OpenCL-enabled High Performance Direct Memory Access for GPU-FPGA Cooperative Computation

Ryohei Kobayashi

Center for Computational Sciences

University of Tsukuba

1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577 Japan

kobayashi@cs.tsukuba.ac.jp

Yoshiki Yamaguchi

Faculty of Engineering Information and Systems

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

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 a high-performance GPU-FPGA data communication using OpenCL and Verilog HDL mixed programming in order to make both devices smoothly work together. OpenCL is used to program application algorithms and data movement control when Verilog HDL is used to implement low-level components for memory copies between the two devices. Experimental results using toy programs showed that our proposed method achieves a latency of $0.6 \mu\text{s}$ and as much as 6.9 GB/s between the GPU and the FPGA, thus confirming that the proposed method is effective at realizing the high-performance GPU-FPGA cooperative computation.

KEYWORDS

GPU, FPGA, OpenCL, BSP, PCIe, DMA, I/O channels, Verilog HDL, Heterogeneous platform, Accelerator computing, Interconnect for accelerators

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. We call this concept Accelerator-in-Switch (AiS) and Figure 1 illustrates what it 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 make all devices, particularly GPUs and FPGAs, work together and to control that operation. In this paper, we focus on GPU-FPGA data

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).

HPCA Asia'19 Workshops, January 14–16, 2019, Guangzhou, China

© 2019 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6236-8/19/01.

<https://doi.org/10.1145/3317576.3317581>

Figure 1: Overview of the AiS

Figure 2: Overview of Intel FPGA SDK for OpenCL

movement approaches for the issue and propose a method for performing high performance direct memory access (DMA) between the two devices. The DMA feature is implemented in an FPGA using a PCIe intellectual property (IP) core and can be controlled using OpenCL code. We evaluate our proposed method in terms of communication latency and bandwidth and as a result, its latency and bandwidth are up to 33.3x and 2.0x better than a classical method to pass the data through CPU memory.

Our contributions in this paper are threefold:

- We propose a DMA method for GPU-FPGA cooperative computation and show that the data movement can be controlled from the FPGA side with OpenCL code.
- We detail our implementation with OpenCL and Verilog HDL mixed programming for realizing the proposed method.
- We measure communication latency and bandwidth and show that our proposed method strongly outperforms a classical one that performs data movement via CPU memory when transferring small size of data.

2 OPENCL-ENABLED GPU-FPGA DMA

2.1 Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL programming model is shown in Figure 2 (a). The host code is 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) and 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 for designing computation parts offloaded to the FPGA and 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 aocxfile that includes FPGA configuration information. The aocxfile 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.

Figure 3: Overview of performing GPU-FPGA DMA and how to control it from OpenCL kernel

Figure 2 (b) 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 board support package (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. 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. In other words, programmers essentially do not have to be concerned about anything other than the host and kernel code implementation and it is possible to port existing OpenCL kernel code for an FPGA board to any other board, as long as its BSP is available.

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. So far, we have implemented a quad small form-factor pluggable plus (QSFP+) controller that can interpret the Ethernet protocol, integrated it into a BSP, and shown that the network controller can be controlled from OpenCL kernel code using an *I/O Channel API* [4]. In this paper, we modify a PCIe controller in a BSP so that an FPGA can access GPU global memory directly through the PCIe bus, and similar to [4], we control the DMA feature from OpenCL kernel code using the *I/O Channel API*.

2.2 How to Control GPU-FPGA DMA from OpenCL Kernel Code using I/O Channels

Figure 3 (a) shows our proposed GPU-FPGA DMA method controlled from OpenCL kernel. Mapping GPU global memory and FPGA external memory to PCIe address space, the DMA controller in the PCIe IP core performs memory copies between both devices. This feature is almost same as a technique proposed in [1], but our

proposed method allows the FPGA to **autonomously perform** the DMA transfer. Our proposed GPU-FPGA DMA method is performed as following procedures. CPU-side procedures are (1) to map GPU global memory to PCIe address space and (2) to send PCIe address mapping information of GPU global memory to the FPGA. FPGA-side procedures are (3) to generate the descriptor [5] based on the PCIe address mapping information and to pass it to the descriptor controller through an I/O channel, (4) to write the descriptor to the DMA controller while preventing any other device (such as CPU) from accessing the controller, (5) to perform GPU-FPGA DMA transfer, (6) to receive the DMA completion notification, and (7) to get the completion notification through an I/O channel.

To map GPU global memory to PCIe address space so that the FPGA can access the global memory through the DMA controller in the PCIe IP core, we use a set of APIs (GPU Direct for RDMA) offered from NVIDIA that is also used in the technique proposed in [1]. The technique gets the global memory address mapped to PCIe address space with the NVIDIA Kernel API and we also use this feature to reduce implementation cost. The mapping information is sent to the FPGA at OpenCL initialization (setting the mapping information to an OpenCL kernel argument).

The PCIe controller in the BSP uses an IP core “Arria 10 Hard IP for PCI Express Avalon-MM with DMA” and this IP core can be controlled by writing the descriptor [5]. It contains source and destination addresses, transfer data size in dwords, and DMA descriptor ID that are necessary information for the DMA transfer. Therefore, by setting the PCIe address mapping information of GPU global memory to source/destination addresses in the descriptor, the FPGA can perform data movement from the GPU to the FPGA or from the FPGA to the GPU.

To write the descriptor to the DMA controller in the PCIe IP core from OpenCL kernel, modifying the descriptor controller and implementing the interface between OpenCL kernel and the descriptor controller are necessary so that the descriptor controller can receive the descriptor and send it to the DMA controller while performing the exclusive access control. To realize that, we add Verilog HDL hardware components surrounded in the red dotted line shown in Figure 3 (b). Because the operating frequencies between OpenCL kernel domain and PCIe domain are different, asynchronous (dual-clocked) FIFOs are required to send the descriptor from OpenCL kernel successfully. After passed through the asynchronous FIFO, the descriptor is dequeued to the DMA controller by a scheduler implemented in the descriptor controller at an appropriate timing. And finally, board_spec.xml has to be modified similar to [4] and the DMA feature can be controlled from OpenCL kernel code using the I/O channel API (the write_channel_intel function).

3 EVALUATION

3.1 Experimental Settings

Figure 4 (a) shows our experimental machine configuration. This is a heterogeneous platform composed of three kinds of devices: two Intel Xeon E5-2660 v4 CPUs, two NVIDIA P100 GPUs, and a single BittWare A10PL4 FPGA board [6] connected to the CPU through a PCIe Gen3 x8 interface. In this evaluation, we use a single GPU and FPGA surrounded by the red line shown in Figure 4 (a).

Hardware specification	
CPU	Intel Xeon E5-2660 v4 x2
Host Memory	DDR4-2400 16GB x4
GPU	NVIDIA Tesla P100 x2 (PCIe Gen3 x16)
FPGA	Intel Arria 10 GX (BittWare A10PL4) (PCIe Gen3 x8)
Software specification	
OS	CentOS 7.3
Host Compiler	gcc 4.8.5, g++ 4.8.5
GPU Compiler	CUDA 8.1.8
FPGA Compiler	Intel Quartus Prime Pro Version 17.1.2.304

(a) Our tested specification

(b) Two communication paths for GPU-FPGA communication on our platform: the classical method (left) and our proposed method (right)

Figure 4: Our experimental environment

Table 1: Communication latency comparison between the classical and proposed methods

FPGA ← GPU comm.	
Classical	17 μ s
Proposed	1.44 μ s
FPGA → GPU comm.	
Classical	20 μ s
Proposed	0.6 μ s

Figure 4 (b) shows two communication paths for data movement between the GPU and the FPGA. The left figure shows how to perform data movement between both devices without our proposed method. To do that, CPU-FPGA and CPU-GPU communications must be separately performed in a classical manner and this evaluation uses the OpenCL APIs and cudaMemcpy for the former and latter, respectively. This entire communication time is measured with high_resolution_clock function implemented in the C++ chrono library. On the other hand, our proposed method is that the FPGA autonomously performs memory copies between the GPU and the FPGA, which is realized by generating the descriptor in OpenCL kernel code using PCIe address mapping information of GPU global memory and writing it to the DMA controller in the PCIe IP core. In this evaluation, we implemented an OpenCL helper function to get elapsed cycles for the DMA transfer, which includes generating the descriptor and writing it to the DMA controller. Please note that elapsed time for getting and sending (the blue dotted line) the PCIe address mapping information of GPU global memory by the host is not included in this evaluation.

3.2 Latency

Table 1 shows the communication latency comparison between the two methods for GPU-FPGA data movement when transferring 4-byte data, which is the minimum data size that the DMA controller in the PCIe IP core can handle. In the data movement from the GPU to the FPGA, the proposed method achieved 1.44 μ s, whereas the classical one did 17 μ s. From the FPGA to the GPU, the former was 0.6 μ s, whereas the latter was 20 μ s. Therefore, the proposed method is 11.8x and 33.3x better when performing the

Figure 5: Communication bandwidth comparison between the classical and proposed methods

data movement from the GPU to FPGA and the data movement from the FPGA to GPU, respectively. This is because the proposed method allows the FPGA to access the GPU global memory directly thanks to mapping the GPU memory to PCIe address space and getting the address information with the NVIDIA Kernel API, while the classical one has to use CPU memory and perform the data movement in a store-and-forward manner and not pipelined. According to these results, we clarified that our proposed method can realize low-latency data movement between the GPU and the FPGA.

3.3 Bandwidth

Figure 5 shows the communication bandwidth results for various data sizes with respect to the two methods. As our proposed method can perform the low-latency data movement described in the previous section, the maximum effective bandwidth was achieved at the earlier phase compared to the classical one. In the data movement from the GPU to the FPGA, the maximum was 4.1 GB/s, which is 51.3 % of the theoretical peak performance, which is 8 GB/s because the PCIe interconnect to the FPGA is the narrowest. From the FPGA to the GPU, our proposed method achieved up to 6.9 GB/s that is 86.3 % of the theoretical peak performance. The reason why the efficiency of the data movement from the GPU to the FPGA is lower than the opposite communication is to perform twice communications, which means that the FPGA sends memory requests to the GPU and then the GPU’s DMA controller sends back data to the FPGA. Furthermore, the DMA controller in PCIe IP core can transfer up to $1024 \times 1024 - 4$ bytes per one descriptor, which means that in order to transfer more than the data size, the DMA transfer has to be performed iteratively by sending the new descriptor to the descriptor controller from OpenCL kernel. This overhead affect both of the effective bandwidths when transferring more than the data size, particularly the GPU-to-FPGA communication.

On the other hand, the effective bandwidth of the data movement from the GPU to the FPGA by the classical method was as much as 4.3 GB/s, and the effective bandwidth of the opposite direction was 4.2 GB/s. This is because the data movement was performed in a store-and-forward manner as previously described. In the data movement from the GPU to the FPGA, a memory copy from the GPU to its host was performed at first. After the entire

data was received, the data movement from the CPU to the FPGA was performed. As a result, the theoretical peak performance of the data movement by the classical method can be given as $\frac{N}{\frac{N}{8 \text{ GB/s}} + \frac{N}{16 \text{ GB/s}}} = 5.33 \text{ GB/s}$ where N represents the data size and both bandwidths of 8 and 16 GB/s derive from the theoretical peak performance of PCIe Gen3 x8 and PCIe Gen3 x16, respectively. Therefore, the maximum effective bandwidth of the data movement from the GPU to the FPGA is 80.7 % of the theoretical peak performance and the opposite direction achieved up to 78.8 %.

According to these results, in both directions, our proposed method outperforms the classical one when transferring data less than 4 MiB, which means that cooperative computation based on short messages is suitable to our proposed method. In the case of transferring more than 4 MiB, the FPGA-to-GPU data movement performed by our proposed method still achieved better performance and when transferring 2 GiB, the effective bandwidth of our proposed method was 6.4 GB/s that is 1.5x better than the classical one. Currently, we have discussed some workarounds to avoid the GPU-to-FPGA communication performance degradation.

4 CONCLUSION

We proposed a high-performance GPU-FPGA DMA method for making both devices work together, and confirmed that data movement can be controlled from FPGA side with OpenCL kernel code. The proposed method is realized by writing the descriptor to the DMA controller in the PCIe IP core, and the descriptor is generated in OpenCL kernel code using PCIe address mapping information of GPU global memory. Experimental results using toy programs showed that the latency was as less as $0.6 \mu\text{s}$ and the effective bandwidth was as much as 6.9 GB/s, thus achieving 86.3 % of the theoretical peak performance. The results suggest that our proposed method is a promising means to realize high-performance GPU-FPGA cooperative computation.

ACKNOWLEDGMENTS

This research is a part of the project titled “Development of Computing-Communication Unified Supercomputer in Next Generation” under the program of “Research and Development for Next-Generation Supercomputing Technology” by MEXT. We thank Intel University Program for providing us both of 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] Ryohei Kobayashi, Yuma Oobata, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. Opencil-ready high speed fpga network for reconfigurable high performance computing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018*, pp. 192–201, New York, NY, USA, 2018. ACM.
- [5] Arria 10 Hard IP for PCI Express Avalon-MM with DMA. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_a10_pcie_avmm_dma.pdf.
- [6] Bittware A10PL4. <http://www.alteraboards.com/product/a10pl4/>.