Figure 1 is as follows:

1. Decide A at node 0;
2. Copy A to nodes 1~N (A′);
3. Allow access to A′ at nodes 1~N;
4. Repeat steps 2–3—O(N^2).

However, our memory-based distributed system has external memory (MemBox) with multiple ports, as shown in Figure 2.

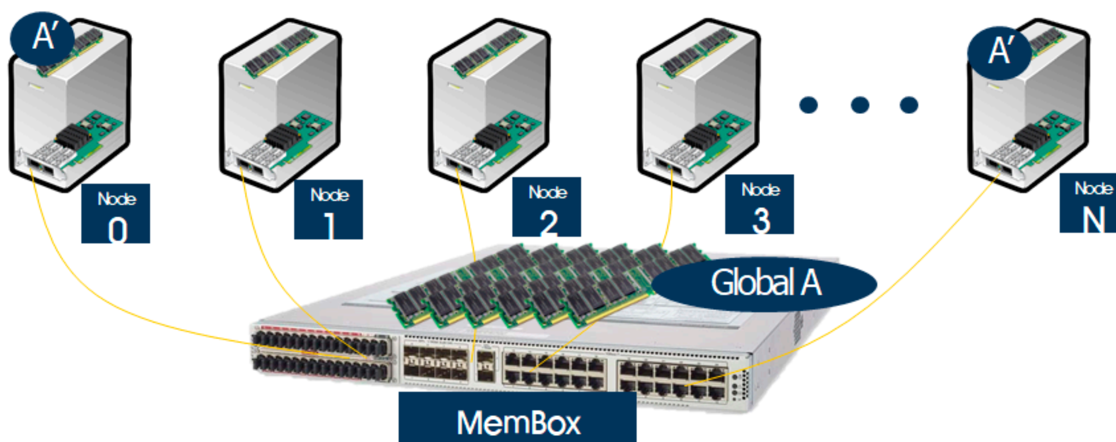

Figure 2. MemBox-based distributed system.

Each compute node can access the MemBox using an external memory access adapter (MemBox adapter) installed at each node.

The memory of a MemBox can have a separate address space. It can be accessed using direct memory access (DMA). Moreover, it can be mapped to an address space on each computer to be accessed using the same method as the local memory access mechanism.

1. Copy A to node 0 (A');
2. Allow access to A' at node 0;
3. Copy A' to the MemBox.

If we do not use DMA to map to the address area of each node, we do not need the copying process from step 1 above, and we can access global A directly. Because a MemBox stores highly correlated data and each node reads and writes data with a uniform mechanism, there is no difference in data access between the nodes, which can contribute to the simplification of the programming model.

The simplification of the programming model ensures consistency in programming. Since a MemBox has no server space limitations, it is easy to expand the memory capacity. MemBox adapters can also be made in the form of add-in cards that provide easy adoption to existing systems. By using our MemBox and applying a MemBox adapter to existing compute nodes, it is easy to configure distributed systems.

To configure the Membox-based distributed system in Figure 2, an external Membox system is required, and a Membox adapter must be installed in each existing node such that it is optically connected to the external Membox system. Figure 3 shows the configuration of a 1:1 connection between a Membox and a Membox adapter for detailing internal architecture. It shows the main components of Membox and Membox adapter required for its operation. Generally, interfaces between the processor and memory require 100 or more parallel signals. Therefore, it cannot be extended externally and memory sharing mainly uses a method for sharing indirectly through a network and a processor rather than directly accessing the memory from other processors. Accordingly, when the traffic to the memory increases, network overhead occurs and overall memory access performance suffers degradation.

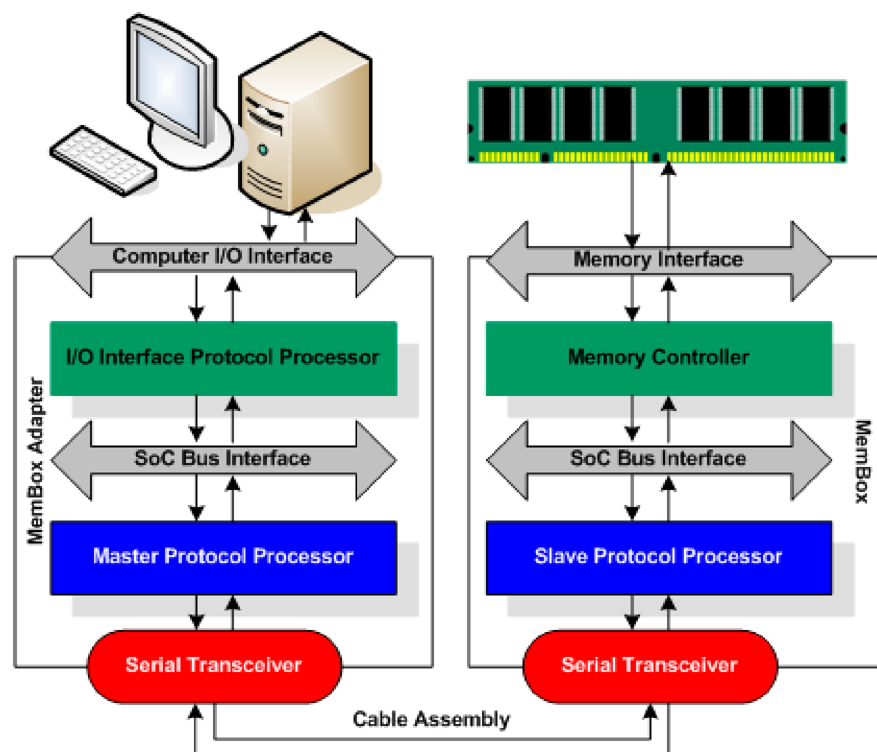

Figure 3. MemBox system architecture.

Our work provides shared memory that each node can connect to directly via an individual path. The interface between the existing processor and memory is converted into a packet-based protocol, each packet can be transmitted and received through a high-performance serial interface, and memory is shared by allowing each node to access the memory with no difference.

5. MemBox System Implementation

The most ideal setup for the shared memory system involves the processor directly providing a high-performance serial interface, and the memory also providing multiple high-performance serial ports, namely, as many as the number of nodes. However, since the existing motherboard does not support an external high-performance serial interface, it is necessary to find a feasible alternative, and the Membox adapter and Membox were configured accordingly.

We designed an IP that converts the parallel interface between the processor and the memory into a high-performance serial interface and configured it to have better performance regarding bandwidth and access time than the packet-processing method that is used in legacy network protocols. Additionally, we suggest the most optimized design within the constraints of implementation, such as the use of four 10G channels, and we also consider future bandwidth expansion.

5.1. MemBox Adapter

MemBox adapters consist of the I/O interface protocol processors that are responsible for the conversion of the protocols between the computer I/O interface and the system-on-chip (SoC) bus interface, the master protocol processor (MPP) that converts the SoC bus interface signals into the packet format for serial transceiver, and serial transceivers. We implemented each component to satisfy various requirements.

5.1.1. Computer I/O Interface Protocol Processor—PCIe

Since MemBox adapters exchange information to access the memory from computers, they must have a computer I/O interface. Therefore, an appropriate I/O interface should be used that accounts for the performance and usage.

Computers that are used in distributed systems have internal interconnections for peripherals that are added to the DRAM interface between the CPU and the memory, of which PCIe is the most widely used. This study also adopted PCIe as the computer I/O interface to apply the MemBox to common distributed systems.

PCIe is a high-speed serial link upgrade from the existing parallel PCI bus that has layer characteristics and the dual simplex signaling and is used inside the motherboard for connection with legacy devices, such as Ethernet or USB adapters that are embedded in the motherboard. Furthermore, it provides slots to separate interfaces for expanding computer functions, which makes it easy to implement a MemBox adapter.

Our MemBox adapter was implemented in the form of add-in cards for connecting with the host computer via a PCI Express interface. Our implementation adopted "Hard IP for PCI Express using Avalon-MM with DMA" for Intel Arria 10 GX FPGA. Specifically, the MemBox adapter uses DMA to directly access the MemBox without a host computer to send and receive large amounts of data. [21].

As proposed in this study, the IP is configured to PCIe Gen3, x8. It has separate read and write DMA interfaces and the ability to generate/consume PCIe transactions. When PCI Express operates as Gen3, the Avalon-MM interface operates at 250 MHz, and the implementation of our work requires a bus width of 512 bit to meet the bandwidth requirement of a 40 Gbps optical connection.

5.1.2. Serial Transceiver

A transceiver that is capable of accommodating the 10GBase-R specifications was used as a serial transceiver in the MemBox adapter. We used a transceiver IP that was supported

by the FPGA device. We could implement 10GBase-R with Arria 10 transceiver native physical layer (PHY) in the Arria 10 FPGA device, and we implemented 40 Gbps using four channels [22]. The 10GBase-R transceiver uses 72 bits as input and output for a 64-bit data signal and an 8-bit control signal, and the clock restored from a serial signal is 156.25 MHz. The inputs and outputs of the data signal and control signal are based on this clock.

5.1.3. MPP

The MemBox adapter sends and receives data at an aggregated 40 Gbps using four lanes. By differentiating the characteristics for each lane, the configuration is made easy when the data width is extended. One lane is the reference lane that generates the header and data, transmits memory write packets (MWP), and receives and processes memory read response packets (MRRP). The other three lanes generate and transmit MWP matching the header of the reference lane and receive and process MRRP.

The MPP takes charge of 128 bits for each lane. Figure 4 shows the internal structure of the MPP. It is the module responsible for the conversion between the Avalon-MM interface that is incorporated into the SoC interface and the packet format for optical transmission in the MemBox system that creates and consumes packets, formats and analyzes packets, controls the data flow, and handles the data integrity.

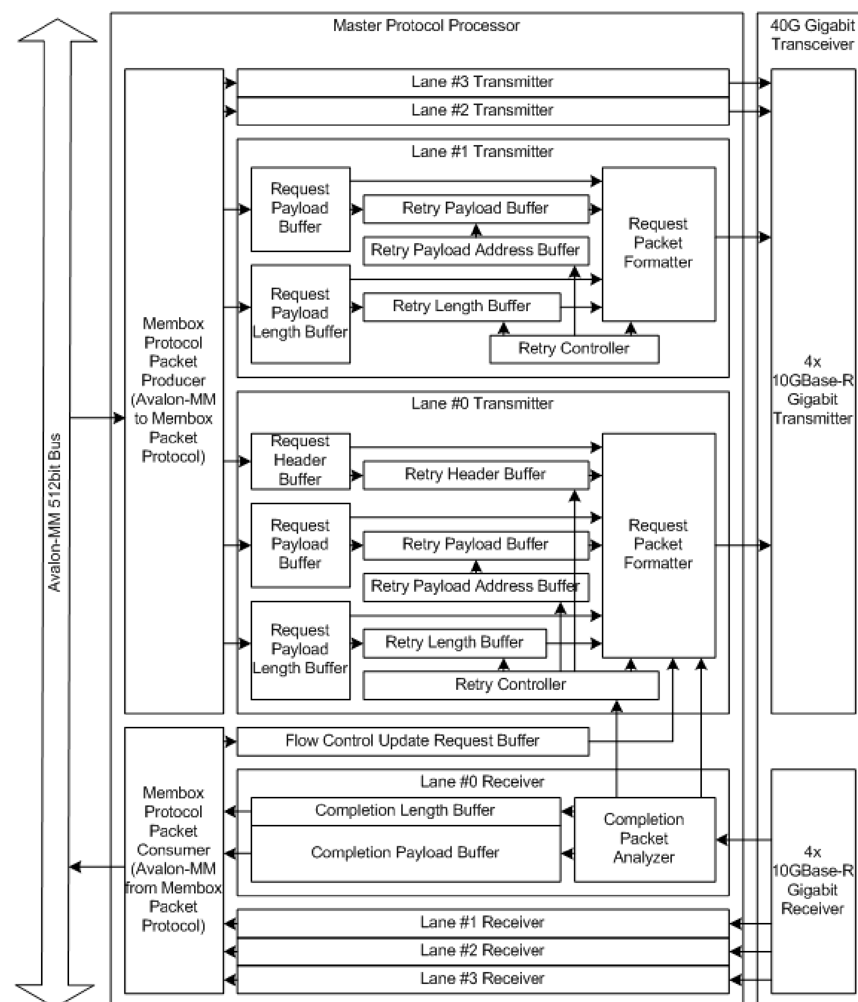

Figure 4. Master protocol processor.

MemBox's protocol packet producer (PPP) turns memory write and read transactions that are received from the Avalon-MM interface into header and data packets and stores them in the request header buffer (RHB) and request payload buffers (RPB) of lane zero

and the RPB of lanes one to three. Furthermore, the length of the data stored in the RPB is stored in the request payload length buffer (RPLB).

The request packet formatter (RPF) in each lane reads the buffer contents and delivers them to a 10GBase-R transceiver in the form of 64 bits per clock. When reading a packet from a buffer, it reads the contents of the RHB, RPB, and RPLB, and is also stored in the retry header buffer (RTHB), retry payload buffer (RTPB), and retry length buffer (RTLB). The address to read data from the RTPB is stored in the retry payload address buffer (RTAB) and later used for retransmission when needed.

The MRRP that are received from the 10GBase-R transceiver are stored in the completion payload buffer (CPB) and completion length buffer (CLB) after packet inspection in the completion packet analyzer. Once the data for each lane are available, the MemBox protocol packet consumer reads and converts the payload data to an Avalon-MM read response and forwards it to the Avalon-MM 512 bit bus. The RPF and completion packet analyzer (CPA) also generates and consumes flow control packets.

CPA informs the RPF of the integrity information for the received packets and the request formatter generates Ack or Nak from this information and delivers it to the other side's receiver. When the CPA receives the Ack or Nak, it notifies the retry controller. Upon receiving the Ack, the retry controller frees the space for the retry buffer by moving the pointer to the RTHB, RTPB, RTLB, and RTAB to the next normally received packet. When the Nak is received, it frees up space in the buffers for correctly received packets and retransmits the packets that are not correctly received.

The packet retransmission occurs not only when Nak is received but also when Ack or Nak for the transmitted packet is not received for a certain period. When retransmitting a packet, data are stored in the RTHB, RTPB, RTLB, and RTAB so that retransmission continues until it receives Ack or Nak for packets that are correctly received and then removed from the buffer.

The clock frequency used in signals on the Avalon-MM side of the PCI Express protocol processor is 250 MHz, the bus width is 512 bit, and the clock frequency used on the 10GBase-R 4 channels is 156.25 MHz. To compensate for this clock difference, each buffer uses separate read and write clocks operating at 250 MHz and 156.5 MHz, respectively. The information to send flow control update packet also uses a buffer to compensate for this difference.

5.2. MemBox

The MemBox consists of the same serial transceiver as in the MemBox adapter, the slave protocol processor (SPP) for signal conversion and packet processing between the serial transceiver and the SoC bus interface; the memory controller, which is responsible for the signal conversion between the SoC bus and the memory interfaces; and the memory that stores the data. It also differentiates the lane characteristics in the MemBox adapter. It receives packet signals into the serial transceiver through a cable assembly that is connected to its adapter and converts them in the SoC master interface, which is used in the FPGA/ASIC. It is done through a protocol conversion process in the SPP. The SoC bus interface has a memory controller with an SoC slave interface, enabling access to the desired memory. The MemBox can connect with several MemBox adapters; therefore, there are multiple serial transceivers and multiple SPPs. A bus arbitration logic is added and connects memory controllers from multiple SPPs. The architecture that was proposed in this study selects and uses the memory in the MemBox according to the type of implementation. The performance of the MemBox varies depending on the memory type. An advantage of our architecture is that the MemBox allows users to choose various types of memory without the restriction of the type of memory bus required by the host computer.

5.2.1. Serial Transceiver

The MemBox uses the same serial transceiver that can accommodate 10GBase-R specifications as the MemBox adapter.

5.2.2. SPP

The SPP is a module that is responsible for the conversion between the packets that are generated by the MPP and the Avalon-MM interface that is adopted as the SoC interface. It is also responsible for the consumption of request packets and the generation of response packets, the formatting and processing of packets, and the support of flow control and data integrity. Figure 5 shows the internal structure of the SPP.

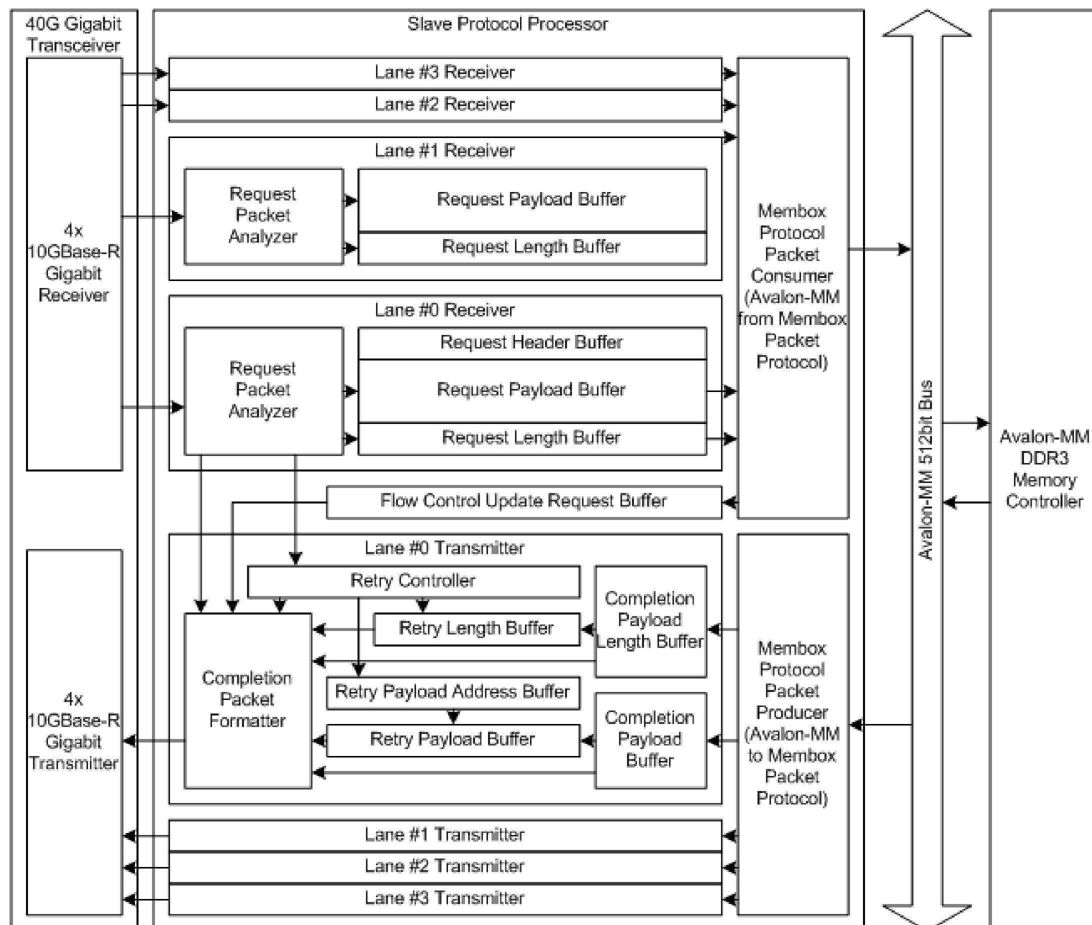

Figure 5. Slave protocol processor.

The SPP used in the MemBox allows data to be transmitted and received to/from the MemBox adapter at 40 Gbps per interface for four lanes.

One lane receives both the header and data upon receipt of the MWP and the other three lanes receive only data packets matching the header of the reference lane. All lanes transmit only data packets. The SPP transmits and receives packet data through four lanes and can handle 128 bits for each lane, summing up to 512 bits of data width. The request packet that is received from the 10GBase-R receiver is stored in the RHB, RPB, and the RPLB, after inspecting the packet in the request packet analyzer.

Once the data for each lane is available, the MemBox protocol packet consumer reads the payload data and generates an Avalon-MM memory write/read signal, which is passed to the Avalon-MM 512 bit bus, and finally to the DDR3 controller.

The Avalon-MM 512 bit bus has multiple slave protocol processors, which connect with the DDR3 memory controller. If multiple slave protocol processors compete for a single DDR3 memory controller, an arbitration logic is required between them, which is automatically created using the Qsys Tool of Intel Quartus Prime software. The clock signals on the 10GBase-R channels are operating at 156.25 MHz, and the clock signals on the Avalon-MM side of the DDR3 controller are operating at 200 MHz, resulting in a

clock difference. Therefore, buffers with separate read and write clocks compensate for clock differences.

The flow control, which updates the information delivery path, also has a separately clocked buffer; therefore, flow control information generated by the MemBox protocol packet user operating at 200 MHz does not suffer clock skew with the completion packet formatter (CPF) operating at 156.25 MHz.

MemBox PPP converts memory read response transactions that are received with the Avalon-MM interface into data packets and stores them in CPB of lanes 0 to 3. Furthermore, the data length stored in the CPB is stored in the CLB. The CPF in each lane reads the buffers, converts them to 64 bits per clock, and passes them to 10GBase-R transmitters. When reading a packet from the buffers, it reads the contents of the CPB and CPLB, which are also stored in the RPB and RPLB, respectively. The address to read data from the RPB is stored in the RPLB and later used for retransmission if the packet transmission was incorrect.

The request packet analyzer (RPA) and CPF also generate and consume flow control update packets.

The RPA informs the CPF of the integrity information for received packets and the CPF generates Ack and Nak by this information. When Ack or Nak is received from RPA, the retry controller detects it, and when an Ack or Nak notification is received from the retry controller, RTLB and RTAB are moved to the next packet of normally received packets. Moreover, packets not received are retransmitted.

The packet retransmission occurs not only when Nak is received, but also when Ack or Nak for the transmitted packet has not been received for a certain period. Retransmitting a packet also stores data back in the RPB, RPLB, and the request payload address buffer so it can be sent for retransmission before the Ack or Nak packets received are removed from the buffer.

5.2.3. Memory Controller and DRAM

In our work, a DDR3 DRAM module was adopted to form a MemBox. This increases the capacity of the MemBox via simple module changes and extends the available memory capacity in proportion to the number of memory controllers in the MemBox. We used DDR3 DRAM memory with a 4 GB capacity and an 800 MHz clock speed. A MemBox converts external optical signals into internal SoC interfaces; therefore, memory controllers are also used to have SoC interfaces.

5.2.4. PCIe Interconnection

PCI Express IP enables a MemBox to act like a MemBox adapter. The MemBox implementation in this study used a PCIe Avalon-MM DMA handler provided by the Arria 10 FPGA with DMA, which has the same functionality as MemBox adapters. Similar to PCIe protocol processors that are used in a MemBox adapter, PCI Express IP was configured for PCIe Gen3 and x8. When PCI Express operated as Gen3, the Avalon-MM interface operated at 250 MHz, and this study required the bus width to be 512 bits. For the clock difference compensation with a 200 MHz Avalon-MM memory controller, the extra logic was automatically generated from Qsys system integration tools of Intel Quartus software.

6. Experiments

6.1. MemBox Access Time Measurement

To analyze the performance of MemBox memory, we compared the time it took to train deep learning problems with memories in various environments.

6.1.1. Test Environment

We installed MemBox and MemBox adapters on each compute node and connected them using an optical connection, as shown in Figure 6. The MemBox memory is accessible through the PCIe interface on the compute node with the MemBox installed. Furthermore,

it is accessible on the compute node with the MemBox adapter having an optical connection with the MemBox.

Figure 6. Access time measurement system.

For performance experiments, a distributed Tensorflow environment was established and the Modified National Institute of Standards and Technology (MNIST) dataset was tested. Their large dataset consists of handwritten numbers that are used for training image processing systems. Tensorflow is an open-source software library developed by Google for numerical computation and is widely used by many large companies. It provides an interface for expressing machine learning algorithms and has an application for executing these algorithms. Our Tensorflow framework using MemBox memory was rebuilt based on version 1.10 of Tensorflow and a Python script was written and executed to train the MNIST dataset using the Tensorflow core.

6.1.2. Results and Analysis

The MNIST Tensorflow test was performed using only the main memory, the MemBox memory that was accessed from the node where the MemBox was installed, and the MemBox memory that was accessed from the node where the MemBox Adapter was installed. This test was also performed using the node where the MemBox Adapter was installed and the node where the MemBox was installed. The results are shown in Table 1 and Figure 7 below.

Table 1. Memory access time measurement between memory types.

Memory Access Type	**Accuracy**			
	30%	**50%**	**70%**	**90%**
1 node, Main Memory	5.0 s	9.9 s	13.4 s	48.9 s
1 node, MemBox (Local)	4.3 s	8.5 s	14.8 s	58.1 s
1 node, MemBox (Remote)	4.4 s	7.9 s	14.2 s	55.4 s
2 node, MemBox	6.7 s	10.6 s	14.5 s	45.6 s

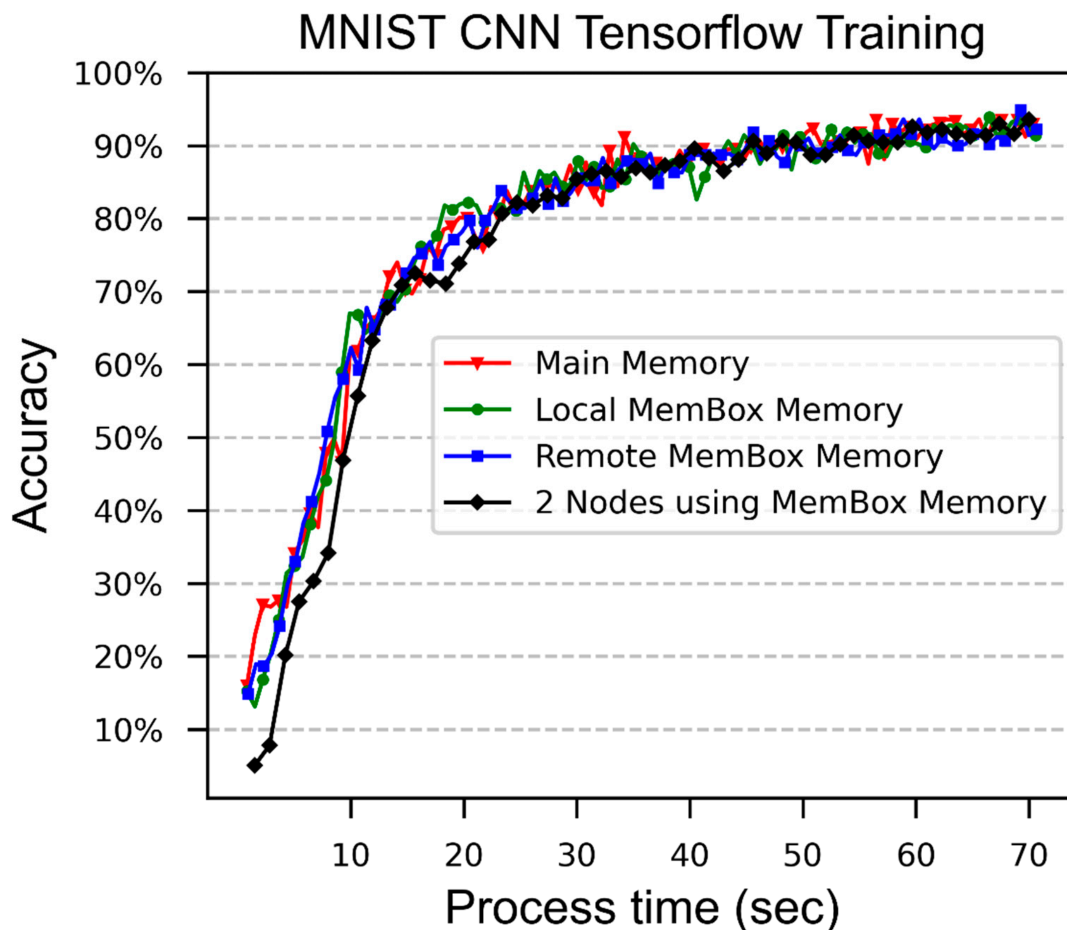

Figure 7. Access time comparison.

Although the main memory access time was much faster (~20 ns) than that of the MemBox's PCIe I/O memory (>1 µs), the time that was taken for the main memory and the MemBox memory to reach the accuracy of 30~70% fluctuated between both memories. However, the time that was required by the main memory to reach 90% or higher accuracy was significantly less than that of MemBox memory because the main memory could access the memory directly but the MemBox memory was accessed via a PCIe I/O and DMA mechanism.

Although the main memory was much faster than MemBox's PCIe I/O memory, when two nodes used the MemBox memory, it took less time to reach 90% accuracy and we expect that if more nodes are used, the overall training performance can be increased further with the MemBox system. The results of a MemBox performance test that was conducted with more nodes and complex applications are given in the next section.

6.2. MemBox Performance Experiment

An experiment was conducted to measure the performance gain that was obtained as the number of nodes using MemBox increased, and the results were compared with the case of using a parameter server.

6.2.1. Test Environment

We used five SuperMicro 4028GR-TRT2 servers, one of which had MemBox and the others had a MemBox adapter. The server with the MemBox adapter was connected to the MemBox via a 40G Optical Cable. The SuperMicro server had a two-socket Intel Xeon CPU (E5-2690 v4, 14 cores, 2.3 GHz), 128 GB DDR4-2400 MHz/ECC memory, and four NVIDIA GPUs. The types of GPUs consisted of NVIDIA Pascal architecture GPUs, such as

Titan X Pascal (3584 cores/12 GB memory) and Titan Xp (3840 cores and 12 GB memory). The HP server had a two-socket Intel Xeon CPU (E5-2609 v2, 4 cores, 2.5 GHz) and 256 GB DDR3-1866 MHz memory.

Compared with the case using a parameter server, each server had an InfiniBand FDR HCA (56 Gbps) and was connected to the InfiniBand switch.

The OS was Ubuntu 14.04-LTS and the kernel version was 3.13.0-123-generic. We used CUDA 8.0 (including cudnn 8.0). We measured the computation time of training actual deep neural network models in a single GPU and the parameter's data size. For this purpose, our Tensorflow 1.10 for the MemBox system was used to measure the computation time during forward and backward training.

6.2.2. Result and Analysis

The training time and accuracy were measured when performing the Resnet-50 Tensorflow tests, respectively, using one, two, and four compute nodes in the parameter server and MemBox system environment.

Figure 8 is a comparison of the results of Resnet 50 CNN Tensorflow training using one, two, and four compute nodes using MemBox system and parameter server when reaching 90% accuracy. In all cases, the MemBox system on Resnet-50 took less training time than the parameter server. The results showed a training time decrease of 8% in the case of using one node, 7% for two nodes, and 17% for four nodes, and as the number of nodes increased, the processing time decreased inversely. Therefore, we can expect that when eight nodes are used, the processing time will be about 49 s in the MemBox system and 65 s in the parameter server, thus showing a 25% time decrease.

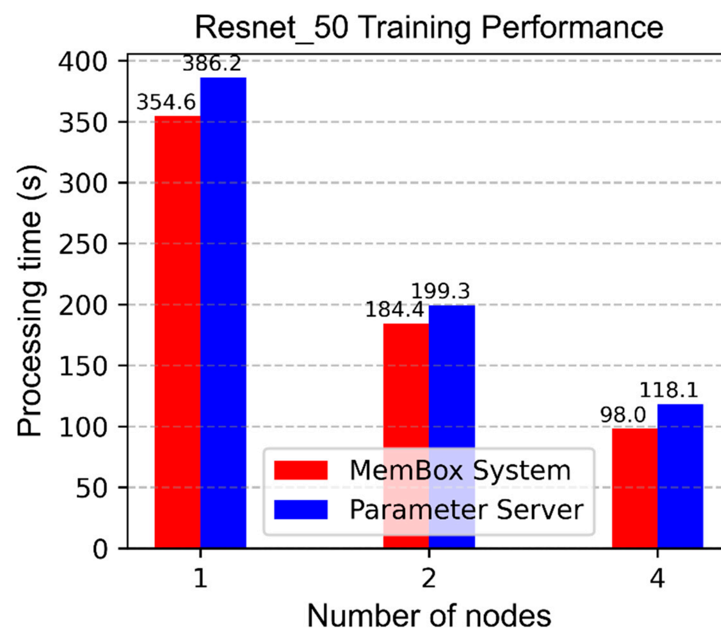

Figure 8. Resnet_50 training result comparison between the MemBox and the parameter server.

Because a memory access bottleneck does not occur in the network port except in the memory itself, we can expect a greater decrease in the training time when using the MemBox system in more nodes, but the parameter server suffers more of a bottleneck in a higher-node environment.

6.3. Membox Repeat Access Test Using Repeater

The Membox development board has four ports; therefore, there are four nodes that can connect to the Membox. To expand the number of nodes that can be connected, a Membox repeater was devised. As with the Membox adapter, it transmits and receives

optical signals through the PCIe interface using the master protocol processor, and has the same slave protocol processor that the Membox has and relays it back to the master protocol processor. Therefore, it can connect other Membox adapters to the other three ports in addition to one port that is connected to the Membox, and also serves as a Membox adapter by itself. Therefore, when using a repeater, up to 16 Membox adapters can be connected to the Membox. Figure 9 shows the configuration in which eight nodes are connected to a Membox by using a Membox with four ports and one Membox repeater connected with three MemBox adapters, which we used in our experiment. We could not connect more than eight nodes due to the lack of FPGA boards.

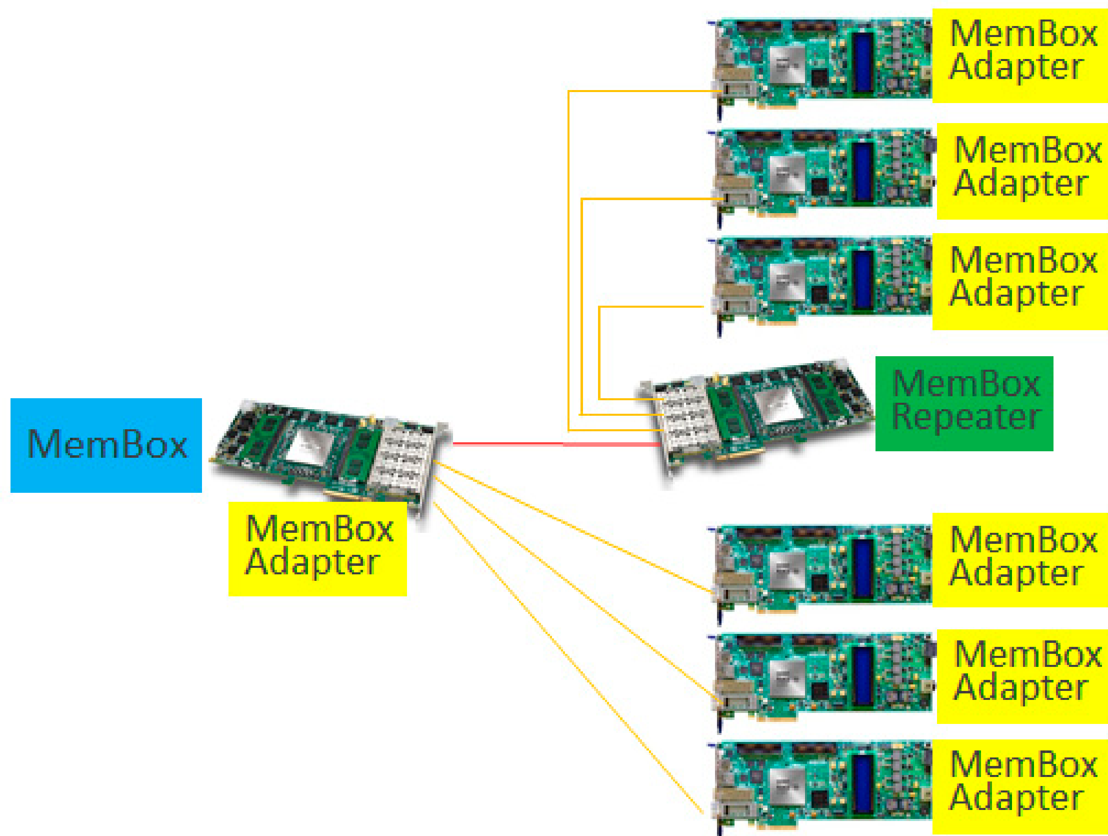

Figure 9. MemBox configuration using a MemBox repeater.

6.3.1. Test Environment

We used nine SuperMicro 4028GR-TRT2 servers, one of which had a MemBox, two of which had a MemBox repeater, and the others had a MemBox adapter. The server with the MemBox adapter and MemBox repeater was connected to a MemBox repeater and MemBox via a 40G optical cable. The SuperMicro server had a two-socket Intel Xeon CPU (E5-2690 v4, 14 cores, 2.3 GHz), 128 GB DDR4-2400 MHz/ECC memory, and four NVIDIA GPUs. The types of GPUs consisted of NVIDIA Pascal architecture GPUs, such as Titan X Pascal (3584 cores/12 GB memory) and Titan Xp (3840 cores and 12 GB memory). The HP server (parameter server) had a two-socket Intel Xeon CPU (E5-2609 v2, 4 cores, 2.5 GHz) and 256 GB DDR3–1866 MHz memory.

Compared with the case using a parameter server, each server had an InfiniBand FDR HCA (56 Gbps) and was connected to the InfiniBand switch.

The OS was Ubuntu 14.04-LTS and the kernel version was 3.13.0-123-generic. We used CUDA 8.0 (including cudnn 8.0). We measured the computation time of training the actual deep neural network models in a single GPU and the parameter's data size. For this

purpose, our Tensorflow 1.10 for MemBox system was used to measure the computation time during forward and backward training.

6.3.2. Result and Analysis

The training time and accuracy were measured when performing the MNIST Tensorflow tests, respectively, using one, two, four, and eight compute nodes in the parameter server and memory-side system environment.

Figure 10 is a comparison of the results of 100 epoch MNIST CNN Tensorflow training average time at each node using one, two, four, and eight compute nodes using a memory-side system and a parameter server. The total number of epochs was 100 for one node, 200 for two nodes, 400 for four nodes, and 800 for eight nodes. In all cases, using the memory-side system on MNIST datasets took less training time than using a parameter server. The results showed a training time decrease of 44% when using one node, 35% for two nodes, 38% for four nodes, and 22% for eight nodes.

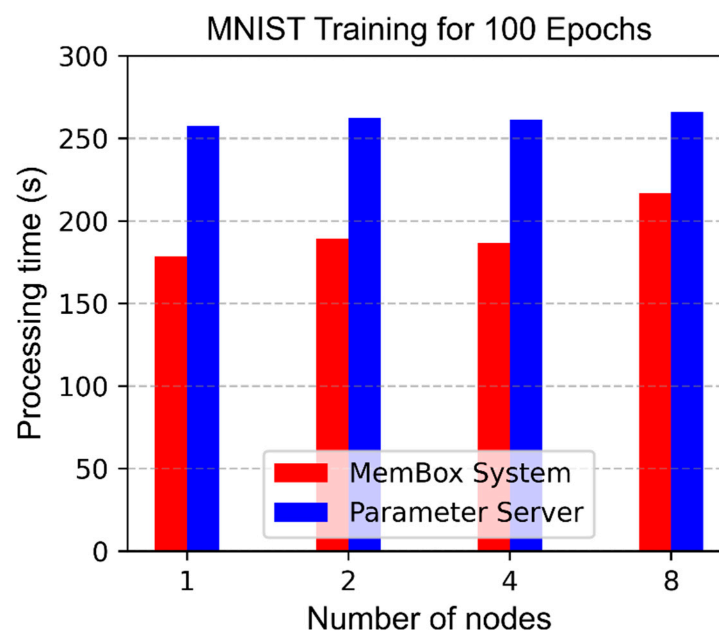

Figure 10. MNIST 100 epoch Training result comparison.

When using the parameter server, the overhead in the path that each node uses to access the parameter server was nearly the same; therefore, even if the number of nodes increased, the time to reach 100 epochs was nearly the same. The test using up to four nodes included three nodes that were directly connected to the Membox and the Membox node itself. However, since eight nodes used Membox repeaters, performance degradation occurred. We think this was due to the serially connected store and forward buffer characteristics of the master protocol processor and the slave protocol processor in the repeater; therefore, it is desirable to change it to a cut-through buffer.

7. Conclusions and Future Work

A new type of shared memory, called a MemBox, was proposed to implement a distributed system, called a MemBox system, that is suitable for modern computing problems in big data and deep learning. An architecture was presented to construct the MemBox system and was implemented using the FPGA board; then, the implementation results were tested. The memory access time comparison using two compute nodes showed that the distributed system handled one task. When using the MemBox system, the distributed system was advantageous in reaching target accuracy. Multinode testing with servers was also performed, which showed that using a MemBox system took a shorter

time than using a traditional parameter server. Moreover, the system increased linearly in performance as the number of compute nodes increased. The experiments demonstrated that the use of a MemBox memory, compared with that of the main memory, could operate without a difference in reaching the desired accuracy and operate better than a legacy parameter server-based system. Additionally, we tested an eight-node environment and showed its feasibility; however, when the number of port connections using a repeater was extended, the performance was degraded in the experiment, and considerations were presented when designing the repeater.

Future studies should seek to remove bottlenecks when accessing memory via multiple MemBox adapters and plan design changes for the repeater for the MemBox port expansion. Our future research will reflect removing these bottlenecks based on memory access pattern analysis and increased ports on the MemBox without performance degradation. It also includes how to add multiple MemBoxes, such as using a switch structure.

Author Contributions: Conceptualization, Y.C. and C.-H.L.; methodology, Y.C. and E.L.; software, E.L., experiments and data analysis, Y.C. and J.S.; writing—original draft preparation, Y.C.; review, editing, and providing some valuable suggestions and administration; C.-H.L. All authors contributed to this work. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute for Information and Communications Technology Planning and Evaluation (IITP), which is funded by the Korean Government (MSIT) through the Researches on Next Generation Memory-Centric Computing System Architecture under Grant 2018-0-00503.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Tanenbaum, A.S.; van Steen, M. *Distributed Systems: Principles and Paradigms*; Pearson Prentice Hall: Upper Saddle River, NJ, USA, 2002; pp. 20–31.
2. Hadoop, A. MapReduce Tutorial, The Apache Software Foundation. Available online: http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html (accessed on 26 May 2021).
3. Weets, J.F.; Kakhani, M.K.; Kumar, A. Limitations and Challenges of HDFS and MapReduce. In Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), Greater Noida, India, 8–10 October 2015; pp. 545–549.
4. Lameter, C. NUMA (Non-Uniform Memory Access): An Overview. *Queue* **2013**, *11*, 40–51. [CrossRef]
5. Guo, X.; Han, H. A good data allocation strategy on non-uniform memory access architecture. In Proceedings of the 2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS), Wuhan, China, 24–26 May 2017; pp. 527–530.
6. Nurvitadhi, E.; Sheffield, D.; Sim, J.; Mishra, A.; Venkatesh, G.; Marr, D. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT), Xi'an, China, 7–9 December 2016; pp. 77–84.
7. NVIDIA Corporation. NVIDIA GPUDirectTM Technology. Available online: http://developer.download.nvidia.com/devzone/devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf (accessed on 26 May 2021).
8. Shainer, G.; Ayoub, A.; Lui, P.; Liu, T.; Kagan, M.; Trott, C.R.; Scantlen, G.; Crozier, P.S. The development of Mellanox/NVIDIA GPUDirect over InfiniBand—A new model for GPU to GPU communications. *Comput. Sci.-Res. Dev.* **2011**, *26*, 267–273. [CrossRef]
9. Knap, M.; Czarnul, P. Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus. *J. Supercomput.* **2019**, *75*, 7625–7645. [CrossRef]
10. Banerjee, D.S.; Hamidouche, K.; Panda, D.K. Designing High Performance Communication Runtime for GPU Managed Memory: Early Experiences. In Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, Barcelona, Spain, 12–16 March 2016; pp. 82–91.
11. Ratnaparkhi, A.A.; Pilli, E.; Joshi, R.C. Survey of scaling platforms for deep neural networks. In Proceedings of the 2016 International Conference on Emerging Trends in Communication Technologies (ETCT), Dehradun, India, 18–19 November 2016; pp. 1–6.
12. Li, A.; Song, S.L.; Chen, J.; Li, J.; Liu, X.; Tallent, N.R.; Barker, K.J. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *31*, 94–110. [CrossRef]
13. Mojumder, S.A.; Louis, M.S.; Sun, Y.; Ziabari, A.K.; Abellán, J.L.; Kim, J.; Kaeli, D.; Joshi, A. Profiling dnn workloads on a volta-based dgx-1 system. In Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC), Raleigh, NC, USA, 30 September–2 October 2018; pp. 122–133.

14. Volos, H.; Keeton, K.; Zhang, Y.; Chabbi, M.; Lee, S.K.; Lillibridge, M.; Patel, Y.; Zhang, W. Memory-Oriented Distributed Computing at Rack Scale. In Proceedings of the ACM Symposium on Cloud Computing, SoCC'18, Carlsbad, CA, USA, 11–13 October 2018; p. 529.
15. Courtland, R. Can HPE's "The Machine" deliver? *IEEE Spectrum* **2015**, *53*, 34–35. [CrossRef]
16. Yu, Z.; Xiao, R.; You, K.; Quan, H.; Ou, P.; Yu, Z.; He, M.; Zhang, J.; Ying, Y.; Yang, H.; et al. A 16-core processor with shared-memory and message-passing communications. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2013**, *61*, 1081–1094. [CrossRef]
17. Kim, H.Y.; Kim, Y.J.; Oh, J.H.; Kim, L.S. A reconfigurable SIMT processor for mobile ray tracing with contention reduction in shared memory. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2012**, *60*, 938–950. [CrossRef]
18. Kwon, Y.; Rhu, M. A case for memory-centric HPC system architecture for training deep neural networks. *IEEE Computer Archit. Lett.* **2018**, *17*, 134–138. [CrossRef]
19. Ahn, S.; Kim, J.; Lim, E.; Kang, S. Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing. *IEEE Access* **2018**, *6*, 26493–26504. [CrossRef]
20. Mai, H.T.; Park, K.H.; Lee, H.S.; Kim, C.S.; Lee, M.; Hur, S.J. Dynamic Data Migration in Hybrid Main Memories for In-Memory Big Data Storage. *ETRI J.* **2014**, *36*, 988–998. [CrossRef]
21. Intel Corporation. Intel Arria 10 Avalon-MM DMA Interface for PCIe Solutions User Guide. Available online: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/archives/ug-a10-pcie-avmm-dma-16.1.1.pdf (accessed on 1 November 2020).
22. Intel Corporation. Intel Arria 10 Transceiver PHY User Guide. Available online: https://www.intel.com/content/www/us/en/programmable/documentation/nik1398707230472.html (accessed on 23 June 2020).