GPU–FPGA-accelerated Radiative Transfer Simulation with Inter-FPGA Communication

Ryohei Kobayashi
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
rkobayashi@ccs.tsukuba.ac.jp

Norihisa Fujita
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
fujita@ccs.tsukuba.ac.jp

Yoshiki Yamaguchi
Institute of Systems and Information Engineering, University of Tsukuba
Tsukuba, Ibaraki, Japan
yoshiki@cs.tsukuba.ac.jp

Taisuke Boku
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
taisuke@ccs.tsukuba.ac.jp

Kohji Yoshikawa
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
kohji@ccs.tsukuba.ac.jp

Makito Abe
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
mabe@ccs.tsukuba.ac.jp

Masayuki Umemura
Center for Computational Sciences, University of Tsukuba
Tsukuba, Ibaraki, Japan
umemura@ccs.tsukuba.ac.jp

ABSTRACT

The complementary use of graphics processing units (GPUs) and field programmable gate arrays (FPGAs) is a major topic of interest in the high-performance computing (HPC) field. GPU–FPGA-accelerated computing is an effective tool for multiphysics simulations, which encompass multiple physical models and simultaneous physical phenomena. Because the constituent operations in multiphysics simulations exhibit varying characteristics, accelerating these operations solely using GPUs is often challenging. Hence, FPGAs are frequently implemented for this purpose. The objective of the present study was to further improve application performance by employing both GPUs and FPGAs in a complementary manner. Recently, this approach has been applied to the radiative transfer simulation code for astrophysics known as ARGOT, with evaluation results quantitatively demonstrating the resulting improvement in performance. However, the evaluation results in question came from the use of a single node equipped with both a GPU and FPGA. In this study, we extended the GPU–FPGA-accelerated ARGOT code to operate on multiple nodes using the message passing interface (MPI) and an FPGA-to-FPGA communication technology scheme called Communication Integrated Reconfigurable CompUting System (CIRCUS). We evaluated the performance of the ARGOT code with multiple GPUs and FPGAs under weak scaling conditions, and found it to achieve up to 12.8x speedup compared to the GPU-only execution.

HPC ASIA 2023, February 27-March 2, 2023, Singapore, Singapore

ACM ISBN 978-1-4503-9805-3/23/02.
https://doi.org/10.1145/3578178.3578231

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**.

KEYWORDS

GPU, FPGA, Multi-hetero Acceleration, Multiphysics, CUDA, OpenCL, Inter-FPGA communication

ACM Reference Format:
Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, and Masayuki Umemura. 2023. GPU–FPGA-accelerated Radiative Transfer Simulation with Inter-FPGA Communication. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023), February 27-March 2, 2023, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3578178.3578231

1 INTRODUCTION

Although graphics processing units (GPUs) are widely used in high-performance computing (HPC) systems as accelerators owing to their good peak performance and high memory bandwidth, they are not suitable for all applications, for example, multiphysics. Multiphysics refers to the coupled processes or systems involving several simultaneously occurring physical fields, as well as the study of such processes and systems. Simulations with multiple interacting physical properties and computations may include processes unsuitable for GPUs. Accordingly, field-programmable gate arrays (FPGAs) have been implemented to handle any such processes. This concept, referred to as Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices (CHARM), was implemented by [10], who demonstrated its usefulness for radiative transfer simulations in astrophysics. However, the study in question used a single node equipped with both GPU and FPGA devices, with a single CPU process invoking and controlling the CUDA and OpenCL kernels running on the GPU and FPGA, respectively.

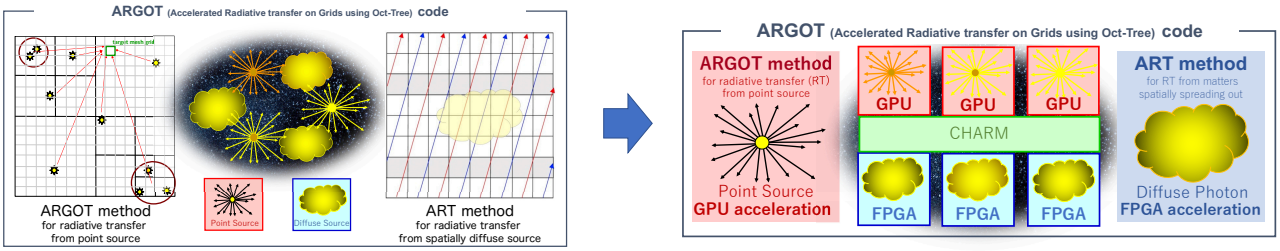

Figure 1: Overview of the ARGOT code and how to accelerate it using the CHARM concept.

In this study, we implemented GPU–FPGA-accelerated computing among multiple computing nodes using the message passing interface (MPI) and an FPGA-to-FPGA communication technology called Communication Integrated Reconfigurable CompUting System (CIRCUS) [3]. Thus, we enabled the parallelization of GPUs via CUDA and MPI programming, and that of FPGAs via OpenCL and CIRCUS. We used CUDA and OpenCL programming in the implementation of GPU–FPGA-accelerated computing primarily because the majority of existing HPC applications are CUDA-based implementations, and rewriting their entire code in OpenCL is very burdensome for developers. Furthermore, most GPUs used in HPC are manufactured by NVIDIA, making maximizing their performance using CUDA, a programming model that follows the GPU architecture, simple.

We applied our proposed approach to the same astrophysics application examined in [10], and quantitatively evaluated its usefulness. The evaluation results show that under weak scaling conditions, the use of FPGAs on one node achieves 6.8x speedup compared to GPU-only execution, whereas the use of FPGAs on two nodes achieves 12.8x speedup compared to GPU-only execution.

Our contributions in this study are as follows:

- We propose a methodology for running GPU–FPGA-accelerated computing on multiple nodes in practical applications.
- We describe our proposed method's implementation with CUDA and OpenCL, as well as its node-parallelization with MPI and inter-FPGA communication technology.
- We demonstrate the proposed method's performance on an HPC cluster equipped with GPUs and FPGAs, and quantitatively show the advantages of GPU–FPGA-accelerated computing over GPU-only execution.

The remainder of this paper is organized as follows. In Section 2, we describe our target astrophysics application for this study. Details pertaining to the computational kernel and FPGA-to-FPGA communication technology for astrophysics applications are described in Section 3. An overview of multi-node parallelization of GPU–FPGA-accelerated ARGOT code is provided in Section 4, and the evaluation results are presented in Section 5. We introduce several related studies in Section 6, and present our conclusions in Section 7.

2 ARGOT: RADIATIVE TRANSFER SIMULATION CODE FOR ASTROPHYSICS

ARGOT is an astrophysics simulation code developed at the Center for Computational Sciences, University of Tsukuba to examine how the first celestial objects were generated in the early stages of the universe. As shown in Figure 1, two methods were combined to solve radiative transfer problems: the ARGOT method [13], which computes the radiative transfer from point sources, and the ART method [15], which computes the radiative transfer from sources spreading out within the target space. Using CHARM, the ARGOT method's execution is offloaded to GPUs, whereas that of ART is offloaded to FPGAs, thereby improving overall application performance. The following subsections provide brief descriptions of both methods.

2.1 ARGOT Method

The ARGOT method performs a computation of the optical depth between each point radiation source and corresponding target mesh grid, which represents the end point of a light ray. Assuming a constant number of mesh grids, the computational complexity is proportional to the number of point radiation sources. To improve this, ARGOT builds an octree representing the distribution of radiation sources, as shown in Figure 1. Although the figure is two-dimensional, the computational space is three-dimensional and hierarchically subdivided into eight cubic cells, until each cell contains a single radiation source or is sufficiently small relative to the computational domain. Consequently, the sources of 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, such as that illustrated in Figure 1, the photon flux originating from each radiation source at the target mesh grid is given by

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

where $L(\nu)$ and $\tau(\nu)$ represent the intrinsic luminosity and optical depth for a given frequency ν, respectively. In addition, $\tau(\nu)$ 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 the gas molecules that absorb light.

The tree data structure is commonly employed in the field of computational astrophysics, with the tree method in the N-body problem being a typical example of this approach [12]. Because this method is highly compatible with GPU implementations, we offloaded ARGOT to GPUs in accordance with the tree method

reported in [14]. A ray is assigned to each CUDA thread, and the corresponding computations are executed in parallel.

2.2 ART Method

The ART method is based on ray-tracing in a three-dimensional space split into meshes. Because the computation phase of ART accounts for more than 90% of the ARGOT code, its acceleration directly improves overall ARGOT performance. As shown in Figure 1, multiple incident rays originate from a single boundary and move in mutually parallel linear paths without reflection or refraction. The ART method solves a radiation transfer equation along parallel light rays spanning the computational volume using the following equation:

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

This calculation is performed whenever a ray passes through a mesh grid. For a given incoming radiation intensity I_ν^{in} along the $\hat{\boldsymbol{n}}$ direction, the outgoing radiation intensity I_ν^{out} after passing 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 absorption coefficient of the mesh grid, respectively. The number of meshes depends on the configuration of the target problem. Our target problems range from 100^3 to 1000^3 meshes. The direction (angle of incidence) of the ray is computed using the HEALPix algorithm [6]. The number of ray angles also varies with respect to problem size. In the present study, this number is always at least 768, corresponding to the resolution parameter $N_{side} = 8$ in HEALPix.

Because the ART method uses ray tracing, the computational order within each ray must be sequential, whereas computations for different rays can be conducted in parallel because no two rays are computationally dependent on each other. However, implementing ART on a SIMD-like architecture presents two challenges [10].

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

Second, the integration of mesh data resulting from two rays in close proximity to each other will cause a conflict. When multiple rays pass through shaded mesh grids, as shown in Figure 1, the physical quantities in those mesh grids must be incremented in an atomic manner. However, the atomic operation itself has a certain overhead. If a large number of threads conduct atomic operations simultaneously, many contentions may occur, and processing may significantly slow 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 [15] does not compute neighboring rays simultaneously. Tracing along the red and blue light rays is separately performed, as illustrated in Figure 1. However, this method further exacerbates memory access problems by scattering the memory access patterns. This overhead is expected to be nearly cubically proportional, and close to the number of atomic operations.

Although the ART method is based on ray tracing, it is inherently different from computer graphics (CG) algorithms, which can be

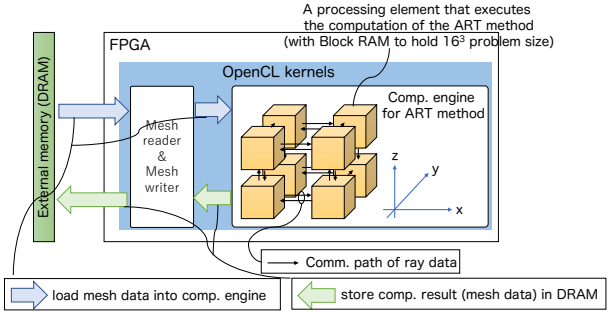

Figure 2: FPGA implementation of ART method with Intel FPGA SDK for OpenCL [4, 9].

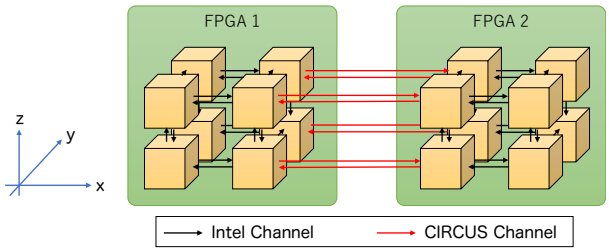

Figure 3: Multi-FPGA implementation of ART method [3].

accelerated by GPUs based on the NVIDIA Turing architecture. Whereas ray tracing in CG retroactively calculates the reflection and transmission of light on object surfaces from the observer's perspective, ART calculates the average radiant intensity in all directions whenever a ray passes through a mesh grid. In other words, the term "ray tracing" is the only thing the two approaches have in common.

Given the ART method's characteristics, we consider SIMD-style processors such as CPUs and GPUs to be unsuitable for acceleration. By contrast, FPGAs can access on-chip internal memory with low latency and high bandwidth for random access. Furthermore, FPGA hardware enables the programming of memory access patterns, making it a viable option to accelerate the ART method. As a demonstration experiment, the authors of [4] implemented an FPGA-based ART method accelerator using OpenCL and quantitatively evaluated its usefulness. We used their accelerator as a foundation for our own FPGA implementation, as described in the following section.

3 PARALLELIZATION OF ART METHOD USING MULTI-FPGA

3.1 FPGA Implementation of the ART Method

Figure 2 illustrates the FPGA implementation of the ART method with Intel FPGA SDK for OpenCL [4, 9]. This implementation consists of processing elements (PEs)—arithmetic cores written in OpenCL—connected in three dimensions ($2 \times 2 \times 2$) using Channel, a proprietary extension of the Intel FPGA SDK for OpenCL. The PEs mutually transmit ray data via the Channel, and execute

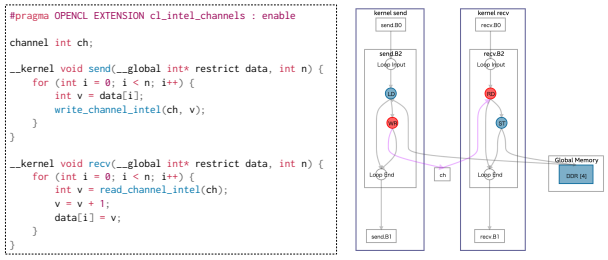

Figure 4: Pipelined hardware generated by the offline compiler of Intel FPGA SDK for OpenCL. The figure on the right was generated using the Graph Viewer included in the compilation report generated by the offline compiler.

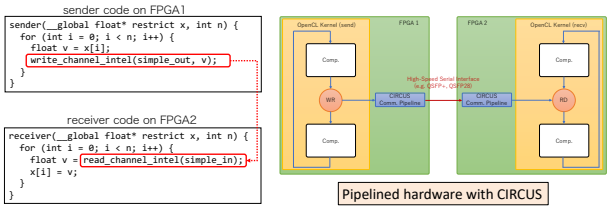

Figure 5: Pipelined hardware built on two FPGAs using CIRCUS.

Equation (3) upon data reception. Subsequently, the ray data reflecting the result of Equation (3) is sent to the PE responsible for the next problem space located in the ray's direction, thereby realizing the ART method on an FPGA. Because all PEs are pipelined at the clock-cycle level simultaneously, the ART accelerator implemented in a single FPGA can take advantage of temporal (pipeline) and spatial parallelism to maximize performance.

The multi-FPGA implementation of the ART method extends the channel connections between PEs to different FPGAs using CIRCUS, an inter-FPGA communication technology [3]. It can therefore be said that a massive PE cluster is constructed by combining multiple FPGAs. Figure 3 presents a schematic diagram of the ART method's accelerator implemented with two FPGAs. The black and red arrows represent channel and FPGA-to-FPGA connections realized by CIRCUS, respectively.

3.2 CIRCUS: Inter-FPGA Communication

CIRCUS is a framework that enables inter-FPGA communication at the level of OpenCL abstraction, with a communication system built upon the assumption of pipeline communication [5]. Pipelining is a fundamental processing structure for FPGAs, and OpenCL compilers build pipelines from loop structures (for-loops, while-loops, etc.) in the code. Figure 4 shows an example.

The code snippet has a `send` kernel that reads a value from memory, and a `recv` kernel that writes the value to memory after incrementing it by 1, with the two connected by a channel. The LD, ST, RD, and WR labels denote read from memory, write to memory, read from channel, and write to channel operations, respectively. Although each loop is implemented as a different kernel and runs

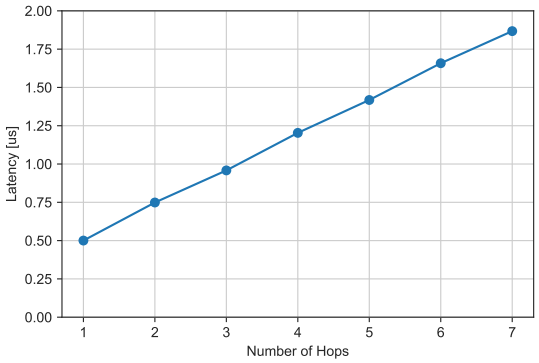

Figure 6: Latency results measured from ping-pong benchmark [5].

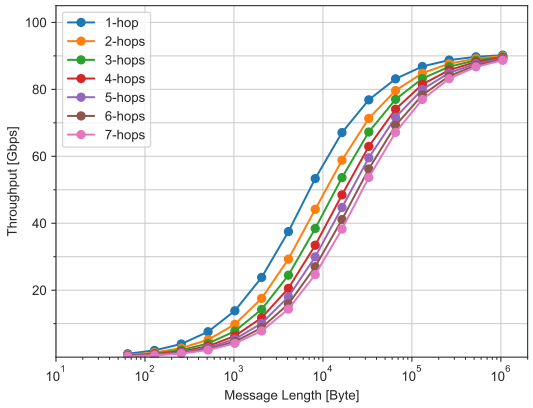

Figure 7: Throughput results measured from ping-pong benchmark [5].

asynchronously, the data dependencies with respect to channel imply that a large overall pipeline has been constructed. The underlying concept of CIRCUS is to realize that feature across multiple FPGAs, as shown in Figure 5. By incorporating a mechanism for pipeline communication into the OpenCL environment, Channel communication is enabled between different FPGAs, and a pipeline that integrates communication and computation is constructed.

Figure 6 and Figure 7 present the latency and throughput measurements reported in [5], respectively, obtained by ping-ponging messages ranging in size from 64 Bytes to 1 MB from 1 hop to 7 hops. In this context, hops were used to measure the distance from the source to the destination. For example, "1-hop" corresponds to the neighboring FPGA, whereas "2-hop" refers to the FPGA next to it. The minimum and maximum latency were 0.5 μs and 1.87 μs, respectively, with the additional latency per hop being approximately 0.25 μs. The maximum throughput was 90.2 Gbps for one hop and 88.7 Gbps for seven hops. Throughput degradation was observed with an increasing number of hops, particularly for small

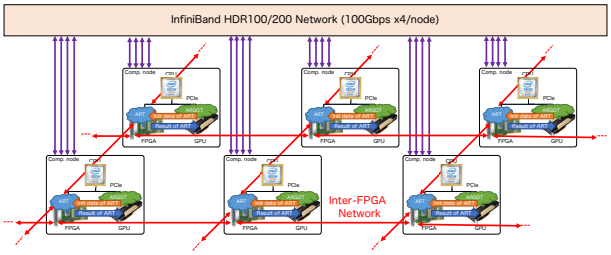

Figure 8: Overview of multi-node parallelization in GPU–FPGA-accelerated ARGOT code. When implemented using the Cygnus supercomputer [1], the ART method is executed in parallel between FPGAs connected by optical links, while the ARGOT method is parallelized on multiple GPUs using the MPI programming model [10].

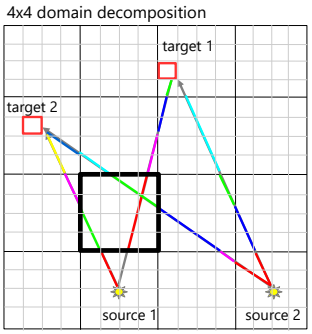

Figure 9: Overview of node parallelization for ARGOT method [13].

to medium message lengths, because the ratio between latency and total communication time increased with communication latency. However, as message size increased, the ratio decreased, with data being eventually transferred at a throughput close to 90% of the theoretical peak bandwidth of 100 Gbps.

4 MULTI-NODE PARALLELIZATION OF GPU–FPGA-ACCELERATED ARGOT CODE

4.1 Overview

Figure 8 presents an overview of the multi-node parallelization of GPU–FPGA-accelerated ARGOT code. All compute nodes are equipped with GPUs and FPGAs, and both types of devices are controlled by MPI processes assigned to each node. Similar to [10], the FPGA on each node runs the ART method and the GPU runs all remaining computation, including the ARGOT method. Transfers of initial data and ART output occur between both devices on each computation node. Although it is possible to perform data transfers using PCIe DMA between GPU and FPGA [11], results obtained by [10] make it clear that the resulting benefit is insufficient, and therefore not applied within this study.

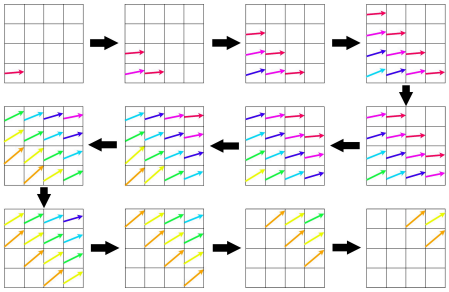

Figure 10: Overview of node parallelization for ART method with Multiple Wave Front [15].

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

// to use comm. kernels provided by CIRCUS
#include <circus.h>

channel rt_ch[6][NPE_X][NPE_Y][NPE_Z];

// CIRCUS Channel
channel extout_x_neg_x0_y0_z0;
channel extin_x_neg_x0_y0_z0;

// Problem space contained by PE
#define PE_MESH_SIZE (16 * 16 * 16)

// 12 channels required (1 input / 1 output per surface)
#define PE_INPUT_X_POS rt_ch[1][1][0][0]
#define PE_INPUT_X_NEG extin_x_neg_x0_y0_z0
#define PE_INPUT_Y_POS rt_ch[3][0][1][0]
#define PE_INPUT_Y_NEG rt_ch[2][0][1][0]
#define PE_INPUT_Z_POS rt_ch[5][0][0][1]
#define PE_INPUT_Z_NEG rt_ch[4][0][0][1]
#define PE_OUTPUT_X_POS rt_ch[0][0][0][0]
#define PE_OUTPUT_X_NEG extout_x_neg_x0_y0_z0
#define PE_OUTPUT_Y_POS rt_ch[2][0][0][0]
#define PE_OUTPUT_Y_NEG rt_ch[3][0][0][0]
#define PE_OUTPUT_Z_POS rt_ch[4][0][0][0]
#define PE_OUTPUT_Z_NEG rt_ch[5][0][0][0]

__kernel void PE_x0_y0_z0(...) {
// Use Block RAM for arrays to store mesh space data (optical
// thickness and source function)
__local struct radiation_mesh rmesh[PE_MESH_SIZE];

// Stores the optical thickness in the mesh and the source function
// of the mesh
set_radiation_mesh(rmesh);

// Perform ray tracing on all 768 x N^2 rays (N = 16)
bool exit = false;
while (!exit) {
  bool x_neg;
  bool x_pos;
  bool y_neg;
  bool y_pos;
  bool z_neg;
  bool z_pos;

  // Detect the ray input
  INPUT_COND(x_neg);
  INPUT_COND(x_pos);
  INPUT_COND(y_neg);
  INPUT_COND(y_pos);
  INPUT_COND(z_neg);
  INPUT_COND(z_pos);

  // Receive the ray data if input is available from adjacent PEs
  if      (x_neg) ray = read_channel_intel(PE_INPUT_X_NEG);
  else if (x_pos) ray = read_channel_intel(PE_INPUT_X_POS);
  else if (y_neg) ray = read_channel_intel(PE_INPUT_Y_NEG);
  else if (y_pos) ray = read_channel_intel(PE_INPUT_Y_POS);
  else if (z_neg) ray = read_channel_intel(PE_INPUT_Z_NEG);
  else if (z_pos) ray = read_channel_intel(PE_INPUT_Z_POS);

  // Execute the arithmetic kernel of the ART method
  // (calculation of radiation intensity)
  calc_intensity(&ray, rmesh);

  bool x_neg_out;
  bool x_pos_out;
  bool y_neg_out;
  bool y_pos_out;
  bool z_neg_out;
  bool z_pos_out;

  // Detect the ray output
  OUTPUT_COND(x_neg_out);
  OUTPUT_COND(x_pos_out);
  OUTPUT_COND(y_neg_out);
  OUTPUT_COND(y_pos_out);
  OUTPUT_COND(z_neg_out);
  OUTPUT_COND(z_pos_out);

  // Send the ray data if the next computed region of a ray is a mesh
  // space held by an adjacent PE
  if      (x_neg_out) write_channel_intel(PE_OUTPUT_X_NEG, ray);
  else if (x_pos_out) write_channel_intel(PE_OUTPUT_X_POS, ray);
  else if (y_neg_out) write_channel_intel(PE_OUTPUT_Y_NEG, ray);
  else if (y_pos_out) write_channel_intel(PE_OUTPUT_Y_POS, ray);
  else if (z_neg_out) write_channel_intel(PE_OUTPUT_Z_NEG, ray);
  else if (z_pos_out) write_channel_intel(PE_OUTPUT_Z_POS, ray);

  // Detect the ray tracing terminated
  CHECK_TERMINATION(exit);
}
}
```

Figure 11: Code snippet of multi-FPGA implementation of ART method with CIRCUS.

4.2 Node Parallelization of the ARGOT Method

Node parallelization is already implemented in the original ARGOT code, which uses the general CUDA and MPI programming model. Figure 9 represents an overview of node parallelization for ARGOT method. Here, the simulation space is equally divided among all dimensions (4×4 domain decomposition in the figure). Rays spanning multiple nodes are divided into "ray segments" at the boundaries between nodes, and each segment's computation is performed on each node in parallel. Because the segments corresponding to each node are mutually independent, each segment is assigned to a thread and processed in parallel. The cumulative computation result is obtained by summing the optical thickness results for each segment. If only one MPI process is used, four ray segments – source 1 $\longrightarrow$ target 1, source 1 $\longrightarrow$ target 2, source 2 $\longrightarrow$ target 1, and source 2 $\longrightarrow$ target 2 – are considered "ray segments," wherein each ray is assigned to a thread and processed in parallel.

4.3 Node Parallelization of the ART Method

Figure 10 presents an overview of node parallelization for the ART method with Multiple Wave Front [15]. Multiple Wave Front is the method used in the CPU/GPU implementation of the original

Table 1: Our experimental environment.

Hardware specifications	
CPU	Intel® Xeon® E5-2690 v4 × 2
Host Memory	DDR4-2400 8GB x8
GPU	NVIDIA Tesla P100 (PCIe Gen3 x16 card version)
GPU Memory	16 GiB CoWoS HBM2 @ 720 GB/s with ECC
FPGA	BittWare 520N (Intel Stratix 10 GX2800)
FPGA Memory	DDR4 2400MHz 32 GB (8GB × 4)
InfiniBand	Mellanox ConnectX-4 Single-port EDR MCX455A-ECAT
Software specifications	
Host OS	CentOS 7.9
Linux Kernel Version	3.10.0-1160.62.1.el7.el7.x86_64
Host Compiler	gcc 4.8.5
GPU Compiler	CUDA 11.2.152
MPI	Open MPI 3.1.0
OpenCL SDK	Intel FPGA SDK for OpenCL 19.4.0 Build 64 Pro Edition

ARGOT code, and achieves pipelined node parallelization while restricting the firing order for each ray direction. The ART method on multiple nodes is realized by receiving rays sent out from one problem space using the MPI process of the following program space according to ray direction, thereby executing a radiative transfer simulation.

Parallelization of ART methods with multiple FPGAs employs an equivalent approach, wherein the MPI processes in the CPU/GPU implementation correspond to the PEs in the FPGA implementation. CIRCUS allows for a massive cluster of PEs across multiple FPGAs. Here, communication between PEs within an FPGA is achieved using the Intel Channel, whereas that between PEs across different FPGAs is achieved using the CIRCUS Channel, which sends and receives ray data for the ART method through the direct inter-FPGA network.

Figure 11 presents a code snippet from a multi-FPGA implementation of the ART method with CIRCUS. In this example, the problem space is split along the x-axis. The code snippet is a PE pseudo code located at (x, y, z) = (0, 0, 0), with access in the x-(x_neg) direction via CIRCUS, and all other dimensions to the internal Channel. The PE calculation requires three components—ray input, radiation intensity, and ray output—that are implemented as hardware running in a pipeline within the FPGA. This calculation is performed after all required data are stored in the FPGA's block RAM, thereby avoiding performance degradation due to memory access operations.

5 EVALUATION

5.1 Experimental Settings

Table 1 illustrates our experimental machine configuration. This is a heterogeneous cluster, wherein each node is composed of three types of devices: two Intel® Xeon® E5-2690 v4 CPUs, a single NVIDIA P100 GPU for PCIe-based servers (Gen3 x16), and a single

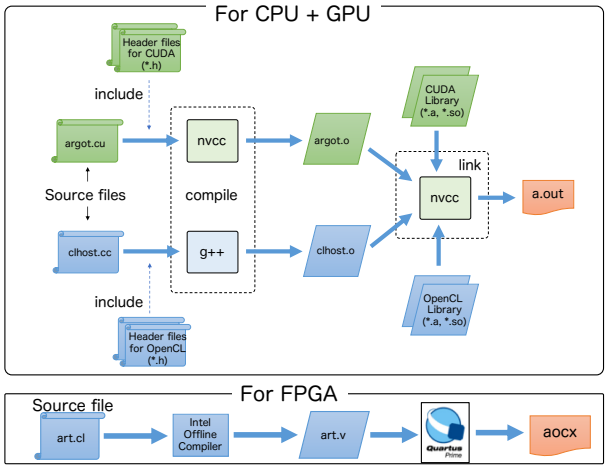

Figure 12: Compilation flow of GPU–FPGA-accelerated ARGOT code [10].

BittWare 520N FPGA board equipped with 100 Gbps Ethernet × 4 channels. In this evaluation, we employed a single CPU, GPU, and FPGA located on the same CPU socket to avoid performance degradation caused by PCIe access over the Intel Quick Path Interconnect (QPI). Although the FPGA board physically connects to the CPU through a PCIe Gen3 x16 interface, the PCIe IP core implemented in the FPGA supports up to PCIe Gen3 x8 data transfer. Therefore, the maximum data transfer capability between GPU and FPGA is 5.33 GB/s, which is the harmonic mean of PCIe Gen3 x16 (16 GB/s: CPU-GPU) and PCIe Gen3 x8 (8 GB/s: CPU-FPGA) data transfer. However, the analysis in [10] indicates that GPU-FPGA communication does not significantly affect performance, as it accounts for approximately 1% of the total execution time.

Our experiments were conducted on the CentOS 7.9 operating system. GPU–FPGA-accelerated ARGOT code was compiled separately using nvcc and g++, as in [10]. Figure 12 illustrates the corresponding compilation flow. We employed CUDA ver. 11.2.152 and GCC ver. 4.8.5. The FPGA-based ART accelerator with CIRCUS was implemented in OpenCL kernel code, compiled with the offline compiler provided by the Intel FPGA SDK for OpenCL (ver. 19.4.0 Build 64 Pro Edition). We used OpenMPI 3.1.0 to enable node parallelization and allocate one process per node, from which the GPU and FPGA kernels are invoked.

Due to time constraints, the current FPGA implementation of ART is not equipped with the feature [10] that divides a large problem stored in DDR memory into small blocks that can be stored in BRAM and executes the ART calculation via time division multiplexing in each block by FPGA. Our FPGA implementation of ART encompasses eight PEs (2^3), each having BRAMs for 16^3 problem space. Consequently, the FPGAs can be assigned a problem size of 32^3. In this paper, we kept the problem size assigned to FPGAs fixed at 32^3 and quantitatively evaluated the improvement in performance with an increasing number of FPGAs, which means parallel efficiency under weak scaling conditions. Nside, which is a parameter used to determine the resolution in HEALpix, was set to 8, generating 768 different ray angles.

Table 2: Resource usage and clock frequency in FPGA implementation of ART method with CIRCUS.

ALMs	Registers	M20Ks	DSPs	fmax [MHz]
691,082 (74%)	1,473,614 (39%)	5,076 (43%)	1,916 (33%)	216.96

In this evaluation, the CPU computation time, including the costs of launching and synchronizing the device, was measured for both FPGA and GPU implementations. The time required to transfer data between the nodes was also accounted for using the `MPI_Barrier()` and `gettimeofday()` functions.

5.2 Resource Consumption

Table 2 illustrates the FPGA resource utilization. The Adaptive Logic Module (ALM) 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 hardware components implemented in the FPGA. The M20K memory block is an internal memory of the FPGA, referred to as a block RAM. All internal buffers such as FIFOs are implemented using memory blocks. The Digital Signal Processor (DSP) is a built-in hardware component that enables faster and more compact implementations of integer multiplications and floating-point operations than programmable logic components. Here, "fmax" denotes the maximum operating frequency in the clock domain for OpenCL kernels.

As listed in the table, the ALMs are the most resource-intensive components in this design, with more than half of the total ALMs used. Of the 74% ALM utilization, 12.5% encompasses the hardware overhead of CIRCUS. Even omitting this overhead, the ALM utilization still incurs a bottleneck when attempting to increase performance. From the perspective of resource utilization, the use of all DSPs is the optimization objective, as they implement floating additions and multiplications. However, we cannot double the number of PEs to increase DPS utilization, as this would incur an ALM overutilization.

We therefore referred to the analysis in [9], and considered the number of possible PEs to be 16 ($2 \times 2 \times 4$). Because each PE has a working memory with 16^3 meshes, the number of PEs for each dimension should be a power of 2. In general, as resource usage per PE decreases, the operating frequency increases because place-and-routing becomes easy to apply. This also improves the performance of the ART method.

5.3 Performance Evaluation of GPU–FPGA-accelerated ARGOT Code on Multiple Nodes

Because the current FPGA implementation can only handle problem sizes of up to 32^3, we allocated a problem size of 32^3 per node and compared the performance of the GPU–FPGA-accelerated ARGOT code compared to that of the GPU implementation under weak scaling conditions. Figure 13 presents our evaluation results. We employed up to two nodes throughout the evaluation, with the total problem size at two nodes being $64 \times 32 \times 32$.

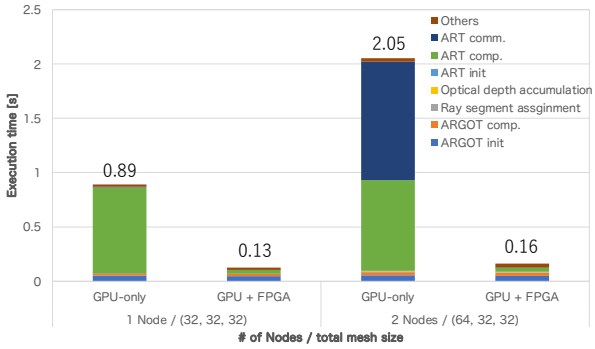

Figure 13: Evaluation result of ARGOT code using 2 GPUs with InfiniBand and 2 FPGAs with CIRCUS.

We first focused on the ARGOT code's performance at one node. The GPU implementation shows that the ART method's execution time dominates the entire ARGOT code. Therefore, as previously described in Section 2, the direct acceleration of ART significantly improves overall ARGOT performance, which we have achieved by offloading the ART method to an FPGA. Consequently, the ARGOT execution time was reduced from 0.89 to 0.13 seconds, representing a 6.8x speedup over the GPU-only execution. The ART accelerator implemented in FPGAs is a hardware that takes advantage of temporal (pipeline) and spatial parallelism to maximize the effective performance of the ART method. Furthermore, the problem sizes reported in [9] are too small to sufficiently exploit the 3,584 CUDA cores of the GPU, which means that the requisite parallelism is not achievable. The GPU kernel activation and CPU-GPU communication may also reduce performance.

Examining the ARGOT code performance on two nodes, it is apparent that the GPU implementation increased the ART method's computation time by 5.4% over the single-node execution. However, the computation time incurred by each node was almost the same because performance was evaluated under the weak scaling condition. As shown in Figure 10, MPI communication enables the transfer of ray data generated by the ART method, with the time required indicated as "ART comm." in Figure 13. The communication overhead accounts for half the total ARGOT execution time, as the communication overhead is manifested by the smaller ratio of operations to communication owing to the small problem size in the GPU.

Conversely, when the ART method is executed on a multi-FPGA with two nodes, the total ARGOT execution time is reduced to 0.16 seconds, as the pipeline that integrates communication and computation is built by CIRCUS. That is, all ART operations – i.e., Equation (3) and inter-FPGA communication in FPGAs on both nodes – are executed completely simultaneously. Therefore, once the pipeline is filled, the ART method targeting a $64 \times 32 \times 32$ problem size executes with almost 100% efficiency. The initial overhead associated with filling the pipeline is expressed as a 13% increase in the ART method's run time. However, the results for two nodes exhibit a relative error of up to 10^{-1}, the cause of which is currently under investigation.

6 RELATED WORK

To the best of our knowledge, research on accelerated computing using GPUs and FPGAs has been conducted for more than a decade. KH Tsoi et al. [17] proposed a heterogeneous computer cluster called Axel that encompasses a collection of nodes – each of which can include multiple types of accelerators such as FPGAs and GPUs – and demonstrated that FPGAs, GPUs, and CPUs can run collaboratively for an N-body simulation. At that time, there were no commercial or open-source frameworks that could comprehensively handle CPUs, GPUs, and FPGAs simultaneously. Therefore, the authors developed their own framework for the Axel cluster based on MapReduce. The study demonstrated that the simultaneous use of FPGAs, GPUs, and CPUs for an N-body simulation yields a 4.4x increase in the execution speed of 16 compute nodes compared to CPU-only execution. However, this improvement in performance is largely a result of decreased CPU performance, and the performance gain from the addition of FPGAs based on the GPU implementation is approximately 2x at most (on a single node).

Our current implementation of GPU–FPGA-accelerated computation is a combination of CUDA and OpenCL programming, as the computation kernels running on GPUs are written in CUDA, whereas those running on FPGAs are written in OpenCL. However, such mixed-programming implementations place a heavy burden on application developers, and research is being conducted into programming environments that enable the comprehensive control of GPUs, CPUs, and FPGAs in a single language. María Angélica Dávila Guzmán et al. [7] developed one such framework by extending EngineCL, an OpenCL-based framework that provides high-level data scheduling and management. The collaborative computation of the CPU, GPU, and FPGA within the enhanced EngineCL improved energy efficiency in five of the six benchmarks.

The key to parallel computing using multiple FPGAs is the communication infrastructure between FPGAs, with research on the subject being conducted worldwide. Tiziano et al. developed the Streaming Message Interface (SMI) [2] on a Noctua supercomputer, which is a communication framework that enables inter-FPGA communication. The authors demonstrated SMI to achieve an almost equivalent communication performance to that of the theoretical peak bandwidth. However, this is only because the FPGA transceivers in Noctua only support up to 40 Gbps communication, and the packet routing of the SMI overhead did not become apparent. The experimental results of [8] show that SMI's performance is not even half of the theoretical peak bandwidth in an environment supporting 100 Gbps Ethernet, owing to the packet routing overhead. We therefore adopted CIRCUS instead of SMI as the platform for FPGA-to-FPGA communication.

As mentioned above, there are several studies related to this study, but most of them have demonstrated the usefulness with benchmarks, and there are few studies of GPU–FPGA-accelerated computing for real applications. We adopted a policy of offloading only appropriate computational components to the FPGA, for which the GPU would yield insufficient performance. As a result, unlike [17], we confirmed that the combined use of FPGAs can achieve up to 12.8x performance improvement compared to GPU-only execution performance. This finding represents a unique contribution of our study.

7 CONCLUSION

In this paper, we present the implementation and performance evaluation of multi-node parallelization of the GPU–FPGA-accelerated ARGOT code. The ARGOT method was parallelized on multiple GPUs using CUDA and MPI programming, whereas the ART method was parallelized on multiple FPGAs using OpenCL and an inter-FPGA communication technology known as CIRCUS. The performance of the GPU–FPGA-accelerated ARGOT code was evaluated under weak scaling with a problem size of 32^3 per node compared to the performance of the GPU implementation. Evaluation results demonstrate that the combined use of FPGAs and GPUs achieves a 6.8x speedup on one node and 12.8x speedup on two nodes. The ART accelerator implemented in FPGAs takes advantage of pipeline and spatial parallelism to maximize performance. In addition, CIRCUS enables the communication pipeline to integrate with the computation pipeline of the ART method, which incurs negligible communication overhead compared to that of GPU implementations, thereby ensuring high parallel efficiency. However, the results for the two-node execution exhibited a relative error of up to 10^{-1}, the cause of which is currently under investigation.

We plan to evaluate the performance of the GPU–FPGA-accelerated ARGOT with an increased number of nodes. However, because CIRCUS lacks flow control, it may not function with a large number of FPGAs, even if the ART communication patterns are simple. In that case, we will consider incorporating Kyokko, [16], an open-source FPGA-to-FPGA communication IP with flow control, into CIRCUS. Thus, our future studies will examine the cause of the large relative error, as well as evaluate the performance of GPU–FPGA-accelerated ARGOT code with an increased number of nodes.

ACKNOWLEDGMENTS

This work used computational resources of TSUBAME3.0 provided by Tokyo Institute of Technology through the HPCI System Research Project (Project ID: hp190099). This work was also supported in part by the Multidisciplinary Cooperative Research Program in CCS, University of Tsukuba, and JSPS KAKENHI, Grant Number 21H04869. We also thank the Intel University Program for providing the hardware and software.

REFERENCES

[1] Taisuke Boku, Norihisa Fujita, Ryohei Kobayashi, and Osamu Tatebe. 2023. Cygnus - World First Multihybrid Accelerated Cluster with GPU and FPGA Coupling. In *Workshop Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP Workshops '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. https://doi.org/10.1145/3547276.3548629

[2] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. 2019. Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 82, 33 pages. https://doi.org/10.1145/3295500.3356201

[3] Norihisa Fujita, Ryohei Kobayashi, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, and Masayuki Umemura. 2020. OpenCL-enabled Parallel Raytracing for Astrophysical Application on Multiple FPGAs with Optical Links. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, New York, NY, USA, 48–55. https://doi.org/10.1109/H2RC51942.2020.00011

[4] Norihisa Fujita, Ryohei Kobayashi, Yoshiki Yamaguchi, Yuma Oobata, Taisuke Boku, Makito Abe, Kohji Yoshikawa, and Masayuki Umemura. 2018. Accelerating Space Radiative Transfer on FPGA Using OpenCL. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable*

Technologies (Toronto, ON, Canada) *(HEART 2018)*. ACM, New York, NY, USA, Article 6, 7 pages. https://doi.org/10.1145/3241793.3241799

[5] Norihisa Fujita, Ryohei Kobayashi, Yoshiki Yamaguchi, Tomohiro Ueno, Kentaro Sano, and Taisuke Boku. 2020. Performance Evaluation of Pipelined Communication Combined with Computation in OpenCL Programming on FPGA. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New York, NY, USA, 450–459. https://doi.org/10.1109/IPDPSW50202.2020.00083

[6] K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. 2005. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *The Astrophysical Journal* 622, 2 (apr 2005), 759–771. https://doi.org/10.1086/427976

[7] María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, and Jose Luis Bosque. 2019. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *Journal of Supercomputing* 75 (3 2019), 1732–1746. Issue 3. https://doi.org/10.1007/S11227-019-02768-Y

[8] Ryuta Kashino, Ryohei Kobayashi, Norihisa Fujita, and Taisuke Boku. 2021. Performance Evaluation of OpenCL-Enabled Inter-FPGA Optical Link Communication Framework CIRCUS and SMI. In *The International Conference on High Performance Computing in Asia-Pacific Region* (Virtual Event, Republic of Korea) *(HPC Asia 2021)*. Association for Computing Machinery, New York, NY, USA, 23–31. https://doi.org/10.1145/3432261.3432266

[9] Ryuta Kashino, Ryohei Kobayashi, Norihisa Fujita, and Taisuke Boku. 2022. Multi-Hetero Acceleration by GPU and FPGA for Astrophysics Simulation on OneAPI Environment. In *International Conference on High Performance Computing in Asia-Pacific Region* (Virtual Event, Japan) *(HPCAsia2022)*. Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3492805.3492817

[10] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, and Masayuki Umemura. 2020. Multi-Hybrid Accelerated Simulation by GPU and FPGA on Radiative Transfer Simulation in Astrophysics. *Journal of Information Processing* 28 (2020), 1073–1089. https://doi.org/10.2197/ipsjjip.28.1073

[11] R. Kobayashi, N. Fujita, Y. Yamaguchi, A. Nakamichi, and T. Boku. 2019. GPU-FPGA Heterogeneous Computing with OpenCL-Enabled Direct Memory Access. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New York, NY, USA, 489–498. https://doi.org/10.1109/IPDPSW.2019.00090

[12] Naohito Nakasato. 2012. Implementation of a parallel tree method on a GPU. *Journal of Computational Science* 3, 3 (2012), 132–141. https://doi.org/10.1016/j.jocs.2011.01.006 Scientific Computation Methods and Applications.

[13] Takashi Okamoto, Kohji Yoshikawa, and Masayuki Umemura. 2012. argot: accelerated radiative transfer on grids using oct-tree. *Monthly Notices of the Royal Astronomical Society* 419, 4 (01 2012), 2855–2866. https://doi.org/10.1111/j.1365-2966.2011.19927.x

[14] Yuka Sano, Ryohei Kobayashi, Norihisa Fujita, and Taisuke Boku. 2022. Performance Evaluation on GPU-FPGA Accelerated Computing Considering Interconnections between Accelerators. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (Tsukuba, Japan) *(HEART2022)*. Association for Computing Machinery, New York, NY, USA, 10–16. https://doi.org/10.1145/3535044.3535046

[15] Satoshi Tanaka, Kohji Yoshikawa, Takashi Okamoto, and Kenji Hasegawa. 2015. A new ray-tracing scheme for 3D diffuse radiation transfer on highly parallel architectures. *Publications of the Astronomical Society of Japan* 67, 4 (05 2015), 62 (1–16). https://doi.org/10.1093/pasj/psv027

[16] Akinobu Tomori and Yasunori Osana. 2021. Kyokko: A Vendor-Independent High-Speed Serial Communication Controller. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies* (Online, Germany) *(HEART '21)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3468044.3468051

[17] Kuen Hung Tsoi and Wayne Luk. 2010. Axel: A Heterogeneous Cluster with FPGAs and GPUs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '10)*. Association for Computing Machinery, New York, NY, USA, 115–124. https://doi.org/10.1145/1723112.1723134