OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing

Ryohei Kobayashi
Center for Computational Sciences
University of Tsukuba
1-1-1 Tennodai, Tsukuba,
Ibaraki 305-8577 Japan
kobayashi@cs.tsukuba.ac.jp

Yuma Oobata
Graduate School of Systems and
Information Engineering
University of Tsukuba
1-1-1 Tennodai, Tsukuba,
Ibaraki 305-8573 Japan

Norihisa Fujita
Center for Computational Sciences
University of Tsukuba
1-1-1 Tennodai, Tsukuba,
Ibaraki 305-8577 Japan

Yoshiki Yamaguchi
Faculty of Engineering Information
and Systems
University of Tsukuba
1-1-1 Tennodai, Tsukuba,
Ibaraki 305-8573 Japan

Taisuke Boku
Center for Computational Sciences
University of Tsukuba
1-1-1 Tennodai, Tsukuba,
Ibaraki 305-8577 Japan

ABSTRACT

Field programmable gate arrays (FPGAs) have gained attention in high-performance computing (HPC) research because their computation and communication capabilities have dramatically improved in recent years as a result of improvements to semiconductor integration technologies that depend on Moore's Law. In addition to FPGA performance improvements, OpenCL-based FPGA development toolchains have been developed and offered by FPGA vendors, which reduces the programming effort required as compared to the past. These improvements reveal the possibilities of realizing a concept to enable on-the-fly offloading computation at which CPUs/GPUs perform poorly to FPGAs while performing low-latency data movement. We think that this concept is one of the keys to more improve the performance of modern heterogeneous supercomputers using accelerators like GPUs. In this paper, we propose high-performance inter-FPGA Ethernet communication using OpenCL and Verilog HDL mixed programming in order to demonstrate the feasibility of realizing this concept. OpenCL is used to program application algorithms and data movement control when Verilog HDL is used to implement low-level components for Ethernet communication. Experimental results using ping-pong programs showed that our proposed approach achieves a latency of $0.99 \mu\text{s}$ and as much as 4.97 GB/s between FPGAs over different nodes, thus confirming that the proposed method is effective at realizing this concept.

KEYWORDS

FPGA, OpenCL, BSP, I/O channels, Verilog HDL, Heterogeneous cluster, Accelerator computing, Interconnect for accelerators, Ethernet communication, QSFP+

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HPC Asia 2018, January 28–31, 2018, Chiyoda, Tokyo, Japan

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5372-4/18/01.

<https://doi.org/10.1145/3149457.3149479>

1 INTRODUCTION

Graphics processing units (GPUs) can offer good peak performance and high memory bandwidth. They have been widely used in high-performance computing (HPC) systems as accelerators. However, in order to make parallel applications run on such heterogeneous clusters, inter-accelerator communication over the nodes is required. This means that multiple memory copies through the CPU are performed, which causes an increase in latency that can severely affect application performance, particularly when a short message is involved. Besides, the GPU has above strong characteristics, but it is not almighty as an accelerator because it is not effective in applications that employ complicated algorithms using exception, non single-instruction-multiple-data streams (SIMD), partially poor parallelism, etc.

To address these problems, field programmable gate arrays (FPGAs) have emerged in this area of research. The FPGA is a semiconductor device on which designers can program and reprogram any digital circuit, thus enabling implementation of circuits that realize application-specific pipelined hardware and data supply systems. In [1], the authors proposed a PCI express (PCIe)-based interconnect for accelerators that can reduce communication latency between them over different nodes. They also developed an FPGA-based network interface card to support direct communication through the PCIe protocol. In addition to the communication logic, the authors in [2, 3] implemented application-specific computation logics on FPGAs, which enabled on-the-fly offloading specific computation to FPGAs while transferring data between nodes, and they achieved a significant improvement in performance. We think that this realization of low-latency communication-enhanced parallel processing running on multiple FPGAs connected by a high-speed interconnect is crucial to more improve the performance of modern HPC systems that use accelerators. Figure 1 illustrates what this concept looks like. Accelerators such as GPUs are used for coarse-grained parallel applications, whereas multiple FPGAs connected by a high-speed interconnect autonomously perform communication and computation where CPUs/GPUs are weak.

One issue to address in realizing this concept is how to implement application-specific hardware on FPGAs. Typically, register transfer level (RTL) modeling with hardware description languages (HDLs) such as VHDL or Verilog HDL remains the de facto standard method. However, it incurs huge implementation costs when compared to classic software development because few abstraction capabilities are supported by conventional software programming languages. In addition, because hardware behaviors have to be specified strictly, source code portability is quite low and hardware must be reimplemented for each application. This means that implementing HPC application-specific hardware using the traditional methods is unrealistic.

To solve these problems and increase the appeal of FPGAs, particularly to software programmers, Intel [4] and Xilinx [5] have recently offered open computing language (OpenCL)-based FPGA development environments. OpenCL [6] is a royalty-free, open source, and portable parallel programming language that functions across a variety of computing devices. The FPGA toolchains can convert OpenCL kernels to synthesizable RTL designs, which renders FPGA development more efficient. However, assuming that these toolchains are used to realize the concept, determining how to control data movement between FPGAs over the nodes from OpenCL code is mandatory.

In this paper, we propose a method of high-performance FPGA-to-FPGA data movement that can be controlled using OpenCL code in order to show the feasibility of the concept. We assume that the number of nodes is scaled to hundreds or thousands. Its communication protocol is Ethernet and each FPGA is connected to an Ethernet switch. To control inter-FPGA Ethernet communication over the nodes through OpenCL code, implementing a communication logic which can interpret Ethernet protocol on an FPGA is necessary. Because considerable developmental effort is required to implement it from scratch, we use an Ethernet intellectual property (IP) core offered by an FPGA vendor and implement an interface between the OpenCL code and IP core by using Verilog HDL. In other words, we demonstrate FPGA-to-FPGA Ethernet communication using OpenCL and Verilog HDL mixed programming. Here, OpenCL is used to develop application algorithms and data movement control, and Verilog HDL is used to implement low-level components such as an interface. We evaluate Ethernet communication latency and bandwidth between FPGAs via quad small form-factor pluggable plus (QSFP+) optical interconnect using ping-pong programs. The QSFP+ is an industrial standard for high-density and low-power Gigabit Ethernet communication that achieves data rates of from 40 to 100 Gbps. As a result, its latency is 0.99 μ s and effective bandwidth is as much as 4.97 GB/s, thus achieving 99.4 % of the theoretical peak performance. We confirm high-performance inter-FPGA Ethernet communication for the concept can be realized.

Our contributions in this paper are threefold:

- We propose an approach for transferring data between FPGAs over nodes, which can be controlled using OpenCL code to realize low-latency communication-enhanced parallel processing running on multiple FPGAs connected by a high-speed interconnect.

Figure 1: Low-latency communication-enhanced parallel processing running on multiple FPGAs connected by a high-speed interconnect

Figure 2: Intel FPGA SDK for OpenCL programming model

- We detail our implementation with OpenCL and Verilog HDL mixed programming to realize the proposed approach.
- We experiment using ping-pong data communication to evaluate the latency and bandwidth between FPGAs over the nodes, and show that conducting high-performance inter-FPGA Ethernet communication can be controlled through OpenCL code.

This paper is organized as follows. In Section 2, we introduce Intel FPGA SDK for OpenCL and describe how to access an FPGA board hardware resource from OpenCL code using the toolchain features. Our proposed approach and implementation are detailed in Section 3, and our experimental evaluation of latency and bandwidth is described in Section 4. We introduce several previous studies in Section 5, and finally this paper is concluded in Section 6.

2 INTEL FPGA SDK FOR OPENCL

2.1 Overview

As an OpenCL-based FPGA developmental toolchain, we use Intel FPGA SDK for OpenCL [4], the programming model for which is shown in Figure 2. Similar to when targeting common computing devices such as CPUs and GPUs, programmers must write two kinds of code: OpenCL host and OpenCL kernel. The host code is

Figure 3: Schematic of the Intel FPGA SDK for OpenCL platform

for programming a host application running on a host PC to manage an FPGA device at runtime with a set of common application programming interfaces (APIs), whereas the kernel code is for designing computation parts offloaded to the FPGA.

In this programming model, the host and kernel codes are compiled separately, because only offline compilation is supported due to the fact that RTL synthesis takes several hours. The host code is compiled using a standard C compiler like GCC on Linux or Visual Studio C/C++ on Windows to generate a host binary. The kernel code is compiled using an Intel FPGA OpenCL compiler offered by the toolchain to convert into synthesizable Verilog HDL files, which are then used in Quartus Prime to generate an aocx file that includes FPGA configuration information. The aocx file is downloaded to the FPGA at runtime of the host application by using the APIs, and data required for the kernel execution as well as its resulting data are transferred via the PCIe bus.

2.2 Board Support Package

Board support packages (BSPs) are provided from FPGA board vendors like Bittware and are required during kernel code compilation. Unlike with a CPU and motherboard, control and access between an FPGA chip and peripheral components such as external memories are different for each FPGA board, and an application running on an FPGA board cannot run on another FPGA board unless programmers address its difference. This is why BSPs are required and can absorb it, as BSPs offer FPGA board-specific parameters, hardware components, and software drivers. As a result, it is possible to port existing OpenCL kernel code for an FPGA board to any other board, as long as its BSP is available.

Figure 3 shows a schematic of the Intel FPGA SDK for an OpenCL platform. As previously described, the host application is implemented using the OpenCL host code, and the application-specific pipelined hardware is generated from the OpenCL kernel code. The PCIe and external memory controllers are offered by the BSP and are automatically connected to the pipelined hardware during kernel code compilation. The PCIe software driver is also provided by the BSP and enables data movement between the host PC and FPGA boards. In other words, programmers essentially do

Figure 4: Integrating an LED controller into a BSP and Qsys screenshot

```

<channels>
  <interface name="led_st_0" port="led_in"
    type="streamsink" width="32" chan_id="led0" />
</channels>

```

Figure 5: Description appended to board_spec.xml file

```

1 channel uint outLED
2 __attribute__((depth(0))) __attribute__((io("led0")));
3
4 __kernel void led(int N)
5 {
6     write_channel_intel(outLED, N);
7 }

```

Figure 6: OpenCL kernel code using an I/O channel to access the LED

not have to be concerned about anything other than the host and kernel code implementation.

BSPs support external hardware component access from an FPGA (OpenCL kernel code). However, they basically provide a minimum set of peripheral controllers to enable OpenCL programming (i.e., PCIe and external memory controllers). In other words, if BSPs do not offer peripheral controllers to access external hardware on an FPGA board that programmers want to use, they must implement BSPs including those controllers on their own.

2.3 Peripheral Accesses from OpenCL Kernel Code using I/O Channels

Intel FPGA SDK for OpenCL offers an *I/O Channel* API to access external hardware on an FPGA board from OpenCL kernel code. This is a vendor extension of OpenCL, and streaming data, for example, into an OpenCL kernel directly from a streaming I/O interface such as Gigabit Ethernet is possible [4]. Here, we explain how to access a peripheral (LED) from OpenCL kernel code using an I/O channel.

As described in the previous section, BSPs provide peripheral controllers that enable access to external hardware from OpenCL kernel code. Otherwise, programmers have to implement their own

BSPs to include these controllers. First, programmers must implement an LED controller and integrate it into a BSP. This controller is implemented with HDL and is connected to the interconnect using an Intel FPGA development tool called Qsys [7]. Figure 4 shows what this tool looks like. In the Qsys screenshot, `led_st_0` represents the LED controller implemented with HDL.

Then, modifying `board_spec.xml` file included in the BSP is necessary in order to associate an I/O channel used in OpenCL kernel code with the implemented controller. This extensible markup language (XML) file contains information about FPGA board hardware resources such as external memory accessed from the kernel code. Figure 5 shows a description appended to its file. In *interface* tag, these attributes of *name*, *port*, *chan_id*, *type*, and *width* specify the controller name used in the Qsys, the port of the controller used for data movement, the I/O channel used in the kernel code, the data movement direction (input or output), and the bit width for the data movement, respectively. According to the figure, port `led_in` of `led_st_0` (the LED controller) is available as an I/O channel named `led0` in the kernel code in order to send data.

Figure 6 shows the OpenCL kernel code using an I/O channel to access the LED. The I/O channel variable `outLED` is declared at line 1. The `io` attribute in the code specifies the I/O channel to be used for data movement. The attribute `io` (“`led0`”) is described at line 2, and the value used in the attribute corresponds to the value of *chan_id* attribute in Figure 5. Then, the data `N` is sent to the I/O channel using `write_channel_intel` function at line 6. This is an embedded function to write the second argument value into the I/O channel variable.

Figure 7 shows the LED access from the OpenCL kernel code. The Terasic DE1-SoC development kit [8] was used for this experiment and offers an OpenCL programming environment. This kit has a Cyclone V SoC chip that contains not only an FPGA but also an ARM Cortex-A9 processor as a fixed processing unit. In addition, a Linux OS running on the processor controls the OpenCL computing device (FPGA) using a dedicated interconnect instead of the PCIe. As shown in the figure, the four red LEDs are controlled from the kernel code. The command line argument “3” (= 0011b) is moved from the host application to the LED blinking control logic generated from the kernel code using the I/O channel. Then, the value is sent to the LEDs through the integrated LED controller. The corresponding positions emit light and it is obvious that the value written into the I/O channel is reflected in the LED light emission.

3 METHODOLOGY

In this section, we describe inter-FPGA Ethernet communication through a QSFP+ optical interconnect using OpenCL and Verilog HDL mixed programming. To accomplish this, we follow the method described in Section 2.3, which is to modify a BSP in order to make use of the I/O Channel API. In other words, we implement a QSFP+ controller that can interpret the Ethernet protocol, integrate it into a BSP, and write OpenCL kernel code to control it. Figure 8 is a block diagram of the QSFP+ controller and shows the data paths. The controller consists of two hardware components: the Ethernet IP core and a hardware logic to control it. These are detailed in the following sections.

Figure 7: Testing LED access from the OpenCL kernel code on the Terasic DE1-SoC development kit

3.1 Ethernet IP Core

An IP core is a reusable hardware component that offers a specific function, which corresponds to a library in software programming. To make the QSFP+ controller implementation efficient, we use the Ethernet IP core [9], low-latency 40- and 100-Gbps Ethernet MAC and PHY MegaCore function, which offers essential features to transfer data in Ethernet protocol and is provided from Intel. As a result of this IP core, programmers do not have to implement any functions of the physical and media access control layers for Ethernet communication. However, please note that the IP core does not support any protocols above these layers.

Figure 9 shows an Ethernet frame generated in the IP core. In Figure 8, the frame generator adds the three kinds of data (preamble, cyclic redundancy check (CRC), and EFD) to the client data composed of media access control (MAC) addresses that include both the source and destination as well as the client payload. These three types of data show an eight-byte preamble that begins with a start byte to indicate the head of the Ethernet frame, a CRC-32 checksum, and one-byte data to indicate the end of the Ethernet frame, respectively. Both MAC addresses are 6 bytes, and the client payload is between 39 bytes and approximately 9 KiB. The variation in data size depends on Ethernet switches.

The Ethernet frame output from the frame generator is passed through the serializer/deserializer (SerDes) and converted into four-lane serial data. Its value of four comes from the QSFP+ specification. The serial signals are sent and received to and from another FPGA via the QSFP+ port, and the received signals are converted into frame data after passing through the SerDes. The client data is retrieved at the frame extractor, and the frame checker with CRC-32 is performed simultaneously. The IP core performs a series of these processes automatically, and programmers must only be concerned with handling the client data. The IP core also automatically obtains the client data size and deals with it as a jumbo frame if it is beyond 1500 bytes. It then inserts a 12-byte inter-packet gap (IPG) between the Ethernet frames to satisfy the IEEE 802.2 Ethernet standard during sending.

3.2 Ethernet IP Controller

As shown in Figure 8, the Ethernet IP controller is the interface between the hardware unit generated from OpenCL kernel code

Figure 8: QSFP+ controller block diagram and data paths

Figure 9: Ethernet frame

and the Ethernet IP core, which is our homemade hardware unit implemented with Verilog HDL.

Similar to the LED example described in Section 2.3, data required by the Ethernet IP controller is sent from OpenCL kernel code using the I/O channel API and indicates the MAC addresses and the client payload. The prepender first selects and sends the MAC addresses, and then extracts data from the client payload stored in the first in first out (FIFO) buffer.

The client data extracted from the IP core is sent to the controller. In this paper, the MAC address is discarded at the remover shown in the figure and the payload data is stored in the FIFO buffer. The controller extracts it from the buffer and performs data transfer with the I/O channel API when detecting that a hardware unit generated from the OpenCL kernel code is ready to receive it.

Basically, the MAC address is essential to identify the FPGA that sends data, but we implemented a minimum set of components to perform inter-FPGA Ethernet communication due to time constraints. Therefore, although our proposed approach can perform FPGA-to-FPGA Ethernet communication, it currently supports only point-to-point data movement, and this is why the MAC address was removed. Furthermore, programmers have to handle retransmission and flow controls by themselves when transmission errors occur. This is because the Ethernet IP core does not support any communication protocol above the data link layer. The IP core detects transmission errors and generates their error messages. However, we did not implement those error handling controllers using them for the same reason.

3.3 Ethernet Communication from OpenCL Kernel Code using I/O Channels

Similar to the LED example, the implemented QSFP+ controller is integrated into a BSP, and modification of its board_spec.xml file

and implementation of OpenCL kernel code to control the controller are necessary, as shown in Figure 10. Between the hardware unit and the Ethernet IP controller identified by the *name* attribute, all data is passed through input or output ports of the controller identified by the *port* attribute.

As shown in the kernel code, the MAC addresses and client payload are first passed from the host application and then are sent to the Ethernet IP controller using the `write_channel_intel` function previously described Section 2.3. To make use of the embedded function, the I/O channel variables are defined with the *chan_id* attributes. The data widths of both MAC addresses are 6 bytes. However, in our implementation, the data lane widths between the hardware unit and controller are set as 4 bytes (32 bits) as a result of the data types supported by OpenCL. Thus, the IP controller assembles 4 bytes of data into a MAC address at the prepender, by using of padding process. On the other hand, the data lane width of the payload is 32 bytes (256 bits), which depends on the maximum size of the data that the Ethernet IP core can take and send from and to the IP controller per clock cycle. Therefore, if sending more than 32 bytes of data from the OpenCL kernel code is required, sending data repeatedly using a loop statement is necessary.

The hardware unit receives the client payload sent from the Ethernet IP controller through the `read_channel_intel` function in the OpenCL kernel code. This is also an embedded function and returns data if an I/O channel variable is specified as the function's argument. Unlike cases of sending, the *type* attribute has to be set as "streamsource" to indicate the data movement direction from the controller to the hardware unit. The data lane width is also 32 bytes (256 bits) for the same reason previously mentioned, and a loop statement is used to obtain more than the 32 bytes of data sent from the controller.

3.4 FPGA Transceiver Parameter Settings for Ethernet Communication via the QSFP+

In our proposed approach, the QSFP+ optical interconnect is used to perform Ethernet communication at ultra-high speed, and specific settings are required. The QSFP+ operates on the order of gigahertz, which often causes transmission errors because the pulse signal of the sent data is disturbed as a result of an operating frequency that is too high, as shown in Figure 11 (a). To prevent this, setting parameters of the transceiver correctly is mandatory. The

Description appended to board_spec.xml file

```
<channels>
  <interface name="ether_ip_con" port="kernel2con_sadr" type="streamsink"
    width="32" chan_id="kernel_send_sadr"/>
  <interface name="ether_ip_con" port="kernel2con_dadr" type="streamsink"
    width="32" chan_id="kernel_send_dadr"/>
  <interface name="ether_ip_con" port="con2kernel_data" type="streamsource"
    width="256" chan_id="kernel_send_data"/>
  <interface name="ether_ip_con" port="con2kernel_data" type="streamsource"
    width="256" chan_id="kernel_recv_data"/>
</channels>
```

OpenCL kernel code to perform Ethernet communication

```
/* I/O channel variable definition */
channel int set_src_addr __attribute__((depth(0))) __attribute__((io("kernel_send_sadr")));
channel int set_dst_addr __attribute__((depth(0))) __attribute__((io("kernel_send_dadr")));
channel int8 set_data __attribute__((depth(0))) __attribute__((io("kernel_send_data")));
channel int8 get_data __attribute__((depth(0))) __attribute__((io("kernel_recv_data")));

__kernel void ether_test(
  const global int8 *restrict send_data,
  global int8 *restrict recv_data,
  const int src_addr,
  const int dst_addr,
  const int datanum)
{
  int i;

  // Set MAC Addresses
  write_channel_intel(set_src_addr, src_addr);
  write_channel_intel(set_dst_addr, dst_addr);

  // Set payload for sending
  for (i = 0; i < datanum; i++) write_channel_intel(set_data, send_data[i]);

  // Get payload received
  for (i = 0; i < datanum; i++) recv_data[i] = read_channel_intel(get_data);
}
```

Figure 10: Board_spec.xml file modification, implementation of the OpenCL kernel code for the Ethernet communication, and data paths between a hardware unit generated from the kernel code and the Ethernet IP controller

Figure 11: Pulse signal of the sent data (a) before and (b) after calibration

transceiver is a firm function unit for communication that is built within an FPGA and used for the Ethernet IP core implementation.

For its parameter settings, we used the Intel transceiver toolkit [10]. It offers real-time access to transceiver settings through the joint test action group (JTAG) chain to facilitate the communication test and optimization. We set two parameters: V_{OD} and the pre-emphasis 1st post-tap. The former is the amplitude of the pulse signal of the sent data and the latter is used to change the waveform. With the parameter settings correctly configured, the envelope shown in the figure (b) is generated and the receiver can capture the sent data by recognizing it as a pulse signal in a normal square waveform. These parameters can be saved as the initial value of a setting file in order to replay the communication status. In addition, the file can be integrated to the BSP.

4 EVALUATION

4.1 Experimental Settings

Here, we describe our experimental environment to evaluate our proposed approach in terms of latency and bandwidth. Table 1 shows our experimental machine configuration. This is a heterogeneous cluster composed of three kinds of devices: CPUs, GPUs, and FPGAs. The number of nodes is six and each node has two Intel Xeon E5-2660 v4 CPUs, two NVIDIA P100 GPUs, and a single

Table 1: Our testbed specification

Node	
CPU	Intel Xeon E5-2660 v4 @ 2.00 GHz × 2 sockets Broadwell-EP 14 cores/socket
Memory	DDR4 2400 MHz × 8 ch, 64 GiB
Motherboard	Supermicro X10DRG-Q
GPU	NVIDIA Tesla P100 × 2 devices (PCIe Gen3 x16 card version)
Memory	16 GiB CoWoS HBM2 @ 732 GB/s with ECC
InfiniBand	Mellanox ConnectX-4 Single-port EDR MCX455A-ECAT
FPGA board	BittWare A10PL4 (PCIe Gen3 x8 connection)
FPGA	Intel Arria10 GX 10AX115N3F40E2SG
Memory	DDR4 2133 MHz × 2 ch, 8 GiB
I/O	QSFP+ (40 Gbps) × 2 ports
System	
No. of nodes	6
Interconnection	Mellanox MSB7790-ES2F (for InfiniBand) Mellanox MSN2100-CB2R (for FPGAs)
OS	CentOS 7.3 kernel-3.10.0-514.16.1.el7.x86_64
Compiler	gcc 4.8.5
MPI library	MVAPICH2-GDR 2.2
FPGA toolchain	Intel Quartus Prime Pro, Intel FPGA SDK for OpenCL Version 17.0.0 Build 289

BittWare A10PL4 FPGA board [11] connected to the CPU through a PCIe Gen3 x8 interface. Please note that in this evaluation, we did not use any GPUs.

For data movement between nodes, a single Mellanox InfiniBand ConnectX-4 EDR adapter card offering 100 Gbps communication speed is installed for each node, and as a result, each host (CPUs) can transfer data directly to another via the InfiniBand network using a single Mellanox MSB7790-ES2F InfiniBand switch. However, FPGAs are not connected to the network directly, and if data movement occurs between FPGAs over the nodes via the network, performing memory copies between an FPGA and its host is

Figure 12: Two communication paths for FPGA-FPGA communication on our platform: via direct Ethernet (green) and via CPU + InfiniBand (red)

```

via Ethernet
write_channel_intel(SET_SOURCE, source_addr);
for(i = 0; i < loop; i++){
    write_channel_intel(SET_DEST, dest_addr);
    for(j = 0; j < data_size; j++){
        write_channel_intel(SEND, send_data[j]);
        for(j = 0; j < data_size; j++){
            receive_data[j] = read_channel_intel(RECEIVE);
        }
    }
}

via IB
for(i = 0; i < loop; i++){
    clEnqueueReadBuffer(..., size, data, ...);
    MPI_Send(data, size, ...);
    MPI_Recv(data, size, ...);
    clEnqueueWriteBuffer(..., size, data, ...);
}

```

Figure 13: Two code snippets for FPGA-FPGA communication on our platform: via direct Ethernet (top) and via CPU + InfiniBand (bottom)

required. By contrast, for the traditional accelerators such as GPU, approaches exist to reduce the memory copy between the accelerator and network devices. For instance, GPU Direct RMA (GDR) by NVIDIA enables direct memory copy between these device memories via PCIe in order to skip host memory relay that increases communication latency. However, this advanced feature is not currently offered for FPGA. Therefore, we relied on a classical method to pass the data from FPGA to the communication device via CPU memory. This kind of heavy latency clearly prevents high performance on parallel FPGA computing with a number of computation nodes for HPC.

The Bittware A10PL4 has two QSFP+ ports, each of which enables 40 Gbit Ethernet communication, but we used a single port in this evaluation. Every FPGA is connected to a single Mellanox MSN2100-CB2R Ethernet switch, which forms a star network topology. Please note that the host (CPUs) is not connected to this network directly, and this is why memory copies between them are required if inter-host data movement over the nodes is performed via this network. The Ethernet switch supports up to 100 Gbit communication speed, but we set each port data rate of the switch as

Figure 14: Ping-pong latency comparison between the two paths

40 Gbps to the FPGA board's communication capability. Furthermore, we set the Ethernet maximum transmission unit (MTU) to the switch's upper limit of 9216 bytes in order to use jumbo frames.

In this evaluation, we performed the ping-pong data communication to measure the latency and bandwidth of inter-FPGA Ethernet communication over the nodes, and compared them to InfiniBand-based data movement shown in Figure 12. Please note that using QuickPath Interconnect (QPI) for the InfiniBand-based data communication depended on our system configuration and there was no intention to give our proposed approach any advantages. For the InfiniBand-based data, we implemented a ping-pong program with C++ using OpenCL APIs and message passing interface (MPI). The APIs were used for memory copies between an FPGA and its host, whereas MPI was intended for data movement using InfiniBand.

Figure 13 shows that two code snippets for the Ethernet- and InfiniBand-based data communications. Although it would seem that the bottom one looks easier than the top regarding ping-pong data communication, programmers must be aware of not only the data movement control using the OpenCL APIs and MPI but also computation parts offloaded to the FPGA when more practical applications like the Himeno benchmark are involved. In addition, it is obvious that its communication latency is increased. Compared to this, our proposed approach offers an all-in-one programming model, which means that programmers can consider both computation and communication under OpenCL kernel programming. As a result, the programming costs decrease. In addition, because the FPGA is directly connected to another, the communication latency is obviously reduced as well.

4.2 Latency

Figure 14 shows the ping-pong latency comparison between the two paths when transferring 1-byte data. In the Ethernet-based data movement, the 1-byte data movement is equivalent to sending 32-byte data because of the 256-bit data lane width. The latency of the Ethernet-based data movement was measured by using clock cycle counters implemented in the Ethernet IP controller, and that of the InfiniBand-based data movement was performed with the `clock_gettime()` function running on the host. As a result, the former was 0.99 μs, whereas the latter was 29.03 μs.

Table 2: Hardware resource usage

	ALMs	Registers	M20K memory blocks	Transceivers
Ethernet IP Core	12,685 (3.0 %)	23,132 (1.3 %)	13 (0.5 %)	8 (8.3 %)
Ethernet IP Controller	417 (0.1 %)	998 (0.1 %)	28 (1.0 %)	0 (0.0 %)
OpenCL Kernel	9,707 (2.3 %)	23,664 (1.4 %)	89 (3.3 %)	0 (0.0 %)
Others	27,741 (6.5 %)	52,852 (3.1 %)	228 (8.4 %)	16 (16.7 %)
Total	50,550 (11.9 %)	100,646 (5.9 %)	358 (13.2 %)	24 (25.0 %)

Figure 15: Ping-pong bandwidth comparison between the two paths

To investigate the results of the InfiniBand-based data movement in detail, we measured the latency of the memory copies between an FPGA and its host, and that of the inter-CPU data movement over the nodes using QPI and InfiniBand. We clarified that the memory copies between them are dominant. They involved transferring data from the FPGA/the host to the host/the FPGA, and the latency summation was $27.70 \mu\text{s}$, whereas that of the inter-CPU data movement was $1.33 \mu\text{s}$. This was due to the interface offered by the Bittware A10PL4 BSP.

Latency breakdown of the Ethernet-based data movement is shown in the figure. The three parts shown represent latencies for passing data through the Ethernet IP controller, Ethernet IP core, and Ethernet switch, respectively. Because the Ethernet IP and Ethernet switch are vendor products, their latencies are fixed values, which means that optimizing them is nearly impossible. By contrast, the Ethernet IP controller is our home-built hardware component and can be improved. This means that the latency of inter-FPGA Ethernet communication might be superior to that of inter-CPU data movement with InfiniBand because the two are nearly the same even without the desired improvement.

4.3 Bandwidth

Figure 15 shows the ping-pong bandwidth results for various data sizes with respect to the two paths. Because of the low latency of the Ethernet-based data movement described in the previous section, the maximum effective bandwidth was achieved at the earlier phase compared to the InfiniBand-based data movement. The maximum was 4.97 GB/s , which is 99.4% of the theoretical peak performance of the QSFP+ optical interconnect itself, which is 5 GB/s .

On the other hand, the effective bandwidth of the InfiniBand-based data movement was as much as 2.32 GB/s , because the data movement was performed in a store-and-forward manner and not pipelined. First, a memory copy from an FPGA to its host was performed. When this process was completed, the host sent data to another host via the InfiniBand network. Finally, data movement from the CPU to the FPGA was performed after the entire data was received. As a result, the theoretical peak performance of the InfiniBand-based data movement can be given as:

$$\frac{N}{\frac{N}{7 \text{ GB/s}} + \frac{N}{12.5 \text{ GB/s}} + \frac{N}{7 \text{ GB/s}}} = 2.73 \text{ GB/s} \quad (1)$$

where N represents the data size and both bandwidths of 7 and 12.5 GB/s derive from the theoretical peak performance of PCIe Gen3 x8 and InfiniBand EDR, respectively. Therefore, the maximum effective bandwidth is 85.0% of the theoretical peak performance, which indicates that the data movement efficiency is inferior to that of the Ethernet-based one.

4.4 Resource Consumption

Table 2 shows the FPGA resource usage breakdown for the proposed method. “Others” in this table represents hardware components offered by the Bittware A10PL4 BSP, including the DDR4 memory and PCIe controllers. The adaptive logic module (ALM) is a term used by Intel and is a logic component that includes a logically partitionable lookup table (LUT) and several registers (flip-flops). The ALM utilization is one of the metrics to estimate the size of the area of the hardware components implemented in the FPGA. The M20K memory block is an internal memory (hard macro) of the FPGA, and internal buffers such as FIFOs are basically implemented with the memory blocks. The transceiver is a built-in hardware component for communication as previously described, and our FPGA transceiver offers data rates of as much as 17.4 Gbps chip-to-chip and 12.5 Gbps backplane.

As shown in the table, the ALM utilization is 11.9% to realize a minimum set of inter-FPGA Ethernet communication controlled from the OpenCL kernel code. This means that using the remaining part to implement application-specific computation logics is possible. However, this depends on the method for implementing other mandatory hardware components such as retransmission and flow control logics, because our approach currently does not support any communication protocol above the data link layer as a result of time constraints. To implement these control logics, not only ALMs but also M20K memory blocks are consumed to a greater

extent and this is the reason we must consider how to save hardware resources. The transceiver is used for the Ethernet IP core and PCIe controller, and total consumption is 24, or 8 and 16, respectively. Each transceiver consumption mainly depends on the number of lanes for sending and receiving data.

5 RELATED WORK

Many other groups of researchers have been interested in OpenCL programming with FPGAs for HPC applications [12–14].

In [12], the authors ported and optimized a subset of the Rodinia benchmark suite to an FPGA platform using Intel FPGA SDK for OpenCL, and they compared the performance and energy efficiency between a modern CPU and GPU. Their evaluation showed that in most benchmarks, only the energy efficiency was superior to the GPU, whereas both the performance and energy efficiency were better than those of the CPU.

Weller et al. [13] proposed a comprehensive set of OpenCL optimization techniques for a partial differential equation including data-set optimization, algorithmic enhancements, as well as data- and control-flow tuning methods that improve the performance and energy efficiency by several orders of magnitude. In addition, the authors compared the FPGA implementations between Intel and Xilinx, and showed that fundamentally different optimization approaches for Intel and Xilinx are required to make OpenCL code efficient. This is very interesting because the Xilinx OpenCL-based toolchain has been offered over the past year or two and few comparative experiments have been conducted between them.

In addition, different frameworks for automatically generating OpenCL code for FPGAs have emerged. Lee et al. [14] proposed an open accelerator (OpenACC)-based framework using an open-source OpenACC compiler called open accelerator research compiler (OpenARC) [15], which can convert C code using OpenACC directives to OpenCL code compatible with Intel’s toolchain. Preliminary evaluation of the OpenACC benchmarks on an FPGA as well as a comparative study of GPUs and a Xeon Phi showed that unlike those accelerators, the FPGA’s unique capability to enable implementation of hardware units dedicated to an input program offers considerable performance tuning opportunities.

Other related studies have been conducted, but they mainly focus on using a single FPGA, and thus how to implement and optimize high-performance and energy-efficient computation units with OpenCL capabilities. Although several studies have used multiple FPGAs connected by high-speed interconnect [16–18], all have focused on building a huge pipelined computation unit across FPGAs and have not assumed OpenCL utilization. Our proposed approach is drastically different in terms of the execution model. For example, although our proposed approach supports only point-to-point data movement, performing large-scale parallel distributed processing on multiple FPGAs is possible if MPI-like libraries to enable collective communication are implemented. Of course, building a huge pipelined execution model is also feasible. In addition, our proposed approach uses a commodity FPGA board. This means that if we provide the modified BSP, every programmer can easily test and use the same features. This also represents one of the big differences of our approach compared to the previous studies.

6 CONCLUSION

We proposed a high-performance FPGA-to-FPGA Ethernet communication through a QSFP+ optical interconnect with mixed programming based on OpenCL and Verilog HDL, and confirmed that data movement was performed successfully. We utilized OpenCL and Verilog HDL to program application algorithms and data movement control, and to implement the Ethernet IP controller as the interface between the Ethernet IP core and kernel code, respectively. Experimental results using ping-pong programs showed that the latency was 0.99 μ s and the effective bandwidth was as much as 4.97 GB/s, thus achieving 99.4 % of the theoretical peak performance. Hardware resource utilization was approximately 12 %, and we clarified that plenty of resources were available to implement application-specific pipelined computation units. The results suggest that our proposed approach is a promising means to realize low-latency communication-enhanced parallel processing running on multiple FPGAs connected by a high-speed interconnect.

Because of time constraints, our approach does not currently support any communication protocol above the data link layer. This means that performing retransmission and flow controls when transmission errors occur is impossible. In addition, although our approach can transfer data through an Ethernet switch, only point-to-point data movement is currently supported as of this writing. Therefore, dealing with these issues and performing an evaluation that uses more practical applications such as the Himeno benchmark are planned for a future work. Furthermore, as long as sufficient FPGA resources exist, we can design and implement as large a multifunctional kernel as possible, which allows multiple FPGAs to perform computation and communication autonomously, thus accelerating the reconfigurable HPC to offer strong scalability.

ACKNOWLEDGMENTS

A part of this research is supported by the Japan Science and Technology Agency’s (JST) CREST program entitled “Research and Development of Unified Environment on Accelerated Computing and Interconnection for Post-Petascale Era.” We also thank Intel FPGA University Program for providing us with both hardware and software.

REFERENCES

- [1] T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Interconnection network for tightly coupled accelerators architecture. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 79–82, Aug 2013.
- [2] T. Kuhara, C. Tsuruta, T. Hanawa, and H. Amano. Reduction calculator in an fpga based switching hub for high performance clusters. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept 2015.
- [3] Chiharu Tsuruta, Yohei Miki, Takuya Kuhara, Hideharu Amano, and Masayuki Umemura. Off-loading let generation to peach2: A switching hub for high performance gpu clusters. *SIGARCH Comput. Archit. News*, Vol. 43, No. 4, pp. 3–8, April 2016.
- [4] Intel FPGA SDK for OpenCL. <https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html>.
- [5] Xilinx SDAccel Development Environment. <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
- [6] OpenCL Overview. <https://www.khronos.org/opencl/>.
- [7] Qsys - System Integration Tool. <https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html>.
- [8] Terasic DE1-SoC Development Kit. <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836>.
- [9] Low Latency 40- and 100-Gbps Ethernet MAC and PHY MegaCore Function. <https://www.altera.com/documentation/n1k141172688901.html>.

- [10] Transceiver Toolkit. <https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/swf-transceiver-toolkit.html>.
- [11] Bittware A10PL4. <http://www.alteraboards.com/product/a10pl4/>.
- [12] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, pp. 35:1–35:12, Piscataway, NJ, USA, 2016. IEEE Press.
- [13] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy efficient scientific computing on fpgas using opencl. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17*, pp. 247–256, New York, NY, USA, 2017. ACM.
- [14] S. Lee, J. Kim, and J. S. Vetter. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554, May 2016.
- [15] Seyong Lee and Jeffrey S. Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14*, pp. 115–120, New York, NY, USA, 2014. ACM.
- [16] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecture, ISCA '14*, pp. 13–24, Piscataway, NJ, USA, 2014. IEEE Press.
- [17] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 25, No. 3, pp. 695–705, March 2014.
- [18] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. Network-attached fpgas for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 36–43, Dec 2016.