Reducing Data Copies between GPUs and NICs

Nguyen Viet Anh*, Yusuke Fujii†, Yuhki Iida†, Takuya Azumi‡, Nobuhiko Nishio* and Shinpei Kato§
*College of Information Science and Engineering, Ritsumeikan University
†Graduate School of Information Science and Engineering, Ritsumeikan University
‡Graduate School of Engineering Science, Osaka University
§School of Information Science, Nagoya University

***Abstract*—Cyber-physical systems (CPS) must perform complex algorithms at very high speed to monitor and control complex real-world phenomena. GPU, with a large number of cores and extremely high parallel processing, promises better computation if the data parallelism often found in real-world scenarios of CPS could be exploited. Nevertheless, its performance is limited by the latency incurred when data are transferred between GPU memory and I/O devices. This paper describes a method, based on zero-copy processing, for data transmission between GPUs and NICs. The arrangement enables NICs to directly transfer data to and from GPU. Experimental results show effective data throughput without packet loss.**

***Keywords*—GPU; GPGPU; Data Transfer; Low Latency;**

I. Introduction

Cyber-physical systems (CPS) are designed the manipulation and control of entities in the real world. A CPS has three components: computation, communication, and control. Computation includes many computing machines in which many complex control algorithms are processed. Communication is used for data transmission within the CPS. For example, data in a CPS may be transferred through 3G, Wi-Fi or wired networks. Control consists of I/O devices such as sensors or motor systems. Input devices collect data and information from the environment around CPS and through the communication component, send them to the computation one. In the computation component, computing machines perform necessary processing, produce control signals, and send them back to the output devices through the communication component. Based on the computations results, the control component changes its action to match the scenarios.

A typical example of a CPS is an autonomous vehicle. When an autonomous vehicle is moving on the road, processors inside the vehicle continuously receive a large amount of information from around the vehicle, which is sent from radars and GPU systems. The results of computation are used to change the speed and direction of the vehicle to avoid accidents and ensure that the vehicle arrives at the correct destination. In such systems, the real-time constrains are very important and reducing latency is critical.

The advanced technologies of general-purpose computing on graphic processing unit (GPGPU) and the integration of many-core processors into a CPS helps augment the performance of computation components in those systems. Many processing cores integrated inside a single chip gives massive computing power to GPU. By leveraging the parallelism often found in real-world problems, a GPU can substantially reduce the execution time for a CPS program. Given that GPU's are increasingly deployed in CPS applications [1], [2], for CPS using GPUs to perform specific tasks, data must be put into GPU memory. In most current systems, data transferred from input devices to GPU memory must go through host memory Similarly, when computation results are sent back from GPU memory to output devices, data transmission must occur through host memory. However, transmitting data between memory areas can reduce system performance. Therefore, we have been working to improve the efficiency of data transfers [3].

First, such data transmission requires multiple data copying, which results in additional latencies for the whole system; systems might need a few microseconds to process data, but hundreds of microseconds could be spent on data transmission. Second, extra buffers must be allocated in host memory to temporarily store data that is going through host memory; which results in the reduction of available memory. Finally, data throughput is also badly affected by data transmission between multiple memory areas, because the CPU has to wait for buffers to be filled with data before it can copy the data to different memory areas. The above three reasons necessitate reducing the number of data copying operations between multiple memory areas.

This study focuses on the specific case of data transmission between I/O devices and GPU memory: the data transmission from a Network Adapter (also known as Network Interface Controller, or NIC) to GPU memory. NIC plays a vital role in the current network model; without NIC, systems cannot exchange data with other machines. Network-based applications, such as web browsers, e-mail, and cloud storage, cannot work without NIC. However, NIC is also a type of I/O device; hence, the multiple data copying problem occurs when data are transferred between NIC itself and GPU memory. By applying a method called Zero-copy I/O [4], the number of data copying operations is reduced and the system performance is improved. We developed programs for quantitative evaluation and used these to compare the performance of our methods with that of a traditional method of transmission.

Organization: The rest of this paper is organized as follows. Section II discusses related work. Section III describes the challenges in data transmission between NIC and GPU memory in current data transmission schemes, as well as our approach for the above problem. Section IV presents the details of the design and implementation of our method in an existing system. The evaluation method and experiments results are presented in Section V. Finally, conclusion and future work are given in Section VI.

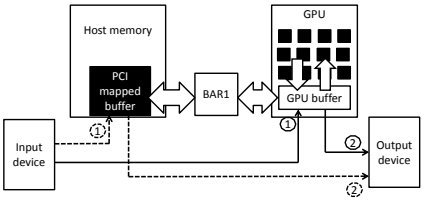

Fig. 1. Implementation of Zero-copy I/O on HBT-EP Tokamak system

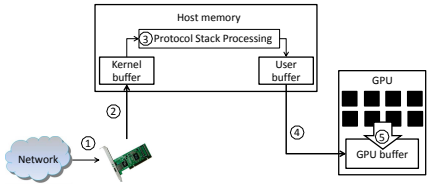

Fig. 2. Typical data transmission model between NIC and GPU memory

II. Related Work

A. GPUDirect

To solve the problem of multiple data copying between I/O devices and GPU memory, NVIDIA proposed GPUDirect [5]. GPUDirect comprises two main steps, as follows:

1) A user program allocates several buffers on the GPU memory and information on those buffers is passed to the I/O device drivers.
2) The I/O device driver converts the received information into bus addresses of the GPU buffers and passes those addresses to a DMA engine. The DMA engine copies data directly between GPU buffers and the I/O devices, bypassing host memory.

To simplify the implementation of GPUDirect, NVIDIA provided a library called nv-p2 to third party manufactures. Mellanox Company, in their project GPUDirect [6], used this library to solve the multiple data copying problem between InfiniBand [7]—a special NIC that is usually used for connection between processors and high-performance I/O such as storage devices—and GPU memory. However, InfiniBand and Ethernet NIC are very different. For example, to guarantee reliable data transmission, Ethernet NIC relies on the TCP/IP protocol, which is implemented in software layer, whereas InfiniBand uses hardware-based retransmission. Therefore, related work on InfiniBand cannot be directly applied to Ethernet NIC.

B. Zero-copy I/O

As mentioned in Section I, our main goal is to solve the multiple data copying problem that occurs between NIC and GPU memory. To this end, we apply a method called Zero-copy I/O [4]. Zero-copy I/O was originally designed for solving the same problem between general I/O devices and GPU memory, not between NIC and GPU memory. Current designed as PCI I/O devices, which connect with the computer system through PCIe ports.Through base address registers (BARs), the CPU can access the specific memory areas associated with I/O devices. The number of BARs may change depending on the type of I/O devices.

To date, only the first function of Zero-copy I/O has been applied in the HBT-EP Tokamak system [8] of Columbia University. Figure 1 shows a basic model of that implementation. The BAR1 region is points to GPU buffers. This BAR1 region is the second PCIe BAR region among those supported by hardware to serve as a window for the device memory. When a user program requests data transmission with GPU buffers, BAR1 will be mapped to the user program's buffers. Subsequently, when the DMA engine of I/O devices perform data copying with mapped virtual address, data will be automatically moved into GPU buffers, thus bypassing host memory. The results of experiments with HBT-EP Tokamak showed that Zero-copy I/O can help in reducing the latency of data transmission between I/O devices and GPU memory approximately 33%. In this study, both functions of Zero-copy I/O are applied to solve the problem of multiple data copying when data are copied from NIC to GPU memory. There are two methods of data transmission corresponding to the two functions of Zero-copy I/O. Therefore, experiments were conducted to compare the performance of both methods.

III. Problem Definition

Typically, network data transmission in a GPU-based system is performed as follows.

Preparing for data reception: A NIC driver allocates some space in the kernel space of host memory for packet storage. The physical addresses of those buffers are passed to the NIC's DMA controller. The DMA controller can later perform data copying to move packets to the destination buffers. When a user program wants to process network data execution on GPU, it creates its own buffers in host memory and GPU buffers in GPU memory, sends a request to the NIC driver, and enters a waiting state.

Data reception: Figure 2 is a graphical summary of how data are transferred from the network and into GPU memory. The transmission involves five steps:

1) A packet received from the network is temporarily stored in NIC memory.
2) Based on the physical addresses set up before, the DMA controller performs copies packet data from NIC memory to buffer into host memory. When data copying is complete, NIC raises an interrupt to inform the NIC driver.
3) Packet header information is analyzed through protocol stack processing.
4) At the end of protocol stack processing, the rest of the packet is copied into a user program buffer.
5) Once the user program buffer has been filled with data, it gets out of the waiting state and calls an explicit copy function that copies the data from its own buffer and into GPU memory. When all data are in GPU memory, the transmission is considered complete.

The user program will then call a GPU kernel program to process the data.

In the above five-step process, data are copied three times between different memory areas. First, when data are copied by the NIC's DMA controller, from NIC memory and into a kernel buffer. Second, when data are copied from the kernel buffer and into the user buffer. Third, when data are copied from the user buffer and into a GPU buffer by using an explicit copy function. Multiple data copying results in additional latencies for the whole transmission process. In addition, the system has to allocate extra buffers in kernel space and user space

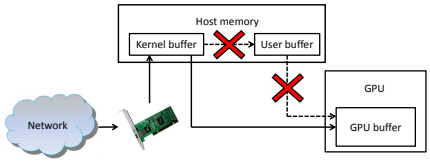

Fig. 3. Model of double data copying on NIC

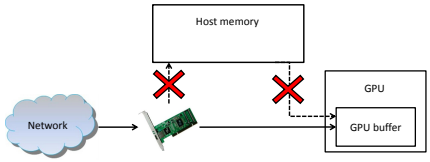

Fig. 4. Model of direct data copying on NIC

for packet storage, all of which results in the reduction of the memory available during data transfer. Moreover, since a user program must wait until its own buffer is filled with data before it can call a copy function, the data throughput of the whole process is badly affected. Consequently, system performance is degraded by this data transmission model.

A. Proposed Data Transmission Method

Two methods of data copying, using two functions of original Zero-copy I/O, to solve the problem of multiple data copying in current scheme of data transmission from NIC to GPU memory are explained briefly in this section.

Double data copying: The basic idea underlying the function of the original Zero-copy I/O method can be used to reduce the number of data copying operations. Figure 3 shows how to transfer data from NIC to the GPU buffers, GPU buffers are mapped to user buffers using the mapping functions described in Section II-B. This method of mapping is similar to the implementation of Zero-copy I/O in the HBT-EP Tokamak system, in which GPU buffers are also mapped to the user buffers. So, when data are copied from kernel buffers to user space buffers, data are automatically moved into GPU buffers, bypassing user buffers. Using this double data copying method, the number of copying operations is reduced from three to two; data just go from NIC and into kernel buffers and from kernel buffers and into GPU buffers.

Direct data copying: The number times that data are copied can be reduced further by applying the second function of the original Zero-copy I/O method. Figure 4 shows how the direct data copying model can be implemented on NIC. Our solution includes three main steps. First, the physical addresses of GPU buffers allocated by the user program are obtained using the second function of the original Zero-copy I/O method. Next, those physical addresses are passed into the NIC driver so that its DMA controller can read them. Finally, when data are transferred from the network, the DMA controller copies them to GPU buffers associated with the physical addresses set up earlier, thereby completely bypassing host memory and eliminating the need for CPU interaction. The total number data copying operations in this case are reduced from three to one, with data going directly from NIC to GPU buffers.

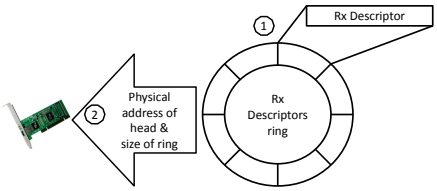

Fig. 5. Rx Descriptors preparation

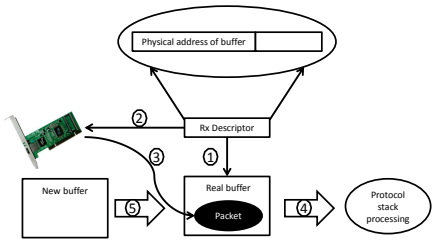

Fig. 6. Data reception with Rx descriptors

IV. Implementation

In this section, we give the details of our implementation of double data copying and direct data copying on the system, which performs data transmission between NIC and GPU. We have made some assumptions to simplify the implementation. First, the protocol stack processing in direct data copying is temporarily ignored. As mentioned in Section II, many existing NICs rely on the TCP/IP protocol to guarantee data transmission reliability. Error checking, routing, etc, are all done in protocol stack processing, which requires that data appear in host memory. Hence, bypassing host memory and eliminating CPU interaction, as happens with Zero-copy I/O, means that protocol stack processing cannot be done. Second, in both double and direct data copying, we only implement Zero-copy I/O on NIC, with a single-task process that performs data reading/writing. Complex scenarios requiring multitask processes are not included in our prototype. Direct data transmission with Zero-copy I/O is implemented on top of Gdev [9], a GPU-resource management environment.

A. Rx descriptors

A detailed explanation of the data transmission from NIC to GPU memory has been given in Section III. To prepare for packet reception, the NIC driver must allocate several buffers to store these packets. To manage those buffers, the NIC driver uses structures called Rx descriptors. (Rx stands for Receiver.) Each buffer used to store a packet has a corresponding Rx descriptor. Information about each buffer is stored in its Rx descriptor. Depending on the type of drivers, the number, as well as the content of data fields in each Rx descriptor, may change, but they always have a very important data field, that is, the bus address of packet buffer. By accessing Rx descriptors, the DMA controller can read the bus addresses of packet buffers and transfer packets to the right destination associated with that address.

Rx Descriptors preparation: Figure 5 shows the process by which Rx descriptors are allocated and initialized.

1) The NIC driver allocates some Rx descriptors using the function dma_alloc_coherent [10]. This function allocates a consistent memory, which is memory for which a write by either the device or processor can

immediately be read by the device or processor without having to worry about caching effects. The number of Rx descriptors allocated may change. In the Intel driver, in which Zero-copy I/O is implemented, the default number of Rx descriptors is 256.

2) The bus address of the Rx descriptors ring and size of the ring are passed to the NIC hardware. This information enables the NIC hardware to access Rx descriptors and read the required information about packet buffers.

After the above procedure, Rx descriptors do not point to anything because at that moment no packet buffer is allocated.

Data transmission with Rx descriptors: Figure 6 shows how the DMA controller performs data transmission using Rx descriptors. This process consists of five steps:

1) The NIC driver allocates several buffers for packet storage and points Rx descriptors to the buffers by storing the bus addresses of these buffers in corresponding Rx descriptors.
2) Based on the bus address of Rx descriptors ring, the DMA controller accesses the Rx descriptors ring to read packet buffers information. The DMA controller then knows where it must copy packets.
3) The DMA controller copies packets into packet buffers. After completing copying, it writes the status of data transmission into Rx descriptors and raises an interrupt to indicate to the NIC driver that data transmission with host memory has been completed.
4) On receiving an interrupt signal from the hardware, the NIC driver calls an interrupt handler to process that signal. Depending on the number of packets the system received, the NIC driver uses one of the two different modes of packet processing.

- Interrupt driven:
  Interrupt-driven processing is outdated and was used when data transmission speed of NIC was quite slow. In this mode, whenever a packet is copied to host memory, the DMA controller raises an interrupt to tell the NIC driver to run an interrupt handler on that packet. When the interrupt handler is running, CPU checks the Rx descriptors content to test the status of transmission and push the packet to a higher layer in the protocol stack.
- NewAPI:
  With improvement in the data transmission speed of NIC became faster, interrupt-driven mode became insufficient as the system would have to process many signals in a short amount of time, thus wasting system resources. To avoid this, NewAPI (NAPI) was proposed. In this mode, instead of raising an interrupt after completing the transmission of each packet, the interrupt is disabled. The system uses polling to check the status of Rx descriptors to detect new packets and also count the number of received packet within a specified time. The packets are then checked and pushed to a higher layer in the protocol stack. If the number of incoming packets drops below a given threshold, which means that there is a small amount of data going through the system, then the polling is disabled, the interrupt is activated, and NIC turns returns to the interrupt-driven mode.

  In both modes, at the end of the packet handler, packets are pushed to the protocol stack's higher layers.

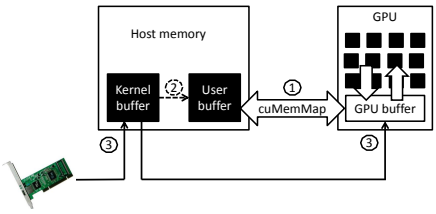

Fig. 7. Detailed implementation of Double data copying on NIC

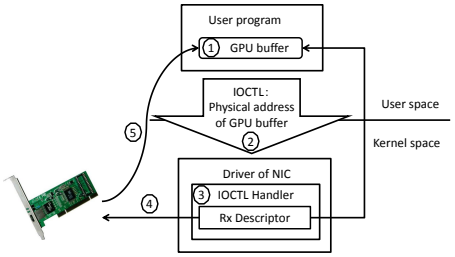

Fig. 8. Detailed implementation of direct data copying on NIC

5) The NIC driver allocates new packet buffers to replace the buffers that went to protocol stack processing, and all the above steps are repeated.

B. Zero-copy I/O implementation with Rx descriptors

Implementation of Double data copying: The implementation of double data copying is very simple, as shown in Figure 7. This process comprises three steps:

1) A user program that allocates some buffers in GPU memory is created. This user program calls to cuMemMap(), a mapping functions provided by the original Zero-copy I/O method and integrated in the Gdev library, and maps all allocated GPU buffers to user space virtual addresses. The virtual address are then associated with GPU buffers instead of user buffers.
2) The user program calls some data reception functions, e.g.,recv(), and points to the mapped virtual addresses.
3) Data transferred from NIC into kernel buffers are copied into GPU buffers using the copy_to_user() function.

Implementation of Direct data copying: The implementation of direct data copying is more complex than double data copying.
The detailed implementation of direct data copying with Zero-copy I/O on NIC is given in Figure 8. Our implementation consists of five steps:

1) As with double data copying, a user program that allocates some buffers in GPU memory is created. The physical address function of original Zero-copy I/O method is then used to calculate the physical addresses of the allocated GPU buffers and the addresses stores in an array.
2) The virtual address and the size of the array in step (1) are passed first to an IOCTL command and then to the NIC driver in kernel space.
3) Within the NIC driver is created a new IOCTL handler that copies the physical addresses of the allocated GPU buffers into kernel space on the basis of the virtual addresses and size of the bus addresses array. Then, the IOCTL handler stores all these physical addresses into

Rx descriptors. The DMA controller then knows that it has to transfer data to the new physical addresses, which are located in GPU memory, not host memory.

4) The DMA controller accesses Rx descriptors and reads the new physical addresses.
5) When NIC receives a new packets, the DMA controller copies that packet to the buffers corresponding to the new physical addresses.

Zero-copy I/O on NIC is then completed and data move directly from NIC to GPU memory. Data copying between NIC and host memory and between host memory and GPU memory are eliminated.

V. Evaluation

A. Evaluation environment

To evaluate the performance of the Zero-copy I/O implementation on NIC, a peer-to-peer connection between two computers is established. One is a client and the other is a server. The client machine runs a " sender " program, which sends a specific number of messages to the server by using a sendto() command with the UDP protocol. The server runs a " receiver " program, which allocates several buffers in GPU memory, receives all packets from the network, and copies them into the allocated GPU buffers. On the server side, three different methods are used to transfer packets from NIC to the buffers:

Triple copying: The flow of data in this method is exactly same as that in the data transmission model with most current systems that use NICs. Network data are transferred from the network to NIC, from NIC to kernel buffers, and from kernel buffers to user buffers, finally ending up in GPU buffers. However, as mentioned at the beginning of Section IV, protocol stack processing is not included. Data are simply copied between different memory areas.

Double copying: Data move from NIC to kernel buffers and then go directly into GPU buffers, bypassing user buffers.

Direct copying: Data directly move from NIC to GPU buffers, ignoring host memory and CPU interruptions.

The client machine consists of an Intel Core i5-2520M 2.5 GHz CPU, 4 GB RAM, an Intel 82579M Gigabit Network Adapter, and Ubuntu OS with kernel 3.8.0. On the server side, the system consists of an Intel Core i7-3770 3.4 GHz CPU, 8GB RAM, NVIDIA GTX480 VGA, an Intel 82579V Gigabit Network Adapter, and Fedora OS with kernel 3.10.0-rc6. A Nouveau [11] GPU driver was modified to run on a Gdev environment. A network adapter driver for PCI-E Gigabit Network Connections under Linux version 2.4.14 [12] was also changed to create the IOCTL command and to disable protocol stack processing with data reception. The above two methods above are evaluated and compared with respect to three factors: data transmission latency, packet loss rate and data completeness, and data throughput.

B. Evaluation results

Data transmission latency: To evaluate how the latency of data transmission changes, the time from when the first packet reaches the destination buffer to when the final packet reaches the destination buffer is measured. To simplify the evaluation, the client machine sent 100 messages to the server.

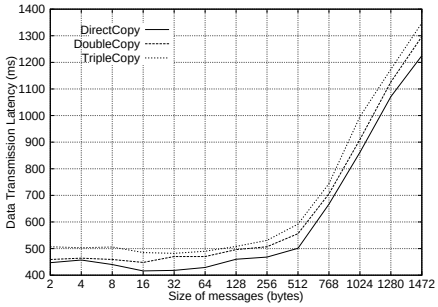

Fig. 9. Data transmission latency

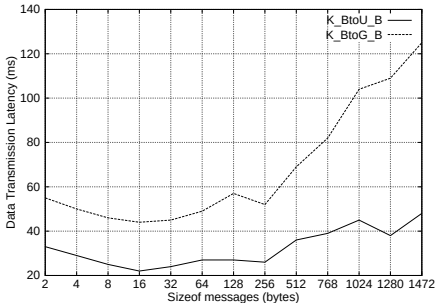

Fig. 10. Data transmission latency for referencing

The sizes of the messages varied from two bytes to 1,472 bytes. Figure 9 shows the result of the data transmission latency experiment. The " *DirectCopy* " is the latency measured in microseconds when 100 messages of corresponding sizes are sent to the server machine by direct data copying. The " *DoubleCopy* " is similar to the DirectCopying, except that the " reciever " program uses double copying instead of direct copying. The " *TripleCopy* " is also similar to the DirectCopying and DoubleCopying, but with copying taking place three times. However, the results in TripleCopying is the latencies of data transmission using the Triple copying.

Figure 10 is used as a reference and to describe how Zero-copy I/O changes the latency of data transmission compared with the traditional method. The " *K_BtoU_B* " represents the latency incurred when data are transferred from kernel buffers to GPU buffers in the case of the triple copying method. The results in this method are used to clarify the reduction in latency of data transmission using double copying relative to the traditional method. As with the " *K_BtoU_B* ", the " *K_BtoG_B* " shows the latency of data transmission from kernel buffers to GPU buffers in the same scheme of data copying. This method is used to clarify the improvement in data transmission using direct copying relative to the traditional method. Figure 9 shows that, compared with the traditional method, Zero-copy I/O can help reduce the latency of data transmission when data are transferred from NIC to GPU memory. Comparing the double and triple methods, the reductions in latency with the double method are quite similar to the result shown in the " *K_BtoU_B* " of Figure 10. With the same transmission time, data can only move from NIC to user buffers, and the system would have to incur extra latency to transfer those data to GPU memory with traditional method. On the other hand, double copying enables data to go to GPU buffers, thus eliminating interaction with user buffers. Similarly, the latency reductions with direct copying, relative to the traditional method, are similar to those " *K_BtoG_B*' of Figure 10, meaning that direct copying has eliminated extra

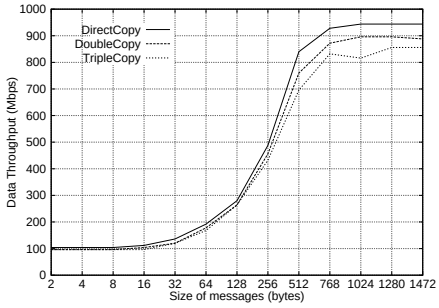

Fig. 11. Data throughput

latencies produced by the data transmission between kernel buffers and user buffers and between user buffers and GPU buffers. The latency reduction with dirct copying is even larger than that with double copying, meaning that direct copying is the fastest method to transfer data from NIC to GPU memory.

However, it is very difficult to measure exactly how much latency is reduced with Zero-copy I/O, because the latency collected in the first table above is not the latency incurred in transferring the 100 packets from NIC to the GPU memory. Packets are created from messages on the client side, and before those packets reache the server side, they have to be processed through the protocol stack, which introduces extra latencies in the data transmission process. Moreover, the protocol stack on the client side cannot be removed, because packets cannot be transferred through the network without their headers.

Packet loss rate and data completeness*:* Here, the baseline is the packet loss rate of the system when applying with Zero-copy I/O. A specific number of messages are sent from the client to the server. On the server side, the number of received packets are counted and compared with the number of sent messages. The packet loss rate is the percentage of lost packet from the total number of sent messages.
To investigate if Zero-copy I/O affects the information inside packets, the client machine continuously sent messages with specific content: each sent messages contained a string of integer number, for instance, 111...1, 222...2,... On the server side, the " receiver " program displayed on the screen the content of packets it received. The displayed content was compared with that of the sent messages.
The result of this evaluation shows that applying Zero-copy I/O does not cause in loss of packets or destruction of the information wrapped in each packet. In both two methods, double copying and direct copying, the packet loss rate are exactly the same as the packet loss rate with triple data copying and the information in each packet is safe.

Data throughput*:* This evaluation set up is the same as for the evaluation of data transmission latency. We estimated the speed at which data were received by the server-side program. The result is shown in Figure 11, and measured in Mbps. Again, this result shows that Zero-copy I/O can help improve system performance relative to that of with the traditional method. Of the three methods, direct copying achieves the highest throughput, because CPU does not need to wait for data transmission from NIC to kernel buffers or from kernel buffers to user buffers to be completed before it can copy those data to GPU using explicit copy function.

VI. Conclusion

This paper introduced the problem of multiple data copying in the existing data transmission scheme in CPS, which require data transmission between NIC and GPU memory. When data are transferred from NIC to GPU memory, they are copied multiple times between different memory areas. Multiple data copying slows down the speed of transmission, reduces the available memory of system and the data throughput. Therefore, the system performance decreases, which may resulting in critical problems, especially in safe-critical systems. To solve the multiple data copying problem, a method based on the implementation of Zero-copy I/O on NIC was introduced. With double data copying, GPU buffers are mapped to user buffers using memory mapping; hence, the replication in data copying operations is reduced from a factor of three times to two times. With direct data copying, by obtaining the physical addresses of GPU buffers and passing these to the NIC driver, through the IOCTL command, the replication in data copying can be reduced—copying need to be done only once. Evaluation experiments proved that applying Zero-copy I/O can help reduce the number of data copying operations, thereby solving the multiple data copying problem, and improve the performance of the whole system, while the packet loss rate and data completeness are not affected.

For future work, our first priority is to solve the protocol stack processing problem that occurs in direct copying, as described in Section IV. TCP Offload Engine NIC could solve this problem. Other future work includes solving the multiple data copying problem in the opposite direction, i.e., when data are transferred from GPU memory to NIC.

References

[1] J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade, "Gpu-accelerated real-time 3d tracking for humanoid locomotion," in *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

[2] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice." in *ICRA*. IEEE, 2011, pp. 4889–4895.

[3] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for gpu computing," in *Proc. of the IEEE 19th International Conference on Parallel and Distributed Systems*, 2013.

[4] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," in *In Proceedings of the 4th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'13)*, 2013.

[5] NVIDIA, "GPUDirect."

[6] Mellanox, "Accelerating High Performance Computing with GPUDirect RDMA," 2013.

[7] ——, "NVIDIA GPUDirect Technology-Accelerating GPU-based Systems," 2010.

[8] N. Rath, S. Kato, J. Levesque, M. Mauel, G. Navratil, and Q. Peng, "Fast, multi-channel real-time processing of signals with microsecond latency using graphics processing units," *Review of Scientific Instruments*, vol. 85, no. 4, p. 045114, 2014.

[9] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System," in *Proc. of the USENIX Annual Technical Conference*, 2012.

[10] D. S. Miller, R. Henderson, and J. Jelinek, "Dynamic DMA mapping guide," https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt.

[11] Linux Open-Source Community, "Nouveau Open-Source GPU Device Driver," http://nouveau.freedesktop.org/.

[12] Intel, "Network Adapter Driver for PCI-E Gigabit Network Connections under Linux," http://downloadcenter.intel.com.