Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Multi-Hybrid Accelerated Simulation by GPU and FPGA on Radiative Transfer Simulation in Astrophysics

Ryohei Kobayashi[1,2,a)] Norihisa Fujita[1,2] Yoshiki Yamaguchi[2,1] Taisuke Boku[1,2]
Kohji Yoshikawa[1,3] Makito Abe[1] Masayuki Umemura[1,3]

Received: March 31, 2020, Accepted: August 10, 2020

Abstract: Field-programmable gate arrays (FPGAs) have garnered significant interest in research on high-performance computing because their computation and communication capabilities have drastically improved in recent years due to advances in semiconductor integration technologies that rely on Moore's Law. In addition to improving FPGA performance, toolchains for the development of FPGAs in OpenCL have been developed and offered by FPGA vendors that reduce the programming effort required. These improvements reveal the possibility of implementing 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 key to improving the performance of heterogeneous supercomputers using accelerators such as the GPU. In this paper, we propose a GPU–FPGA-accelerated simulation based on the concept and show our implementation with CUDA and OpenCL mixed programming for the proposed method. The results of experiments show that our proposed method can always achieve a better performance than GPU-based implementation and we believe that realizing GPU–FPGA-accelerated simulation is the most significant difference between our work and previous studies.

Keywords: GPU, FPGA, CUDA, OpenCL, heterogeneous platform

1. Introduction

Graphics processing units (GPUs) offer good peak performance and high memory bandwidth. They have been widely used in high-performance computing (HPC) systems as accelerators. However, enabling the execution of parallel applications on such heterogeneous clusters requires inter-accelerator communication between nodes. This means that maintaining multiple copies of memory is required; this results in increased latency and severely degraded application performance, particularly when short messages are involved. Moreover, while the GPU has the above beneficial characteristics, it is not effective as an accelerator in applications that employ complicated algorithms using exceptions, non-single instruction multiple data streams (SIMD), and partially poor parallelism.

Field-programmable gate arrays (FPGAs) have emerged in research on high-performance computing (HPC), and several studies have been reported in the past several years. In Ref. [1], the authors proposed a PCI express (PCIe)-based interconnect for accelerators that can reduce accelerator to accelerator communication latency 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 Refs. [2], [3] implemented application-specific computational logic on FPGAs that enabled the on-the-fly offloading of certain specific computational loads to FPGAs while performing cross node data transfers; a significant improvement in performance was achieved. We think that this implementation of low-latency communication-enhanced parallel processing running on multiple FPGAs connected by a high-speed interconnect is crucial to further improving the performance of modern HPC systems that use accelerators. We call this concept Accelerator-in-Switch (AiS) as shown in **Fig. 1**. 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 in areas where CPUs/GPUs are inefficient.

One reason to need such a GPU–FPGA coupling is to accelerate multiphysics applications. Multiphysics is defined as the coupled processes or systems involving more than one simultaneously occurring physical fields and the studies of and knowledge about these processes and systems [4]. Therefore, multiphysics applications perform simulations with multiple interacting physical properties and there are various computations within a simulation. Because of that, accelerating simulation speed by GPU only is quite difficult and this is why we try to combine GPU and FPGA and make the FPGA cover GPU-non suited computation. In this paper, we focus on a radiative transfer simulation code that is based on two types of radiation transfer: the radiation transfer from spot light and the radiation transfer from spatially

[1] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] Degree Programs in Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[3] Degree Programs in Pure and Applied Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] kobayashi@cs.tsukuba.ac.jp

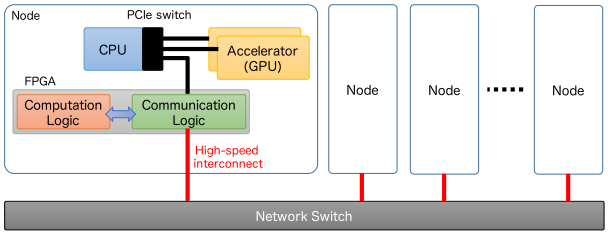


Fig. 1 Overview of the AiS.

distributed light. We make GPUs and FPGAs work together, and perform the former radiation transfer on the GPU and the latter radiation transfer on the FPGA. As a result, we realized GPU–FPGA-accelerated simulation of which the performance was as much as 17.4× higher than GPU-based implementation and was still 1.32× higher even when solving problems of the largest size, which is the fastest problem size for GPU-based implementation.

Our contributions in this paper are:

- We propose how to accelerate a multiphysics application with the GPU and the FPGA. In other words, we analyze the characteristics of the target application qualitatively and quantitatively, and present a methodology to show what kind of computation should be offloaded to the GPU and the FPGA, taking into account the characteristics of each computing device.
- We detail our implementation with CUDA and OpenCL mixed programming for the proposed method.
- We analyze the application code by performing experimental evaluations in detail. As a result, we are able to quantitatively derive in which cases our proposed method of combining the GPU and the FPGA is most effective.

This paper is organized as follows. We describe our target application code accelerated by the AiS concept in Section 2 and show how to use the FPGA for the target application in Section 3. The GPU–FPGA cooperative computation for our target application is shown in Section 4. In Section 5, we perform experimental evaluations. We introduce several previous studies in Section 6 and show how to extend the proposed method to multiple computation nodes in Section 7. And finally, this paper is concluded in Section 8.

2. ARGOT: Radiative Transfer Simulation Code for Astrophysics

ARGOT is an astrophysics simulation code developed in our organization. As shown in **Fig. 2** (a), it combines two algorithms to solve radiative transfer problems: the ARGOT algorithm [5], which computes the radiative transfer from point sources, and the ART algorithm [6], which computes the radiative transfer from sources spreading out in the target space. To accelerate the ARGOT code, we make the ARGOT algorithm run on GPUs and make the ART algorithm run on FPGAs separately, as shown in Fig. 2 (b). We give a brief description of the two algorithms in the next section.

2.1 ARGOT Algorithm

To solve the radiative transfer from point radiation sources, computation of the optical depth between each pair of a point

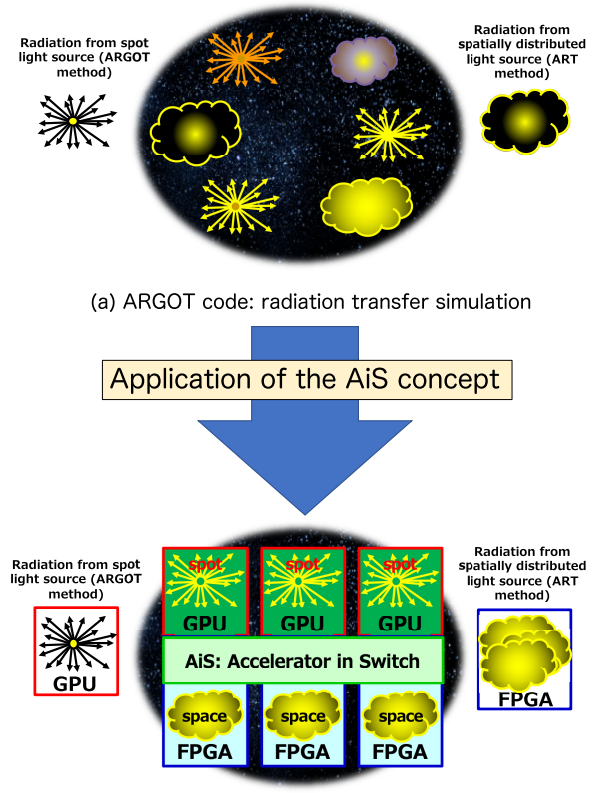


Fig. 2 (a) Overview of the ARGOT code and (b) how to accelerate it by the AiS concept.

radiation source and a target mesh grid, i.e., an end point of each light-ray (see **Fig. 3** (a)), is necessary. Assuming that the number of mesh grids is constant, the computational complexity is proportional to the number of point radiation sources. To address this, the ARGOT algorithm builds an oct-tree data structure for the distribution of radiation sources, as shown in Fig. 3 (b). A cubic computational domain is hierarchically subdivided into eight cubic cells until each cell contains only one radiation source or the size of a cell becomes sufficiently small compared to that of the computational domain. In other words, sources in a distant tree node can be treated as a single luminous source and the effective number of point radiation sources is reduced from N to $logN$. When targeting a mesh grid, e.g., a target mesh grid in Fig. 3 (b), photon flux coming from each radiation source at target mesh grids is given by

$$f(\nu) = \frac{L(\nu)e^{-\tau(\nu)}}{4\pi r^2} \tag{1}$$

where $L(\nu)$ and $\tau(\nu)$ stand for is the intrinsic luminosity and the optical depth for a given frequency ν, respectively. $\tau(\nu)$ is given by

$$\tau(\nu) = \sigma(\nu) \int n(\boldsymbol{x})dl \simeq \sigma(\nu) \sum_i n(\boldsymbol{x}_i)\Delta l \tag{2}$$

where $n(x)$ is the number density of gas molecules that absorb light.

Figure 3 (c) shows how to parallelize the ARGOT algorithm. This parallelized technique is called “Node Parallelization”. It decomposes the simulation volume evenly along each direction.

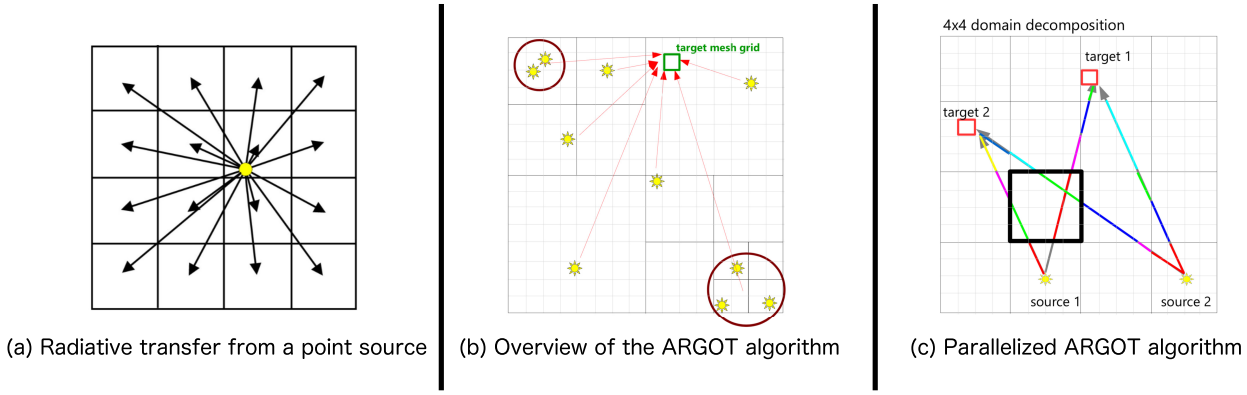


Fig. 3 (a) Schematic illustration of the ray-tracing method for the radiation emitted by a point radiation source in the two-dimensional mesh grids, (b) how to solve it with the ARGOT algorithm, and (c) parallelized ARGOT algorithm.

Light rays are divided by boundaries of parallel domains into several "ray segments" and computation of optical depths of assigned ray segments in each parallel domain is performed. In GPU simulations, this computation is performed on GPUs concurrently. After that, sum reduction of optical depths of each ray segment to their target mesh grids is performed. However, in this paper, the ARGOT code runs on a single node and we do not use this parallelization technique. In that case, the number of ray segments is treated as the number of light rays. The radiative transfer for each ray is assigned to each CUDA thread of the GPU, and then each computation is performed in parallel.

2.2 ART Algorithm

To solve radiation transfer from spatially diffuse sources, the ART algorithm is used that is based on a ray-tracing method in a 3D space split into meshes. The computation part of the ART algorithm accounts for more than 90% of the ARGOT code, and this is why accelerating the ART algorithm directly results in the performance improvement of the ARGOT code. As shown in **Fig. 4**, multiple incident rays come from a boundary and move in a straight direction parallel with each other, without any reflection or refraction. The ART algorithm solves a radiation transfer equation along parallel light-rays starting from one edge to another of computational volume, using the following equation.

$$I_\nu^{out}(\hat{\boldsymbol{n}}) = I_\nu^{in}(\hat{\boldsymbol{n}})e^{-\Delta\tau_\nu} + S_\nu(1 - e^{-\Delta\tau_\nu}) \tag{3}$$

This calculation is performed every time the ray is passed through a mesh grid. For a given incoming radiation intensity I_ν^{in} along a direction $\hat{\boldsymbol{n}}$, the outgoing radiation intensity I_ν^{out} after getting through a path length ΔL of a single mesh is computed by the above integrating equation, where $\Delta\tau$ is the optical depth of the path length ΔL (i.e., $\Delta\tau = \kappa_\nu \Delta\tau$), and S_ν and κ_ν are the source function and the absorption coefficient of the mesh grid, respectively. The direction (angle) of the ray is computed using the HEALPix algorithm [7]. The number of meshes depends on the configuration of the target problem. There will be between 100^3 and 1000^3 meshes in our target problems. The number of ray angles also depends on the problem size. It will be at least 768, where resolution parameter $N_{side} = 8$ in the HEALPix.

Because the ART method uses ray tracing, computational order within a ray must be sequential, whereas computations for different rays can be performed in parallel because no two rays are computationally dependent on each other. However, implementing the ART method on SIMD-like architecture is problematic in two ways.

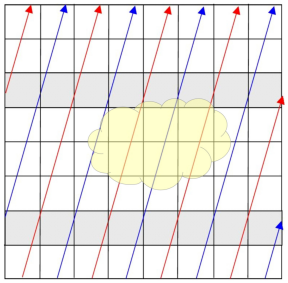


Fig. 4 Ray tracing method used in the ART method. Arrows and yellow cloud show rays and gas to compute reactions, respectively.

First, because the memory access pattern of the mesh data varies depending on the ray direction, hundreds or thousands of different patterns are possible. In some cases, the computation of multiple ray interactions in the SIMD manner requires the mesh data to be accessed in non-continuous locations in memory, which causes a low cache hit ratio on the CPU and long latency in the GPU.

Second, the integration of mesh data resulting from two rays being close to each other would cause conflict. When multiple light rays pass through shaded mesh grids, as shown in Fig. 4, the physical quantities in those mesh grids must be incremented in an atomic manner. However, the atomic operation itself has a overhead. If a large number of threads perform atomic operations at the same time, many contentions may occur and the processing speed may be significantly slowed down. The number of atomic operations is cubically proportional to the size N of one side of the mesh, that is $O(N^3)$. To avoid this atomic operation, the method proposed in Ref. [6] is to not compute the neighboring rays at the same time, which means that ray tracing along the red and blue light rays is separately performed as shown in Fig. 4. However, this method further exacerbates the memory access problems in the ART algorithm described in the first reason because this method causes the memory access patterns to become more scattered. And, this overhead is expected to be close to cubically

proportional as well as the number of atomic operations.

Given this ART algorithm's characteristics, we consider that SIMD-style processors such as CPUs and GPUs are unsuitable for this algorithm. On the other hand, FPGAs can access their on-chip internal memory with low latency and high bandwidth for random accesses. In addition to its performance, we can program memory access patterns as a part of the FPGA hardware. Therefore, we consider that the use of the ART method on an FPGA is suitable. There is a previous study of implementing an FPGA-based hardware engine with OpenCL programming framework [8] and we integrate the engine to the ARGOT code. The implementation of Ref. [8] is described in the next section.

Please note that the ART algorithm is based on a raytracing one, but it is essentially different from that of computer graphics (CG) that can be accelerated by Turing architecture-based GPUs. The CG's raytracing retroactively calculates the light reflection and transmission on the object surface from the observer's viewpoint. On the other hand, the ART algorithm calculates radiation intensity every time the ray is passed through a mesh grid and takes an average intensity by calculating radiation intensity on each ray direction. In short, their common point is only the phrase "raytracing".

3. ART on FPGA Implementation

3.1 Intel FPGA SDK for OpenCL

3.1.1 Overview

For the proposed method of Ref. [8], an OpenCL-based FPGA development toolchain is used, the Intel FPGA SDK for OpenCL; **Fig. 5** shows its programming model. The host code is for programming the host application; it runs on a host PC and manages an FPGA device at runtime with a set of common application programming interfaces (APIs). The code is compiled using a standard C compiler such as GCC on Linux or Visual Studio C/C++ on Windows to generate a host binary. The kernel code is for designing units of computation that are offloaded to the FPGA; it 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 during execution of the host application by using the APIs; any data required for kernel execution as well as any data generated are transferred via the PCIe bus.

Figure 6 shows a schematic of the Intel FPGA SDK for an OpenCL platform. As described above, 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. 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.

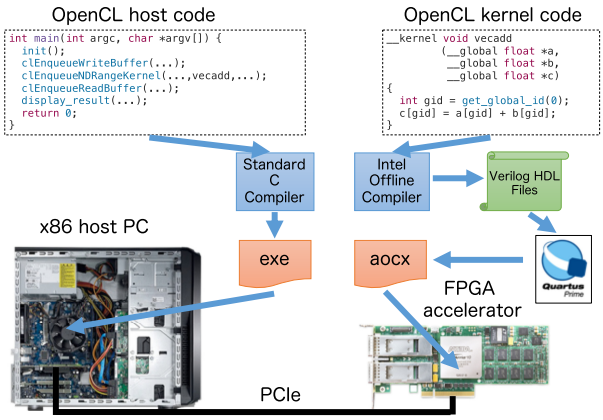


Fig. 5 Intel FPGA SDK for OpenCL programming model.

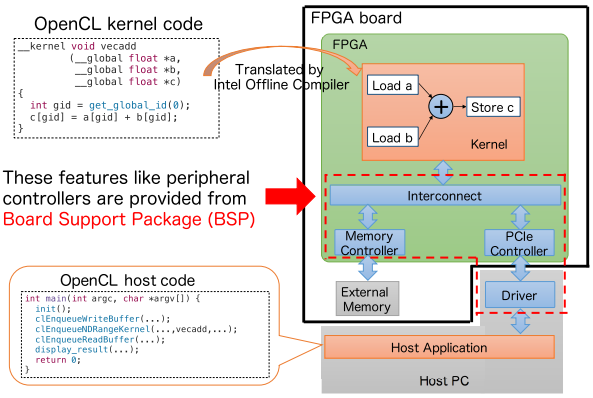


Fig. 6 Schematic of the Intel FPGA SDK for OpenCL platform.

3.1.2 Inter Kernel Communication Using Channel

"Channel" is one of the extensions of the Intel SDK for the OpenCL language. It makes it possible to exchange data between two kernels without any external memory access. A channel can directly connect two kernels with an optional First-In-First-Out (FIFO) buffer.

When two kernels are connected through a channel, the data can be transferred between them without reading or writing on external memory. Instead of that, they are transferred through the buffer inside an FPGA chip to reduce the latency and increase the bandwidth.

3.1.3 Launching Kernels Automatically Using Autorun Attribute

The "autorun" attribute is another extension for the OpenCL language. In the standard OpenCL environment, OpenCL kernels must be managed by the host and invoked explicitly. If a kernel has an autorun attribute, it will be started automatically after the FPGA becomes ready without any interaction from the host.

Autorun kernels are commonly used in combination with the channels described in the previous subsection. If a kernel uses no global memory access and uses channels only as its input or output, we can make it as an autorun kernel. This programming model is similar to a daemon in a general operating system, where a daemon is started automatically in the background and uses network sockets to do its work.

In general, if an FPGA design consists of multiple kernels connected to each other by channels, we have to start or finish a large number of kernels even for a stream of computation. Each API

call to manage the execution of a kernel has control overhead, including PCIe communication overhead. Therefore, it is important for us to keep the overhead small by using the autorun attribute. Because autorun kernels are controlled inside the FPGA, they have low control overhead and low resource usage on the FPGA. No connection between the host and the FPGA is required for them.

3.2 Implementation Overview

The basic strategy of the ART on FPGA implementation is to allocate multiple computation kernels into an FPGA and connect them using channels. Each kernel computes the reaction between a mesh and a ray in its own computation space. During this computation, a ray traverses multiple computational kernels and takes different meshes in the space depending on its location. If a ray leaves a kernel's space, its data will be transferred to the kernel for a neighboring mesh through a channel.

Figure 7 shows how the implementation is. The "memory reader" reads the mesh and ray data from the DDR memory, which is seen as a global memory in the OpenCL environment. The "memory writer" is the counterpart to the reader. It writes ray data to the memory and updates mesh data based on the computational results. Because the ART algorithm computes the integration of a gas reaction, both read and write memory accesses are required for the mesh data. The "buffer" is a mesh data buffer used to improve the memory access performance. The "PE array" is an array of Processing Elements (PEs). A PE computes a kernel using the ART algorithm (see Equation (3)), and the array consists of multiple kernels. All computations use a single precision floating point and are implemented by Digital Signal Processors (DSPs) in an FPGA including the exponent functions.

In our implementation, the "buffers" and "PE array" kernels shown in Fig. 7 are marked as "autorun" kernels. The remaining kernels are regular kernels and controlled by the host. Because of their global memory access, making them autorun kernels is impossible. Applying the autorun attribute to kernels reduces the number of kernels managed by the host. As a result, the control overhead is decreased, and the total computational performance is increased.

3.3 Parallelization Using Channel in FPGA

In this section, we describe the structure of the "PE array" shown in Fig. 7. The "PE array" is a group of OpenCL kernels that implement the ART algorithm and is composed of the Processing Element (PE) and Boundary Element (BE), as shown in **Fig. 8**. For running the ART method on the FPGA, the PEs and BEs mutually communicate the ray data with each other via channels.

The PE is an arithmetic kernel for the ART algorithm. In our implementation, the problem space for an FPGA is divided into small blocks and each block is assigned to each PE. The data for computation is stored in Block Random Access Memories (BRAMs) because of the need for high frequency random access, and each PE has its own BRAM for the computation. BRAMs are memories (generally SRAMs) that are implemented inside the FPGA and are distributed in blocks of a certain size on the

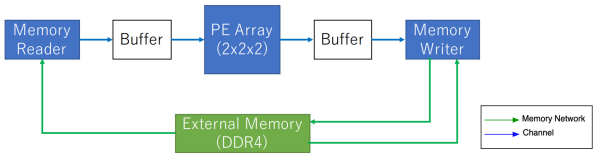


Fig. 7 Overview of the ART on FPGA implementation.

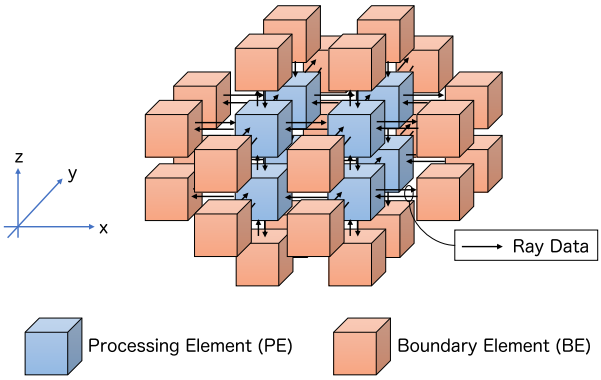


Fig. 8 Structure of the "PE array".

FPGA chip. They have powerful random access capability with low latency and high bandwidth, but their capacity is lower than that of external memory, such as the DDR memory. Therefore, it is essential to use BRAM in conjunction with the DDR memory when running the ART algorithm for a problem size that does not fit within the FPGA.

The BE is a kernel that performs the input and output processing of the ray required by the PE, which means that the BE is responsible for the boundary processing of ray tracing. Therefore, the BE performs the initial generation of the ray, the discarding of unneeded rays, the reading of the rays generated by past computation from a ray buffer, and the writing of the rays to the ray buffer for the computation in the next time step. It is essential to use the DDR memory for the process of using the ray buffer as well. We describe how the DDR memory is used for the ART algorithm on the FPGA in the next section.

As described in the beginning of this section, the FPGA implementation developed in this study uses channels for parallel execution of the ART algorithm. This implementation is based on the Multiple Wave Front (MWF) method [9] that is an inter-node parallelization method used in the original CPU/GPU implementation of the ART algorithm, and is refined for the FPGA. Since the cost of channel communication is lower than inter-node communication on CPUs and GPUs, channel accesses are built into the computation pipeline for finer-grained communication. Inside the FPGA, the problem space handled by a single FPGA is divided into several small spaces and the processing for the problem space is also divided among the PEs. In other words, the MPI process in the CPU's MWF implementation corresponds to the PE.

A PE is connected with a neighboring PE in the x, y and z dimensions. There are two PEs in the x-, y-, and z-dimensions, i.e., $2 \times 2 \times 2 = 8$ PEs implemented in the FPGA. The number of the PEs depends on hardware resources of the FPGA and we have taken the maximum number of PEs that can be implemented in the FPGA. A BE is connected with an adjacent PE,

as it is responsible for the boundary processing. The connection of a BE differs from the connection of a PE, which means that A BE is connected with only one adjacent PE and not an adjacent BE. The two channels are used for the ray data communication in both directions. The bit width of the channel is determined by the size of the structure that the ART algorithm uses to represent the ray.

Channel accesses are embedded in the computation pipeline, and one element of data (the channel bit-width's data) is communicated per clock cycle if both the sender and receiver operate without pipeline stalls. Moreover, when there are multiple channels in the FPGA, each channel can operate and be responsible for the ray data communication independently. Therefore, the cost of communication using channels in FPGAs is lower than that of inter-node communication of the CPU/GPU implementation.

3.4 DDR Memory Access of the FPGA-based ART Algorithm

As described in the previous section, the problem size that can be solved by the FPGA is limited by the capacity of the BRAM. We believe that we need to be able to allocate at least 128^3 problem space per FPGA to compute practical problems in the ARGOT code. While 128^3 meshes require 192 MB of memory, and current state-of-the-art FPGAs have BRAM capacities of at most 20~30 MB, which means that it is essential to use DDR memory in conjunction with the BRAM to solve larger problems. The data used for the ART algorithm is stored in DDR memory, and the BRAM is treated such as a scratchpad cache. In other words, a large problem stored in the DDR memory is divided into small blocks that can be stored in the BRAM, and the FPGA performs the computation of the ART in block-wise and time-division-multiplexing manner.

The pseudo code of the DDR implementation is shown in Algorithm 1. Please note that this shows the flow of DDR memory accesses and does not reflect the actual implementation. `ray_directions[]` is an array of positive and negative ray directions for the X, Y and Z axes, `small_blocks[]` is an array of the small blocks to be computed, `ipix_list(dir)` is a function that returns angles pointing towards `dir` from angles computed using the Healpix library. In the actual implementation, the algorithm is divided into multiple kernels and they are connected using channels, as described in the previous section.

To use DDR memory, there are two key OpenCL kernels; one is to replace the mesh data along with the computation progresses, and the other is about the ray data. The mesh data refers to the region that stores the source function parameters (absorption and diffusion of photons) in Eq. (3) and the intermediate results of the integration, which requires $4 \times sizeof(float) \times \nu$ bytes. The ray data contains the radiation intensity and occupies $sizeof(float) \times \nu$ bytes. As previously described in Section 2.2, the ART algorithm solves the radiation transfer by dividing the three-dimensional space into meshes, and therefore, the amount of memory used for mesh data is cubically proportional to the problem size. Also, since the number of rays is squarely proportional to the mesh size of the problem's boundary plane, the maximum amount of memory required is $12 \times N_{side}^2 \times N^2$ bytes (N_{side} is resolution parameter in the Healpix. $12 \times N_{side}^2$ ray angles are generated).

Algorithm 1 Pseudo code of the DDR implementation

```
1:  for dir in ray_directions[] do
2:    for b in small_blocks[] do
3:      mesh_load(b)
4:      for ipix in ipix_list(dir) do
5:        for iray in rays(ipix) do
6:          r = ray_load(iray)
7:          for m in compute_path(r) do
8:            compute reaction between r and m
9:          end for
10:         ray_store(iray, r)
11:       end for
12:     end for
13:     mesh_store(b)
14:   end for
15: end for
```

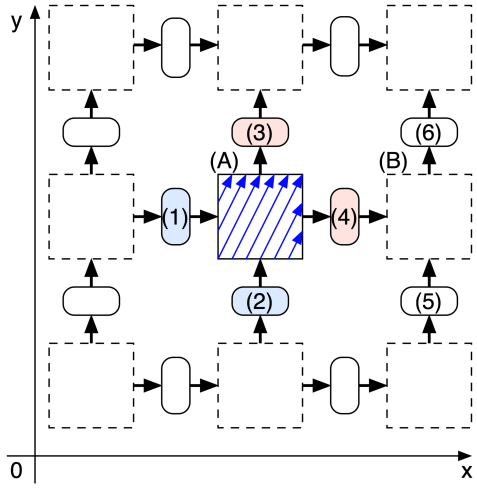


Fig. 9 Overview of the ray buffer. Red boxes, blue boxes and blue arrows represent output ray buffers, input ray buffers, and rays to compute, respectively.

For DDR memory access to mesh data, the BRAM for mesh data in the PE is used as a cache. The mesh data is copied from DDR memory to BRAM (Line 3 in Algorithm 1), the computation of the ART algorithm is performed on BRAM (Line 8 in Algorithm 1, no DDR memory access at this point), and then the computation results are written back from the BRAM to the DDR memory (Line 13 in Algorithm 1). Since the DDR memory access order of the mesh data is fixed and known beforehand, the next required data can be pre-fetched from the DDR memory into the FIFO buffer during the computation. As a result, minimizing the computation pipeline stall can be achieved.

DDR memory access for ray data is more complicated than that for mesh data. Since the amount of memory required for the ray buffer is large compared to the size of the BRAM, the ray buffer must be allocated on the DDR memory. **Figure 9** shows how the ray buffer is used for the ART algorithm. This figure is simplified to 2D space, but there is a similar structure in the Z-dimension because the actual computation deals with 3D space. Also, This figure shows only the case where the ray is computed along the X+ and Y+ axes. However, since the ray is input at various angles, there are eight directions in total. The eight di-

rections mean that there are positive and negative directions in the X, Y and Z axes, respectively. Therefore, there are 8 combinations of data flows in the actual computation as well. In this figure, there are nine boxes and each of them represent mesh data. The solid black box (A) in the center represents the block that is currently being computed (variable b in Algorithm 1). The ray data used in the calculation of block A are loaded from the two blue input ray buffers (numbers 1 and 2) and the results are stored in the two red output ray buffers (numbers 3 and 4). The input and output ray buffers vary according to the position of the mesh to be computed, e.g., when computing block B, the 4th and 5th are the input buffers and the 6th is the output buffer. In addition, there is no output buffer to the right of block B because block B is located at the edge of the computational domain, and if a ray appears outside the computational domain, it is discarded.

4. GPU–FPGA-accelerated ARGOT code

4.1 Overview

The implementation overview of the ARGOT code with the GPU and the FPGA is shown in **Fig. 10**. The GPU implementation of the ARGOT code that performs both the algorithms on the GPU has already been implemented. Based on the GPU implementation of the ARGOT code, we replace the GPU-based ART algorithm with the ART on the FPGA implementation. To do that, data transfer between the GPU and the FPGA has to be performed appropriately. The initial data of the ART algorithm is generated on the GPU and it is sent to the FPGA, and then the result data of the ART is sent back to the GPU. Here, the initial and result data of the ART refer to the mesh data representing the initial state before ray tracing and the mesh data reflecting the effects of ray tracing, respectively. Normally, the data transfer between the GPU and the FPGA is performed through the CPU memory, but such an indirect data movement is obviously inefficient. To address that issue, a method for high performance direct memory access (DMA) between the two devices has been proposed [10]. The DMA feature is implemented in an FPGA using a PCIe intellectual property (IP) core and can be controlled using OpenCL code.

The ARGOT code was implemented in multilingual programming composed of CUDA and OpenCL, and therefore a separate compilation was needed. **Figure 11** shows the flow of the compilation. The CUDA code and OpenCL host code were compiled with `nvcc` and `g++` separately, and the generated object files were linked using `nvcc` to generate an executable and linkable-format (ELF) file. As previously described in Section 3.1, the OpenCL kernel code for the ART algorithm was compiled offline using the Intel FPGA OpenCL compiler.

4.2 OpenCL-enabled GPU–FPGA DMA

As previously described in Section 3.1, BSPs support external hardware component access from an FPGA (OpenCL kernel code). However, they provide a minimum set of peripheral controllers to enable OpenCL programming (i.e., PCIe and external memory controllers). We modify a PCIe controller into a BSP so that an FPGA can access the GPU's global memory directly through the PCIe bus. In other words, realizing OpenCL-enabled

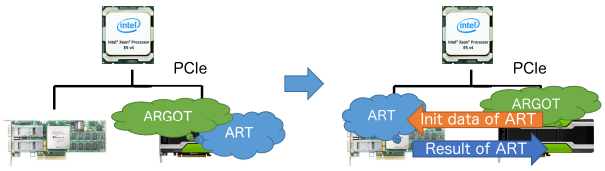


Fig. 10 The implementation overview of GPU–FPGA-accelerated ARGOT code.

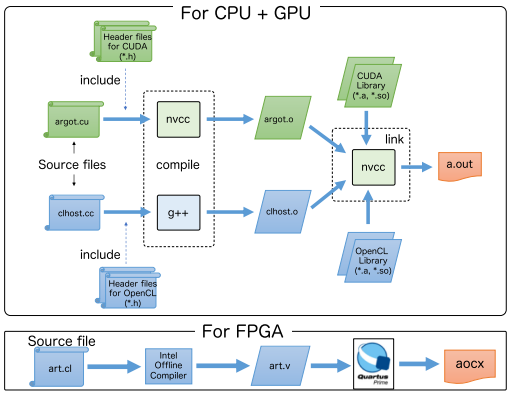


Fig. 11 The compilation flow of GPU–FPGA-accelerated ARGOT code.

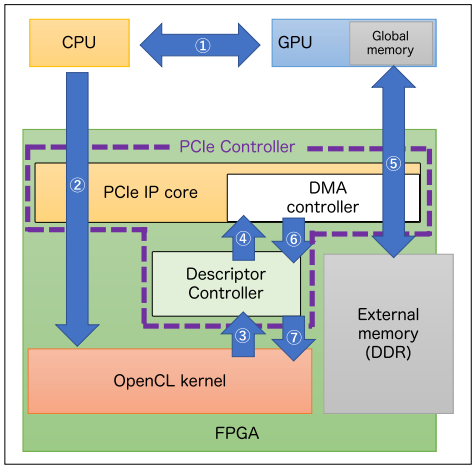


Fig. 12 Overview of GPU–FPGA DMA.

GPU–FPGA DMA is essentially synonymous with modifying the BSP.

Figure 12 shows our developed GPU–FPGA DMA method controlled by the OpenCL kernel. By mapping the GPU's global memory and external FPGA memory to PCIe address space, the DMA controller in the PCIe IP core performs memory copies between the devices. This feature is almost identical to a technique proposed in Ref. [1], but our developed method allows the FPGA to **autonomously perform** the DMA transfer without support from the CPU. Our developed GPU–FPGA DMA method is performed using the following procedures.

- The procedures at the CPU
  - (1) Mapping GPU's global memory to the PCIe address space
  - (2) Sending PCIe address mapping information of the GPU's global memory to the FPGA
- The procedures at the FPGA
  - (3) Generating the descriptor [11] based on the PCIe ad-

dress mapping information and passing it to the descriptor controller through an I/O channel

(4) Writing the descriptor to the DMA controller while preventing any other device (such as the CPU) from accessing the controller

(5) Performing GPU–FPGA DMA transfer

(6) Receiving the DMA completion notification

(7) Obtaining the completion notification through an I/O channel

Procedures at the CPU are performed **only once** because the address information of GPU's global memory mapped to PCIe address space is stored and reused in the FPGA.

4.2.1 PCIe Address Mapping

To map GPU's global memory to PCIe address space so that the FPGA can access it through the DMA controller in the PCIe IP core, we use a set of APIs (GPU Direct for RDMA) offered in NVIDIA that was also used in the technique proposed in Ref. [1]. The technique has the global memory address mapped to PCIe address space with the NVIDIA Kernel API, and we use this feature to reduce the cost of implementation.

Figure 13 shows how to map GPU's global memory to PCIe address space and obtain the memory addresses. By passing the pointer `ptr` to the `tcaCreateHandleGPU()` function that is an API of the technique, the GPU's global memory is mapped to PCIe address space, and memory address information is stored in the variable `paddr` with a set of NVIDIA APIs. The mapping information is sent to the FPGA upon the initialization of OpenCL (setting the mapping information to an OpenCL kernel argument).

4.2.2 DMA Descriptor

Our target FPGA board for the implementation of the developed method was a PCIe-based FPGA board with Intel Arria 10 FPGA whose BSP was ready to use. The PCIe controller in the BSP used an IP core "Arria 10 Hard IP for PCI Express Avalon-MM with DMA" that can be controlled by writing the descriptor [11]. **Table 1** shows the format of the descriptor, and contains the source and destination addresses, the size of transfer data in **dwords**, and the DMA descriptor ID. Note that the size of transfer data is encoded in 18 bits, and therefore, the DMA controller in the IP core can transfer up to 1,024 × 1,024 − 4 bytes per descriptor.

For example, by setting the mapping information of the PCIe addresses of GPU's global memory to the source/destination addresses in the descriptor, the FPGA can move data from the GPU to the FPGA or the other way round. As described in the previous section, the FPGA (OpenCL kernel) receives the address information of GPU's global memory mapped to PCIe address space from the CPU and generates a descriptor with the address. Following this, the descriptor is sent to the descriptor controller through an I/O channel.

4.2.3 Descriptor Controller

The descriptor controller is a hardware module to write the descriptor to the DMA controller in the PCIe IP core, which is implemented in the PCIe controller provided by the BSP. Our developed method needs to control the descriptor controller from OpenCL kernel, but this module is instantiated in the PCIe IP core

```
1  #define SIZE  1000000
2
3  tcaresult tcaCreateHandleGPU(unsigned long long *paddr,
       void *ptr, size_t size);
4
5  int main(void) {
6    uint32_t data[SIZE/4];
7    void* ptr;
8    cudaSetDevice(0);
9    cudaMalloc(&ptr, SIZE);
10
11   unsigned long long paddr;
12   tcaCreateHandleGPU(&paddr, ptr, SIZE);
13
14   printf("paddr = 0x%016llx¥n", paddr);
15
16  return 0;
17 }
```

Fig. 13 Mapping GPU's global memory to PCIe address space. Line 12: The `tcaCreateHandleGPU()` function performs the mapping and stores the memory address information in `paddr`.

Table 1 DMA descriptor format.

Bits	Name
[31:0]	Source Low Address
[63:32]	Source High Address
[95:64]	Destination Low Address
[127:96]	Destination High Address
[145:128]	DMA Length
[153:146]	DMA Descriptor ID
[158:154]	Reserved
[159]	Immediate Write

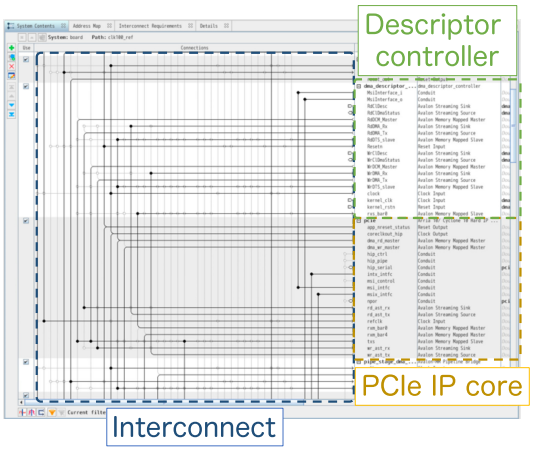


Fig. 14 External instantiation of the descriptor controller.

and only CPU can access it by default. Therefore, the descriptor controller must be externally instantiated so that OpenCL kernel can access it as well. **Figure 14** shows what it looks like. Necessary connections between the descriptor controller and the PCIe IP core is done with the GUI tool Platform Designer and OpenCL kernel can access the descriptor controller by adding Avalon-ST interface used for I/O channels to the externally instantiated module.

Figure 15 shows a schematic of the descriptor controller. Our developed method performs GPU–FPGA DMA transfer by sending a descriptor generated in the OpenCL kernel to this hardware module through an I/O channel and writing the descriptor to the DMA controller. To realize this, exclusive access control is necessary because the CPU also manipulates this module for CPU–FPGA DMA transfer using OpenCL APIs such as `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`.

As shown in Fig. 15, the descriptor controller is composed of

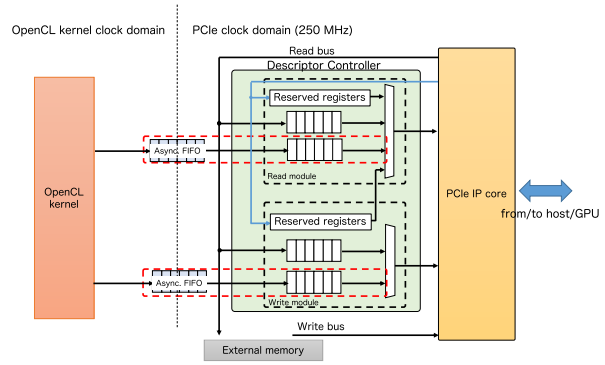


Fig. 15 Schematic of the descriptor controller. The hardware components of Verilog HDL surrounded by the red dotted line are added to enable the descriptor controller to write the descriptor sent from OpenCL kernel to the DMA controller in the PCIe IP core.

a write module and a read module, and each contains registers reserved for the CPU and a FIFO buffer to store a descriptor for DMA transfer between the CPU and the FPGA. The CPU first manipulates a reserved register of the read or write module with programmable IO (PIO) access to assemble a descriptor to fetch the CPU-to-FPGA or FPGA-to-CPU DMA descriptor from the host (system memory). The read module writes the assembled descriptor to the DMA controller in the PCIe IP core, and the CPU–FPGA DMA descriptor is fetched and forwarded to the read or write module's FIFO buffer. Finally, CPU–FPGA DMA transfer is performed by dequeuing the descriptor from an FIFO buffer to the DMA controller. In case of CPU-to-FPGA DMA transfer, the FPGA receives data from the CPU and forwards them to its external memory through the read bus, and sends data from the external memory to the CPU through the write bus if the FPGA-to-CPU DMA transfer is performed.

To perform GPU–FPGA DMA transfer without disturbing the operation, we add the hardware components of Verilog HDL, surrounded by the red dotted line shown in Fig. 15. Descriptors sent from the OpenCL kernel with an I/O channel are stored in the read or write module's FIFO buffer and dequeued to the DMA controller by a scheduler implemented in the descriptor controller at an appropriate time. Because the operating frequencies between the domains of the OpenCL kernel and PCIe are different, asynchronous (dual-clock) FIFOs are required to send the descriptor successfully from the OpenCL kernel. Similar to CPU–FPGA DMA transfer, the FPGA sends data from the external memory through the write bus or forwards data to it through the read bus.

4.2.4 Example of OpenCL code

Figure 16 shows an example of OpenCL kernel code to perform DMA transfer from the GPU to the FPGA. As described in the previous section, the descriptor controller is manipulated through I/O channels and therefore, the pragma `#pragma OPENCL EXTENSION cl_intel_channels: enable` has to be written in OpenCL kernel code at first.

The DMA descriptor is generated in OpenCL kernel code and, based on Table 1, a struct data structure for the descriptor is declared from lines 3 to 10. The size of the descriptor is 160 bits but we set it to 256 bits because channel width that is not a power of two currently does not work in this toolchain. Therefore, a 256-bit data structure is generated and passed through I/O channels,

```
1  #pragma OPENCL EXTENSION cl_intel_channels : enable
2
3  typedef struct __attribute__((packed)) cldesc {
4    ulong src;
5    ulong dst;
6    uint  id_and_len;
7    uint  unused0;
8    uint  unused1;
9    uint  unused2;
10 } cldesc_t;
11
12 channel cldesc_t fpga_dma __attribute__((depth(0)))
       __attribute__((io("chan_fpga_dma")));
13 channel ulong dma_stat __attribute__((depth(0)))
       __attribute__((io("chan_dma_stat")));
14
15 __kernel void fpga_dma(__global float *restrict fpga_mem,
16                         const ulong gpu_memadr,
17                         const uint id_and_len)
18 {
19   cldesc_t desc;
20   // DMA transfer GPU -> FPGA
21   desc.src = gpu_memadr;
22   desc.dst = (ulong)(&fpga_mem[0]);
23   desc.id_and_len = id_and_len;
24   write_channel_intel(fpga_dma, desc);
25   ulong status = read_channel_intel(dma_stat);
26 }
```

Fig. 16 OpenCL kernel code to perform GPU-to-FPGA DMA transfer.

and the descriptor's 160 bits are retrieved in the Verilog HDL layer.

An I/O channel variable is used to define connectivity between kernels and the I/O. It needs to be declared with an `io` attribute in the OpenCL kernel code, which is shown in lines 12 and 13. The io("chan_fpga_dma") and io("chan_dma_stat") attributes specify paths to transfer the descriptor and the DMA completion notification, respectively, and the value used in the attribute must be the chan_id of the I/O interface listed in the board_spec.xml file. The `depth` attribute is set to zero because FIFO buffers required to perform GPU–FPGA DMA transfer have already been implemented using Verilog HDL.

Finally, all necessary data for GPU-to-FPGA DMA transfer are set from lines 21 to 23, and the generated descriptor is sent to the descriptor controller using the `write_channel_intel()` function. Following this, DMA transfer from the GPU to the FPGA is performed and the DMA controller in the PCIe IP core notifies the descriptor controller of the completion of the data transfer. The completion notification is stored in a FIFO buffer implemented in the descriptor controller and OpenCL kernel fetches it using the `read_channel_intel()` function.

Please note that the DMA controller in PCIe IP core can transfer up to $1{,}024 \times 1{,}024 - 4$ bytes per descriptor, which means that in order to transfer more than the data size, the DMA transfer has to be performed iteratively by sending a new descriptor to the descriptor controller from OpenCL kernel. The new descriptor is generated with recalculation of source and destination addresses and the GPU global memory address is reused for that calculation as a base address. Therefore, there is no need to resend the GPU memory address from the CPU again.

4.3 ART on FPGA with OpenCL-enabled GPU–FPGA DMA

As shown in Fig. 7, the data flow of the pipelined hardware is Memory Reader → PE Array → Memory Writer, which means that the ART execution starts from loading initialized data on the external memory by the Memory Reader and is done by storing

```
1  __kernel void MemoryReader (
2      __global art_in volatile* restrict mem,
3      // init data of the ART is stored here
4      const ulong gpu_memadr
5  ) {
6      // Recv init data from GPU
7      dma_gpu_to_fpga(gpu_memadr, mem, data_size);
8
9      // ~ send data to the next stage of the pipeline ~
10     ...
11 }
12
13 __kernel void MemoryWriter (
14     __global art_out volatile* restrict mem,
15     // rslt data of the ART is written here
16     const ulong gpu_memadr
17 ) {
18     // ~ get rslt data generated at PE Array ~
19     ...
20
21     // Send rslt data to GPU
22     dma_fpga_to_gpu(mem, gpu_memadr, data_size);
23 }
```

Fig. 17 An OpenCL kernel code snippet of the ART on FPGA with GPU–FPGA DMA.

its execution result by Memory Writer. As previously described in Section 4.1, the initialized data is located on the GPU memory and the ART execution result is sent back to the GPU. Therefore, by making the Memory Reader and the Memory Writer access the GPU memory directly, more efficient FPGA offloading can be realized and our developed GPU–FPGA DMA feature enables it.

Figure 17 shows an OpenCL kernel code snippet of the ART on FPGA with GPU–FPGA DMA. As previously described in Section 3.2, the Memory Reader and Memory Writer are implemented as OpenCL kernels. The function `dma_gpu_to_fpga()` in the Memory Reader is an OpenCL helper function to perform GPU-to-FPGA DMA transfer as shown in Fig. 16 and requires the address information of GPU's global memory mapped to PCIe address space, a external memory pointer as the DMA transfer destination, and the size of transfer data. The GPU's global memory address is set as a kernel argument at line 4 and the CPU obtains this address information at first by using the API shown in Fig. 13 and sends it to the FPGA. By performing this function, the initialized data is transferred from the GPU to the FPGA directly and is stored in the memory location pointed to by `mem`.

In the Memory Writer kernel, the ART result data is transferred to the GPU memory by the `dma_fpga_to_gpu()` function. The GPU's global memory address is set as a kernel argument at line 16 and this is used as a destination address. By performing this function, the ART execution result located on the external memory pointed to by `mem` is sent back to the GPU, and then the FPGA terminates the ART execution.

5. Evaluation

5.1 Experimental Settings

Figure 18 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 for PCIe-based servers (Gen3 x16), and a single BittWare A10PL4 FPGA board [12] connected to the CPU through a PCIe Gen3 x8 interface. In this evaluation, we used a single GPU and an FPGA as surrounded by the red line shown in Fig. 18 in order

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 9.1.85
FPGA Compiler	Intel Quartus Prime Pro Version 17.1.2.304

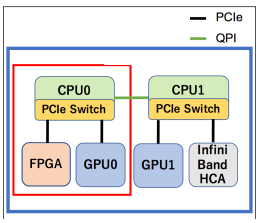


Fig. 18 Our experimental environment.

to avoid the performance degradation caused by a PCIe access over a Quick Path Interconnect (QPI).

The operating system of our experimental machine was CentOS 7.3 and the GPU–FPGA-accelerated ARGOT code was separately compiled using `nvcc` and `g++`, as shown in the previous section. The CUDA version was 9.1.85 and the GCC version was 4.8.5, respectively. The ART algorithm working on the FPGA was implemented in OpenCL kernel code and was compiled with the offline compiler provided by the Intel FPGA SDK for OpenCL whose version was 17.1.2.304 Pro edition.

The problem size used for the evaluation ranged from 16^3 to 128^3. The ART on FPGA implementation used a design with 8 PEs (2^3), and each PE had BRAMs for an 8^3 meshes. Therefore, 16^3 meshes are stored in an FPGA in each step. N_{side}, which is a parameter used to determine the resolution in the HEALpix, is set at 8, which generates 768 of different angles of rays (768 is the number of angles, not the number of rays for the ray tracing). In this evaluation, the computation time on a CPU is measured and included the cost of launching and synchronizing the device, both for FPGA and GPU implementations. And, the time for data transfer between the host and the devices is also included.

5.2 Resource Consumption

Table 2 shows a comparison of FPGA resource usage. 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). ALM utilization is a metric used to estimate the size of the area of the hardware components implemented in the FPGA. The M20K memory block is an internal memory of the FPGA that is called a Block RAM, and internal buffers such as FIFOs are implemented using memory blocks. The digital signal processor (DSP) is a built-in hardware component that is faster and offers more compact implementations of floating-point operations than programmable logic components. "Freq." means the operating frequency in the clock domain for OpenCL kernels.

As shown in Table 2, resource consumption differences between designs with and without the GPU–FPGA DMA are quite negligible. The frequency is slightly dropped, but is acceptable for performing the ART algorithm. Verilog HDL codes generated by the OpenCL compiler are not human readable, and it is nearly impossible to understand how OpenCL kernels are implemented as circuits.

There is a large difference in resource usage between the 16^3 implementation and the 32^3 implementation and this is because the DDR control kernels have to be implemented for solving more than 32^3 problem sizes. They are not necessary in the 16^3 imple-

Table 2 Resource usage and clock frequency of the ART on FPGA implementation.

Mesh size	# of PEs	DMA	ALMs	(%)	Registers	(%)	M20K	(%)	DSP	(%)	Freq. [MHz]
(16, 16, 16)	(2, 2, 2)	w/o DMA	160,308	38%	312,477	18%	1,104	41%	752	50%	210.4
		w/ DMA	163,203	38%	315,692	18%	1,116	41%	752	50%	198.3
(32, 32, 32)	(2, 2, 2)	w/o DMA	191,477	45%	368,778	22%	1,179	43%	752	50%	184.4
		w/ DMA	195,447	46%	373,350	22%	1,197	44%	772	51%	181.7
(64, 64, 64)	(2, 2, 2)	w/o DMA	191,120	45%	368,875	22%	1,180	43%	752	50%	182.5
		w/ DMA	195,576	46%	373,546	22%	1,198	44%	772	51%	181.5
(128, 128, 128)	(2, 2, 2)	w/o DMA	191,001	45%	368,997	22%	1,180	43%	752	50%	185.7
		w/ DMA	195,378	46%	373,517	22%	1,198	44%	772	51%	182.5

Table 3 DDR memory utilization.

Mesh size	Allocated for the mesh data	Ray buffer
(16, 16, 16)	0.375 MB	0 MB
(32, 32, 32)	3 MB	264 MB
(64, 64, 64)	24 MB	1,128 MB
(128, 128, 128)	192 MB	4,584 MB

mentation because the problem size is small, and all of the meshes can be stored in an FPGA's BRAM.

As listed in the table, the DSP is the greatest resource user in this design, with the half of the total DSPs used, which becomes a bottleneck when attempting to increase performance. However, an Intel Stratix 10 FPGA has more than 3x DSPs compared to the Intel Arria 10 FPGA used for this evaluation, and there is room for further performance improvement. In addition, if we optimize the OpenCL code to decrease the resource usage in the design, implementing more PEs into the FPGA is possible and this directly leads to the reduction of the execution time. And in general, as resource usage is reduced, the operating frequency becomes higher because place-and-routing is easy to be performed. This also improves the ART performance.

Table 3 shows DDR memory utilization. As described above, 16^3 problem size is small and all of the meshes can be stored in an FPGA's BRAM, and this is why the ray buffer is not necessary because it is used when the FPGA computes a large problem that cannot be stored in the BRAM in block-wise and time-division-multiplexing manner described in Section 3. In this evaluation, we used the A10PL4 FPGA board and the amount of memory available for the board is 8 GB. According to Table 3, memory usage for the ray buffer is dominant and is squarely proportional to the mesh size of the problem's boundary plane. Therefore, this evaluation took up to 128^3 problem size because the FPGA-based ART algorithm for solving 256^3 problem size requires four times as much memory as 128^3 problem size, which exceeds the amount of memory available for the A10PL4.

5.3 Performance Evaluation of the ARGOT code

Figure 19, **Fig. 20**, **Fig. 21**, and **Fig. 22** show the performance comparison between the CPU, GPU, and FPGA implementations, depending on the problem size. These results are execution time per simulation step. ARGOT(CPU) / ART (CPU) represents that both the algorithms are the CPU implementation, and the remaining graph items represent each implementation in the same manner. The CPU implementation is written in C and uses OpenMP for the thread parallelization. In this evaluation, we use a single Xeon CPU and the CPU implementation is performed with 14 OpenMP cores (threads). The GPU implementation is based on the CPU implementation but written in CUDA.

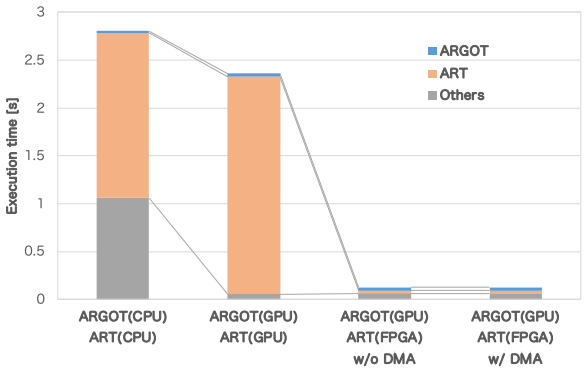


Fig. 19 Performance comparison between FPGA, CPU, and GPU implementations. The problem size is 16^3.

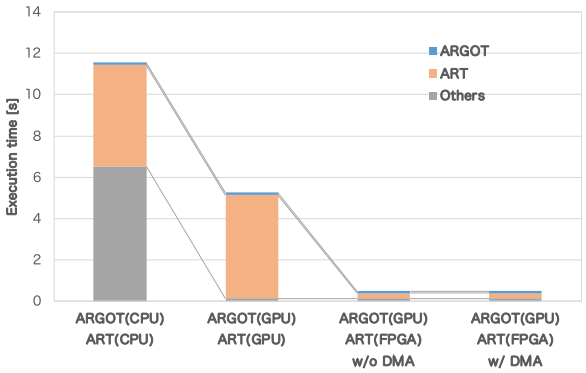


Fig. 20 Performance comparison between FPGA, CPU, and GPU implementations. The problem size is 32^3.

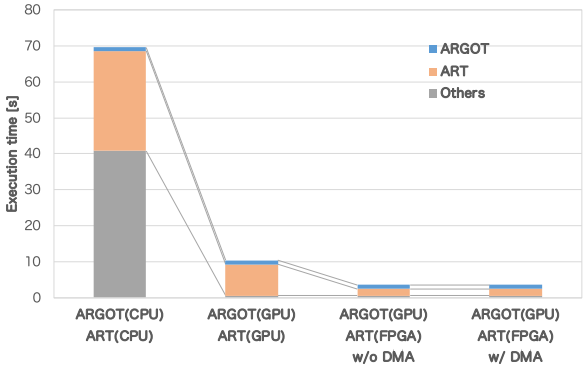


Fig. 21 Performance comparison between FPGA, CPU, and GPU implementations. The problem size is 64^3.

As shown in these figures, not only the ART algorithm but also the "Others" execution are dominant in the CPU implementation. This part mainly solves chemical reactions and radiative heating/-cooling of each mesh, based on the execution results of the ARGOT and ART algorithms. Fortunately, solving chemical reactions and radiative heating/cooling of each mesh is independent

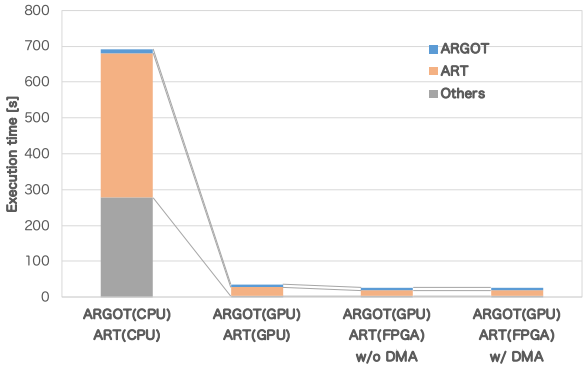


Fig. 22 Performance comparison between FPGA, CPU, and GPU implementations. The problem size is 128^3.

and this is why the GPU implementation can accelerate its execution.

However, the ART algorithm does not benefit from this situation and is still dominant in the ARGOT code even when the GPU is used. Considering the GPU implementations in Fig. 19 and Fig. 20, the execution of the ART algorithm is not accelerated at all. This is because SIMD-style processors, such as CPUs and GPUs, are unsuitable for this algorithm as previously described in Section 2.2. In addition to this, their problem sizes are too small to sufficiently exploit the 3,584 CUDA cores of the GPU because the requisite parallelism is not achievable. On the contrary, the performance is worse compared to the CPU, due to the lower performance of the cores themselves compared to the CPU cores, and due to the overhead of GPU kernel activation and CPU-GPU communication.

On the GPU, the performance of the ART algorithm improves significantly when solving larger sized problems, such as in Fig. 21 and Fig. 22. This is because the parallelism increases on the order of $O(N^2)$ and computational complexity increases on the order of $O(N^3)$, where N is the size of one side of the mesh. As described in Section 2, the ART algorithm is based on a ray-tracing method in a 3D space split into meshes, and this is why the increase of the computational complexity is cubically proportional. And, the ART algorithm solves radiation transfer equation along parallel light-rays starting from one edge to another of computational volume, which means that each CUDA thread is mapped to each ray in two dimensions, and this is why the increase of the parallelism is squarely proportional. In other words, by increasing the problem size, sufficient computational complexity and parallelism begin to appear in the GPU-based ART algorithm, which push away the inherently unsuitable for SIMD-type processors to perform the ART algorithm and the overhead of offloading it to the GPU, and can fully operate all of the CUDA cores. As a result, the performance of the GPU-based ART algorithm becomes better due to these reasons and the performance difference between the GPU and the FPGA becomes smaller. Therefore, the performance of the GPU and the FPGA performing the ART algorithm may be reversed because of the further increase in the computational complexity and parallelism at more than 256^3 problem sizes. However, the experiments to clarify this cannot be conducted due to the maximum amount of memory in the FPGA and this is our future work.

On the other hand, the FPGA-based ART implementations with and without GPU–FPGA DMA are always better than the GPU-based one when solving any problem size. As reported [8], this high performance comes from the pipelined ART algorithm implemented in the FPGA. Both FPGA implementations are 1.6x better even when solving 128^3 problem size that is the fastest problem size for the GPU, and as the problem size becomes smaller, the performance of the ART on the FPGA is much more significant compared to the GPU. Understandably, the problem sizes 16^3 and 32^3 are too small to use in practice. The reason why we measured performance at such small problem sizes is to quantitatively clarify that offloading small-sized ART to the FPGA is much better than the GPU in order to offer strong scalability. If the problem size is fixed and the system size is increased, the problem size of each computation kernel becomes smaller and this is why the obtained result in this research shows good characteristics to realize that offering. Furthermore, the latest FPGAs (Stratix 10) are equipped with multiple high speed optical communication interfaces, such as 400 Gbps Ethernet unlike the GPU. We expect that this strength is also suitable for offering strong scalability and have plans to integrate such communication feature into the ART algorithm in future work.

Some readers may also wonder if it would be better to offload the ARGOT algorithm to an FPGA in addition to the ART algorithm. However, it has not been done for the following two reasons. First, since the ARGOT algorithm is already sufficiently accelerated on GPUs, it is not worth the time and effort to offload such a GPU-friendly algorithm to the FPGA. Second, offloading to an FPGA means consuming hardware resources in the FPGA, which could deplete resources that should be spent on hardware to further accelerate the ART algorithm and implement the communication mechanism described above. In addition, the consumption of hardware resources affects the place-and-route, which may reduce the operating frequency in some cases. For these reasons, it is not practical to offload both the ARGOT and ART algorithms to the FPGA, and therefore, the significance of the proposed method of using the GPU and the FPGA together is unassailable.

5.4 Performance Analysis of the GPU–FPGA DMA

In this section, we analyze how much the GPU–FPGA DMA contributes to the performance improvement of the ARGOT code. To investigate this, we measured execution time for data transfer when offloading the ART algorithm to the FPGA that is included in the execution time of the ART on the FPGA shown in the previous section.

Table 4 shows performance breakdown of the FPGA-based ART of the ARGOT code between with and without GPU–FPGA DMA. For measurement of data transfer performance with and without the DMA, we implemented an OpenCL helper function[*1] to obtain the number of elapsed cycles for the former case and used `gettimeofday` function for the latter case. In Table 4, “Comp.”, “G2F Comm.”, and “F2G Comm.” correspond the exe-

[*1] We could not integrate that function into the DMA-enabled ART algorithm solving 16^3 problem size because of unknown failure of the FPGA toolchain.

Table 4 Performance breakdown of the FPGA-based ART of the ARGOT code between with and without GPU–FPGA DMA.

Mesh size	DMA	Comp. [s]	G2F Comm. [s]	Size [MB]	F2G Comm. [s]	Size [MB]
(32, 32, 32)	w/o DMA	0.240	0.003	1	0.004	2
	w/ DMA	0.241	0.002	1	0.002	2
(64, 64, 64)	w/o DMA	1.920	0.015	8	0.025	16
	w/ DMA	1.923	0.016	8	0.015	16
(128, 128, 128)	w/o DMA	15.13	0.104	64	0.229	128
	w/ DMA	15.28	0.133	64	0.118	128

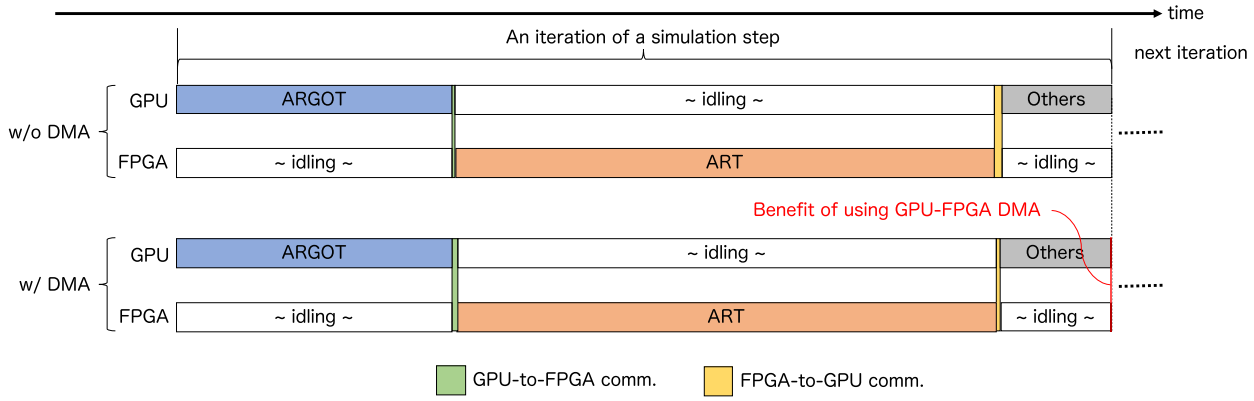


Fig. 24 An ideal timing chart for an iteration within a simulation step for the ARGOT code with the 128^3 problem size running on the GPU and the FPGA.

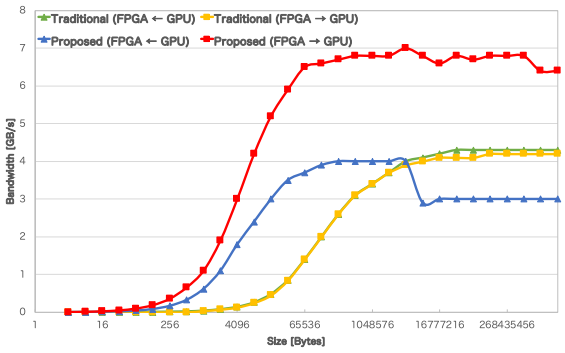


Fig. 23 Comparison of communication bandwidth between the GPU–FPGA DMA and non-DMA (through CPU memory) [10].

cution part of the ART on the FPGA, data transfer for initialized data of the ART, and data transfer for sending the ART execution result back to the GPU, respectively. Therefore, this summation is equal to the execution time of the ART on the FPGA shown in Fig. 20, Fig. 21, and Fig. 22. Please note that a simulation step has two or more ART executions and these measured execution times are the total value within a simulation step.

Taking a look at the Comp. of both cases, the FPGA-based ART without the DMA shows a slightly better performance because of the operating frequency shown in Table 2. In the GPU-to-FPGA data transfer, the FPGA-based ART with the DMA shows a better communication performance when transferring the initialized data for solving 32^3 problem size. However, when solving more than 64^3 problem sizes, its communication performance becomes lower than in the case without the DMA. This is because performance degradation when transferring more than 8 MiB data occurred as reported [10] and its experiment is shown in **Fig. 23**. According to NVIDIA engineers, this is caused by L2 cache of P100 overflows and we are currently investigating this reason in detail. In the opposite data movement, the FPGA-based ART with the DMA always achieves better communication performance that is shown in Fig. 23.

We discuss in detail how the developed GPU–FPGA DMA feature improves the overall performance of the ARGOT code. **Figure 24** shows an ideal timing chart for an iteration within a simulation step for the ARGOT code with the 128^3 problem size running on the GPU and the FPGA. The sequence of processes in the iteration shown in the figure is repeated until the convergence condition is satisfied. Then, when the conditions are met, the next simulation step begins. Therefore, as described above, the execution time of the ARGOT code shown so far is the sum of the execution times for all the iterations. And, "ideal" means that GPU–FPGA communication is 100% of the performance shown in Ref. [10], and using of the GPU–FPGA DMA feature does not cause a decrease in the operating frequency, and the FPGA's computational performance is equivalent to that of the case without the GPU–FPGA DMA feature. In other words, the difference in communication time between with and without the DMA feature in Fig. 24 is the difference in the time required to communicate the amount of data shown in Table 4 with the communication bandwidth reported in Ref. [10].

Based on this timing chart, the analysis of the performance in Fig. 22 and Table 4 shows the same behavior as the timing chart, that is, the ART algorithm is still the dominant computation, accounting for more than 50% of the total, while GPU-FPGA communication accounts for only about 1% of the total processing. However, unlike the ideal timing chart, the actual operating frequency of the FPGA was slightly reduced by the addition of the DMA feature. Therefore, the advantage of using the DMA feature, which was only marginally available in an ideal case, is lost.

This is an analysis for 128^3 problem sizes, but even if the problem size is changed, the processing time is globally cubic proportional and the ratio of the processing time breakdown remains the

same. In terms of communication time, although the smaller the problem size is the more beneficial for the use of the DMA feature, it cannot be the key to a significant performance improvement (or it may be ineffective due to the reduction in the operating frequency) because communication time accounts for only a small percentage of the total processing. Therefore, the bottleneck of the GPU–FPGA-accelerated ARGOT code is still the ART algorithm, and a method to accelerate this part is required.

One way to achieve this is to increase the number of PEs implemented in the FPGA to increase the throughput of the ART algorithm. As previously described in Section 5.2, the latest FPGA, Intel Stratix 10, has three times as many DSPs as the FPGA used in this experiment, and we have been developing a version with twice as many PEs for the Intel Stratix 10. And, the execution time of the ARGOT algorithm and Others can be expected to be shortened by using V100 that has a better processing performance than P100 used in the experiments.

As shown in the timing chart, the ARGOT and ART algorithms are executed in series in the ARGOT code, but this is due to the version of the ARGOT code that we have targeted in this study. This means that since these algorithms are inherently independent, they can be concurrently performed on each device using OpenMP's task-parallel programming. As a result, achieving a higher simulation speed can be expected and this is our future work.

6. Related Work

In recent years, many groups of researchers have been interested in applying FPGAs to HPC applications using OpenCL programming frameworks [13], [14], [15].

In Ref. [13], the authors ported and optimized a subset of the Rodinia benchmark suite to an FPGA platform using Intel FPGA SDK for OpenCL, and compared the performance and energy efficiency of a modern CPU and a GPU. Their evaluation showed that in most benchmarks, only energy efficiency was superior to that of the GPU, whereas both performance and energy efficiency were better than those of the CPU. The authors also proposed a combined spatial and temporal blocking method for stencil computation with OpenCL and built a performance model [14].

Weller et al. [15] proposed a comprehensive set of OpenCL optimization techniques for a partial-differential equation including dataset optimization, algorithmic enhancements, and data and control flow tuning methods that improved performance and energy efficiency by several orders of magnitude. The authors also compared FPGA implementations between Intel and Xilinx, and showed that fundamentally different optimization approaches for Intel and Xilinx are required to make OpenCL code efficient. So far, such a comparative experiment has hardly been conducted because the Xilinx OpenCL-based toolchain has been offered in the last two years or three, and therefore, this research outcome is quite interesting.

Reference [16] compared the performances and resource usages between VHDL and OpenCL with the same algorithm. The performance of the OpenCL implementation was almost equal to that of the VHDL implementation. However, the resource usage of the OpenCL implementation was much larger than that of the VHDL implementation.

Reference [17] compared the irregular memory access performance in XSBench on FPGAs using OpenCL. Intel Arria 10's performance was 35% slower than that of an 8-core Xeon CPU, but the energy efficiency was 50% better than the CPU.

However, these related studies have focused on using only the FPGA, and thus on implementing and optimizing high-performance and energy-efficient computation units with OpenCL capabilities. They have not considered how to enable GPUs and FPGAs to work together. One of the important points to enable that is to understand each device's strength and weakness and to offload appropriate computation to each device. The absolute performance of an FPGA is not comparable with those of other accelerators such as GPUs. Therefore, the type of computation that is offloaded to an FPGA is important.

In this study, we optimized the ART algorithm on an FPGA using OpenCL because its memory access pattern is complicated to SIMD-style processors, and performed the rest of the computation on the GPU. As a result, we achieved performance improvement of the ARGOT code compared to the GPU-based implementation and we believe that realizing GPU–FPGA-accelerated simulation is the most significant difference between our work and previous studies.

7. Next Step: Parallelizing ARGOT Code with Multiple GPUs and FPGAs

As discussed in Section 5, it was clarified that the performance of the FPGA-based ART was significantly better than that of GPUs at small mesh sizes, and slightly better than that of GPUs even at large mesh sizes. Since these results show good properties for offering strong scaling, we plan to add a network feature to our ART on FPGA implementation to further increase the problem size and improve performance.

In the GPU programming model, the GPU is a slave device and has to be controlled by the host CPU, including inter-node communication. Because of that, the CPU and the GPU must be synchronized before the communication starts. When using a combination of NVIDIA GPUs and Mellanox HCAs, the GPUDirect for RDMA (GDR) can be used to improve communication performance. The GDR will surely improve the bandwidth and latency of the communication, and yet non-negligible communication overhead still exists because the communication is initiated by the CPU.

In contrast, modern high-end FPGAs, such as the Intel Stratix 10, include several high-speed communication mechanisms that currently support up to 100 Gbps $\times$ 4 links. Other research institutes have already been working on direct communication with FPGAs without going through the CPU, and the high communication performance of FPGAs has been shown in Ref. [18], [19]. In addition to the above benefit of offloading the ART to the FPGA, small communication overhead of the FPGA is another benefit compared to the GPU because the FPGA is capable of autonomously initiating such the powerful communication feature. Given those advantages, we believe that the effective performance of ART with multiple FPGAs can easily exceed that of ART with multiple GPUs. Therefore, we have been studying a framework

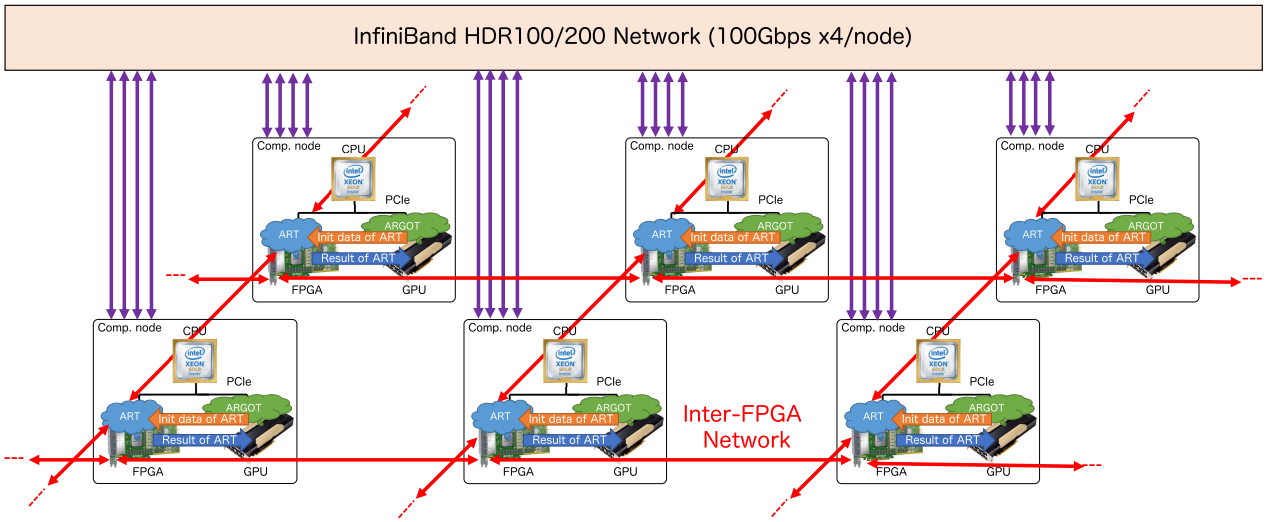


Fig. 25 Implementation plan when extending the proposed method to the Cygnus supercomputer.

for controlling the powerful communication capabilities of FPGAs from the OpenCL environment [20] and will integrate it into our ART on FPGA implementation.

Figure 25 shows our implementation plan when extending the proposed method to the Cygnus supercomputer that is installed into our organization. A computation node of the Cygnus is composed of the CPU, the GPU, and the FPGA as same as our experimental machine, and we will try to run the ART method on the FPGA and the remaining computation part including the ARGOT method on the GPU, respectively. When the ART method is parallelized, the ray data needs to be communicated through the high-speed inter-FPGA direct network, and the BE described in Section 3 will be responsible for this inter-FPGA communication. This is why we have adopted a separate implementation of the PE and the BE in order to facilitate the implementation of a parallel ART algorithm using multiple FPGAs connected the inter-FPGA network. Unlike the ray data, the mesh data is not transferred between FPGAs. The initial and post-computed mesh data of the ART method are then sent and received between the GPU and the FPGA on each node independently by using the GPU-FPGA DMA function described in this paper. And meanwhile, the remaining computation part is parallelized on multiple GPUs the MPI programming model that enables the GDR. For parallelizing the ARGOT algorithm, we will use the "Node Parallelization" technique shown in Fig. 3 (c).

8. Conclusion

In this paper, we focused on accelerating a radiative transfer simulation in astrophysics by combining the GPU and the FPGA. The radiative transfer simulation code is based on two types of radiation transfer: the radiation transfer from spot light and the radiation transfer from spatially distributed light. Given the latter radiation transfer characteristics, we decided to offload its computation to the FPGA while performing the rest of the simulation code on the GPU. As a result, we realized GPU–FPGA-accelerated simulation and its performance was always better than GPU-based implementation. In particular, as the problem size becomes smaller, offloading the ART algorithm to the FPGA becomes the best option to accelerate the ARGOT code. This tendency is suitable for offering strong scalability and our next step is to prove this hypothesis by implementing the ARGOT code running on multi GPUs and FPGAs.

We have also developed the OpenCL-enabled GPU–FPGA DMA feature for making efficient cooperative computation and integrated it into the ARGOT code. The evaluation result showed the communication performance between the GPU and the FPGA when executing the ARGOT code was as reported in our previous study. However, that communication part accounts for only 2% of the ART on the FPGA itself, and therefore, we conclude optimizing the pipelined hardware of the ART by increasing the number of PEs and improving the operating frequency is the most effective way in order to accelerate the ARGOT code further. Currently, we are developing OpenCL-enabled inter-FPGA communication hardware to enable the ART algorithm performed on multiple PEs over FPGAs.

Finally, in order to address how an FPGA efficiently detects whether GPU computation is completed, a sophisticated synchronization mechanism needs to be implemented. Furthermore, we plan to develop a software framework to comprehensively control both of the proposed DMA functionality and such a mechanism from a CPU in order to make cooperative GPU–FPGA computation more practical. These are our future works.

Acknowledgments This work was supported in part by the "Next Generation High-Performance Computing Infrastructures and Applications R&D Program" (Development of Computing Communication Unified Supercomputer in Next Generation) of MEXT. This research was also supported (in part) by Multidisciplinary Cooperative Research Program in CCS, University of Tsukuba, and JSPS KAKENHI Grant Number 18H03246. We also thank the Intel University Program for providing hardware and software.

References

[1] Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: Interconnection Network for Tightly Coupled Accelerators Architecture, *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp.79–82 (online), DOI: 10.1109/HOTI.2013.15 (2013).

[2] Kuhara, T., Tsuruta, C., Hanawa, T. and Amano, H.: Reduction calculator in an FPGA based switching Hub for high performance clusters, *2015 25th International Conference on Field Programmable Logic and Applications* (*FPL*), pp.1–4 (online), DOI: 10.1109/FPL.2015.7293985 (2015).

[3] Tsuruta, C., Miki, Y., Kuhara, T., Amano, H. and Umemura, M.: 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 (online), DOI: 10.1145/2927964.2927966 (2016).

[4] Liu, Z.: *Multiphysics in Porous Materials* (2018).

[5] Okamoto, T., Yoshikawa, K. and Umemura, M.: ARGOT: Accelerated radiative transfer on grids using oct-tree, *Monthly Notices of the Royal Astronomical Society*, Vol.419, No.4, pp.2855–2866 (online), DOI: 10.1111/j.1365-2966.2011.19927.x (2012).

[6] Tanaka, S., Yoshikawa, K., Okamoto, T. and Hasegawa, K.: A new ray-tracing scheme for 3D diffuse radiation transfer on highly parallel architectures, *Publications of the Astronomical Society of Japan*, Vol.67, No.4, p.62 (online), DOI: 10.1093/pasj/psv027 (2015).

[7] Gorski, K.M., Hivon, E., Banday, A.J., Wandelt, B.D., Hansen, F.K., Reinecke, M. and Bartelmann, M.: HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere, *The Astrophysical Journal*, Vol.622, No.2, pp.759–771 (online), DOI: 10.1086/427976 (2005).

[8] Fujita, N., Kobayashi, R., Yamaguchi, Y., Oobata, Y., Boku, T., Abe, M., Yoshikawa, K. and Umemura, M.: Accelerating Space Radiative Transfer on FPGA Using OpenCL, *Proc. 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, *HEART 2018*, pp.6:1–6:7, ACM (online), DOI: 10.1145/3241793.3241799 (2018).

[9] Nakamoto, T., Umemura, M. and Susa, H.: The effects of radiative transfer on the reionization of an inhomogeneous universe, *Monthly Notices of the Royal Astronomical Society*, Vol.321, No.4, pp.593–604 (online), DOI: 10.1046/j.1365-8711.2001.04008.x (2001).

[10] Kobayashi, R., Fujita, N., Yamaguchi, Y., Nakamichi, A. and Boku, T.: GPU-FPGA Heterogeneous Computing with OpenCL-Enabled Direct Memory Access, *2019 IEEE International Parallel and Distributed Processing Symposium Workshops* (*IPDPSW*), pp.489–498 (online), DOI: 10.1109/IPDPSW.2019.00090 (2019).

[11] Arria 10 Hard IP for PCI Express Avalon-MM with DMA, available from ⟨https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_a10_pcie_avmm_dma.pdf⟩.

[12] Bittware A10PL4, available from ⟨http://www.alteraboards.com/product/a10pl4/⟩.

[13] Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M. and Matsuoka, S.: Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, *SC '16*, pp.35:1–35:12, IEEE Press (online), available from ⟨http://dl.acm.org/citation.cfm?id=3014904.3014951⟩ (2016).

[14] Zohouri, H.R., Podobas, A. and Matsuoka, S.: Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL, *Proc. 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '18*, pp.153–162, ACM (online), DOI: 10.1145/3174243.3174248 (2018).

[15] Weller, D., Oboril, F., Lukarski, D., Becker, J. and Tahoori, M.: Energy Efficient Scientific Computing on FPGAs Using OpenCL, *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '17*, pp.247–256, ACM (online), DOI: 10.1145/3020078.3021730 (2017).

[16] Hill, K., Craciun, S., George, A. and Lam, H.: Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA, *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors* (*ASAP*), pp.189–193 (online), DOI: 10.1109/ASAP.2015.7245733 (2015).

[17] Luo, Y., Wen, X., Yoshii, K., Ogrenci-Memik, S., Memik, G., Finkel, H. and Cappello, F.: Evaluating irregular memory access on OpenCL FPGA platforms: A case study with XSBench, *2017 27th International Conference on Field Programmable Logic and Applications* (*FPL*), pp.1–4 (online), DOI: 10.23919/FPL.2017.8056827 (2017).

[18] Weerasinghe, J., Polig, R., Abel, F. and Hagleitner, C.: Network-attached FPGAs for data center applications, *2016 International Conference on Field-Programmable Technology* (*FPT*), pp.36–43 (2016).

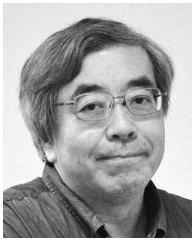
[19] Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y. and Burger, D.: A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services, *Proc. 41st Annual International Symposium on Computer Architecuture*, *ISCA '14*, pp.13–24, IEEE Press (online), available from ⟨http://dl.acm.org/citation.cfm?id=2665671.2665678⟩ (2014).

[20] Fujita, N., Kobayashi, R., Yamaguchi, Y. and Boku, T.: Parallel Processing on FPGA Combining Computation and Communication in OpenCL Programming, *2019 IEEE International Parallel and Distributed Processing Symposium Workshops* (*IPDPSW*), pp.479–488 (online), DOI: 10.1109/IPDPSW.2019.00089 (2019).

Ryohei Kobayashi received his Ph.D. degree from Tokyo Institute of Technology, Japan in 2016. He is an assistant professor of Center for Computational Sciences, University of Tsukuba, Japan. His research interests include GPU–FPGA-accelerated computing and FPGA systems for high performance computing. He is a member of the IPSJ, IEICE, IEEE, and ACM.

Norihisa Fujita received Ph.D. degrees (Doctor of Engineering) from Graduate School of Science and Technology, University of Tsukuba in 2016. From 2016 until 2019, he was a post-doctoral researcher in the center. Since 2019, he is an assistant professor in the center. His research interests are accelerators in high-performance computing and interconnection in HPC systems. He is a member of IPSJ.

Yoshiki Yamaguchi received his M.S. degree in science and engineering and his Ph.D. degree in engineering from University of Tsukuba, in 2000 and 2003, respectively. From 2000 to 2003, he was a JSPS Research Fellow with University of Tsukuba. From 2003 to 2005, he was a Research Scientist with the Yokohama Institute, RIKEN. Since 2005, he has been an Assistant Professor with the Graduate School of System and Information Engineering, University of Tsukuba. He was a Visiting Academic with the Department of Computing, Imperial College London, from 2010 to 2011. Since 2011, he has been a Collaborative Fellow with the Center for Computational Science, University of Tsukuba. His research interests include reconfigurable architecture, real-time applications, and power-efficiency computing for image, sound, and bioinformatics applications. He is also interested in heterogeneous computing, including FPGA, GPU, and CPUs.



Taisuke Boku received his M.S. and Ph.D. degrees from the department of electrical engineering, faculty of science and technology, Keio University. He joined to Center for Computational Sciences (former Center for Computational Physics) at University of Tsukuba where he is currently the director and the HPC division leader. He has been working there more than 25 years for HPC system architecture, system software, and performance evaluation on various scientific applications. In these years, he has been playing the central role of system development on CP-PACS (ranked as number one in TOP500 in 1996), FIRST, PACS-CS, HA-PACS and Cygnus as the representative supercomputers in Japan. He is currently the Director of Center for Computational Sciences, University of Tsukuba, and the Vice Director of JCAHPC (Joint Center for Advanced HPC) which is a joint organization by University of Tsukuba and the University of Tokyo. He is also the President of HPCI Consortium Japan from 2020. He received ACM Gordon Bell Prize in 2011, Paper Awards at IPSJ in 2004 and 2005, Best Paper Award at HPC Symposium of IPSJ in 2003 and 2016, Best Paper Award at HEART Symposium in 2014, and SCA Asia HPC Leadership Award in 2020.

Kohji Yoshikawa received his Ph.D. degree in physics and astrophysics from Kyoto University in 2002. He worked as a postdoctoral researcher in the Department of Physics, the University of Tokyo. Since 2007, he has been a lecturer of Center for Computational Sciences, University of Tsukuba. His current research interests include numerical simulations of the large-scale structure in the universe, and high performance computing in numerical astrophysics.

Makito Abe received his Ph.D. from University of Tsukuba in 2016. The current position is a postdoctoral fellow in Center for Computational Sciences, University of Tsukuba. His research interests include computational astrophysics, radiative transfer and bioimaging.

Masayuki Umemura received his Ph.D. degree from the Department of Physics, Hokkaido University, Sapporo, Japan. He worked as an assistant professor of National Astronomical Observatory, Japan from 1988 to 1993. He was a visiting professor in Princeton University in 1992. He worked as an associate professor of Center for Computational Physics, University of Tsukuba from 1993 to 2002, and has been a professor of Center for Computational Sciences, University of Tsukuba. His research interests include First stars, Galaxy Formation, Supermassive Black Holes, Astrobiology, and Computational Medical Science.