Accelerating Space Radiative Transfer on FPGA using OpenCL

Norihisa Fujita
University of Tsukuba
fujita@hpcs.cs.tsukuba.ac.jp

Ryohei Kobayashi
University of Tsukuba

Yoshiki Yamaguchi
University of Tsukuba

Yuma Oobata*
University of Tsukuba

Taisuke Boku
University of Tsukuba

Makito Abe†
University of Tsukuba

Kohji Yoshikawa
University of Tsukuba

Masayuki Umemura
University of Tsukuba

ABSTRACT

One of the recent challenges faced by High-Performance Computing (HPC) is how to apply Field-Programmable Gate Array (FPGA) technology to accelerate a next-generation supercomputer as an efficient method of achieving high performance and low power consumption. Graphics Processing Unit (GPU) is the most commonly used accelerator for HPC supported by regularly executed high degree of parallel operations which causes performance bottleneck in some cases. On the other hand, there are great opportunities to flexibly and efficiently utilize FPGAs in logic circuits to fit various computing situations. However, it is not easy for application developers to implement FPGA logic circuits for their applications and algorithms, which generally require complicated hardware logic descriptions. Because of the progress made in the FPGA development environment in recent years, the High-Level Synthesis (HLS) development environment using the OpenCL language has become popular. Based on our experience describing kernels using OpenCL, we found that a more aggressive programming strategy is necessary to realize true high performance based on a "co-design" concept to implement the necessary features and operations to fit the target application in an FPGA design. In this paper, we optimize the Authentic Radiation Transfer (ART) method on an FPGA using OpenCL. We also discuss a method to parallelize its computation in an FPGA and a method to optimize the OpenCL code on FPGAs. Using a co-designed method for the optimized programming of a specific application with OpenCL for an FPGA, we achieved a performance that is 6.9 times faster than that of a CPU implementation using OpenMP, and almost the same performance as a GPU implementation using CUDA. The ART code should work on a larger configuration with multiple FPGAs requiring interconnections between them. Considering the current advanced FPGAs with interconnection features, we believe that their parallelized implementation with multiple FPGAs will achieve a higher performance than GPU.

*Present affiliation is DOWANGO Co., Ltd.
†Present affiliation is Tohoku University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HEART 2018, June 20–22, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6542-0/18/06...$15.00
https://doi.org/10.1145/3241793.3241799

ACM Reference Format:
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 *The 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018), June 20–22, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3241793.3241799

1 INTRODUCTION

In recent years, heterogeneous systems with a CPU and an accelerator such as a GPU or an Intel Xeon Phi in the same node have been widely used for HPC applications. In these systems, CPUs are treated as latency-oriented processing units and accelerators are treated as throughput-oriented processing units because accelerators realize a high peak computational performance by utilizing multiple cores and wide Single Instruction Multiple Data (SIMD) units. In addition to these accelerators, an FPGA has become recognized as a device that is the middle of trade-off between CPUs and accelerators. However, it is difficult for software developers to develop computation logic using a traditional Hardware Description Language (HDL). On the other hand, an efficient performance per energy consumption in HPC has become a critical issue, along with the absolute computing performance. There is an acceleration limit when simply using the high degree of operation level parallelism provided by GPUs.

Because of the progress made in the FPGA development environment in recent years, the HLS environment has become popular. HLS is a technology that makes it possible to use a higher level language such as C, C++, or OpenCL for an FPGA's circuit. For example, Intel and Xilinx provide the OpenCL Software Development Kit (SDK) as an HLS solution. It supports not only circuit programming using the OpenCL language but also controlling FPGAs using the OpenCL standard APIs through PCIe from the host. We evaluated the fundamental functionality, performance, and productivity of the OpenCL SDK.

We selected a computational astrophysics application for practical use of OpenCL. The purpose of this research is to optimize the ART method on an FPGA using the Intel FPGA SDK for OpenCL. The ART method is an important algorithm used in the Accelerated Radiative transfer on Grids using Oct-Tree (ARGOT) program, which solves space radiative transfer problems to challenge the analysis of an early stage universe where the stars and galaxies were made as the first objects there. This paper discusses a method for parallelizing its computation in an FPGA and the optimizations used for OpenCL on an FPGA.

2 RELATED WORK

Some applications and benchmarks have been evaluated using OpenCL on FPGAs. [8] reported that OpenCL code originally made for GPUs had poor performance on FPGAs and OpenCL code must be optimized to target FPGA. [2] 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. [4] 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.

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 space radiative transfer code on an FPGA using OpenCL. Its algorithm is suitable for FPGAs because of its complex memory access pattern.

3 ARGOT: SPACE RADIATIVE TRANSFER CODE

ARGOT is an astrophysics code developed in the Center for Computational Sciences (CCS) at University of Tsukuba. It combines two algorithms to solve radiative transfer problems: the ARGOT algorithm [6], which computes the radiative transfer from point sources, and the ART algorithm [7], which computes the radiative transfer from sources spreading out in the target space. In this paper, we just focus on ART method as the fundamental one. The ART method is based on a ray-tracing method in a 3D space split into meshes. As shown in Figure 1, multiple incident rays come from a boundary and move in a straight direction parallel with each other, without any reflection or refraction. Each ray is shot from the center of a boundary mesh. A gas reaction is computed on a mesh for each passage of the ray. The direction (angle) of the ray is computed using the HEALPix algorithm [1]. 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.

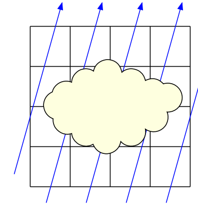

Figure 1: Ray tracing used in the ART Method. Blue arrows show rays. Yellow cloud shows a gas used to compute a reaction.

Because the ART method uses ray-tracing, the computation order in a ray must be sequential, while the computations for different rays can be performed in parallel because there is no computation dependency between any two rays. There are two problems in implementing the ART method on an SIMD-like architecture. First, the memory access pattern of the mesh data varies depending on the ray direction, which means hundreds or thousands of different patterns will be possible. In some cases, to compute multiple ray interactions in the SIMD manner, we have to access the mesh data in non-continuous locations in memory, which causes a low cache hit ration on the CPU and long latency in the GPU. Secondary, integrations on mesh data caused by two rays closed to each other will be conflicted. We have to use atomic operations or compute non-near rays in parallel to keep the computations correct. The former method degrades the computation performance while the latter method causes more scattered memory access patterns.

Because of this ART method'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.

4 INTEL FPGA SDK FOR OPENCL

4.1 Overview

Intel provides the Intel FPGA SDK for OpenCL for their FPGA products. This makes it possible to design an FPGA circuit using the OpenCL language. It provides an all-in-one development environment that contains an OpenCL to HDL compiler and an OpenCL runtime library for the host and driver to control an FPGA over a PCIe bus from the host. A design generated by the compiler contains not only circuits from OpenCL kernels but also peripheral controllers such as DDR memory and PCIe. Therefore, we can use an FPGA by writing OpenCL code and do not have to write any HDL code. We can manage FPGAs through the OpenCL standard's Application Programming Interfaces (APIs).

An HLS environment is necessary to open the FPGA use for a wide range of application programmers and to reduce the effort needed to program their codes. Therefore,

we need to find the advantages and disadvantages of the HLS environment for various applications.

4.2 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.

4.3 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.

5 ART ON FPGA IMPLEMENTATION

5.1 Implementation Overview

The basic strategy of our 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 2 shows the design outline of our implementation. The "memory reader" reads the mesh and ray data from the DDR memory, which is seen as a global memory in the OpenCL language. 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 method, and the array consists of multiple kernels. We show the details of the PE network in the next subsection.

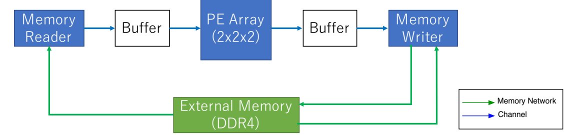

Figure 2: Design Outline of ART on FPGA.

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

Equation (1) shows the computation of the ART method. ν, I_ν^{in}, I_ν^{out}, $\hat{\boldsymbol{n}}$, $\Delta\tau$, and S_ν describe the frequency, the incoming radiative intensity, the outgoing radiative intensity, the direction of the ray, the optical depth on the mesh, and the source function of the mesh, respectively. The target space is divided into 3D meshes, and Equation (1) is computed on each mesh. All computations use single precision floating point and implemented by Digital Signal Processors (DSPs) in an FPGA including the exponent functions.

In our implementation, the "buffers" and "PE array" kernels shown in Figure 2 are marked as "autorun" kernels. The remaining kernels are regular kernels and controlled by the host. We cannot make them autorun kernels because of their global memory access. 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.

Because our implementation is a work-in-progress, it lacks some features from the CPU implementation. The CPU implementation supports inter-node parallelization using a Message Passing Interface (MPI), but the FPGA implementation in the current version does not support any networking functionality and uses only one FPGA.

5.2 Parallelization using Channel in FPGA

We next describe the structure of the 'PE array" shown in Figure 2. A PE array consists of PEs and Boundary Elements (BEs), as shown in Figure 3. A PE is a computation kernel for the ART method, while a BE is a unit for the boundary processing of ray tracing. Figure 3 shows a PE array network on the X and Y dimensions. The actual computation is performed on 3D space, but we just show 2D version for easy understanding.

In our implementation, the problem space for an FPGA is divided into small blocks, as shown in Figure 4. Each

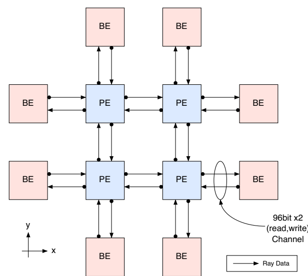

Figure 3: Network of PEs and BEs in X and Y dimensions.

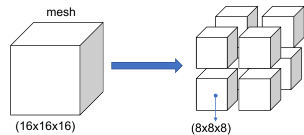

Figure 4: Decomposing large space into small ones.

PE usescthe ART method for a computation on every small block. They are connected by two channels and exchange ray data with each other. We need to have two channels to support bi-directional communication between two elements because a channel supports one-sided communication only. A BE handles the ray I/O to or from a neighbor boundary.

Generally speaking, the ___*local* memory access complexity in OpenCL has an impact on the kernel performance and resource usage of the kernel. We can expect four levels of complexity in OpenCL programming, as described below. If the compiler can analyze the memory access pattern at the compile time and the pattern is linear access in a loop, it will be stall-free and can read or write data on every clock cycle. If the pattern is random access or depends on runtime values, it may stall, resulting in a performance degradation because of stalls and increasing the resource usage. If the compiler cannot determine the memory access pattern statically at the compile time, the Initiation Interval (II) of a loop containing memory access becomes greater than one, and its throughput becomes poor. If the memory access is more complex than the previous case, the compilation time will also be very long, with more than hours to enlarge the design turn-around time.

The benefit of our design using channel parallelisms is that each PE has its own dedicated Block RAMs (BRAMs) declared as a ___*local* memory in the OpenCL language for its computation. The compiler can easily analyze the OpenCL source and generate the best stall-free design.

5.3 DDR Memory Access

The previous section described how the computation is performed on BRAMs distributed among PEs, which means our implementation can solve only small problems because the size of BRAM blocks in an FPGA is small. We consider an FPGA capable of solving a mesh with a size of at least 128^3 for problems solved by ARGOT. To obtain the best performance on BRAMs, the size of the mesh should be 64 bytes, including padding. Therefore, a 128 MB memory space is required for 128^3 meshes, which cannot be fit inside an FPGA.

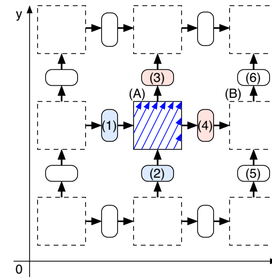

Figure 5: Ray buffer outline. Red boxes show the output ray buffers, blue boxes show the input ray buffers, and blue arrows show the rays to compute.

We divide the entire large ART computation into small steps based on the mesh size that can fit into the BRAMs in an FPGA. Once a small block is computed, the next small block will be computed sequentially. Figure 6 shows the pseudo code of the implementation using DDR memory. It only shows the algorithm, without showing the implementation in detail. The algorithm is divided into multiple kernels that are connected by channels, as described in the previous subsection.

We implement two kinds of kernels that access the DDR global memory to solve large problems. One of these is used to replace mesh data involved during the progression of the computation. The BRAMs in a PE for mesh data work like an addressable cache: mesh data are read from the DDR memory, stored into BRAMs (A in Figure 6), used in ART computations on the BRAMs (B in Figure 6; no DDR access in this phase), and written into the DDR memory (C in Figure 6).

The other DDR access modules are used for ray data. The memory access for ray data is more complex than that for mesh data. Ray buffers are allocated on the DDR memory to store ray data. They are not stored in BRAMs because they need larger size than that of BRAMs. Figure 5 shows a 2D simplified image of the memory access for ray data. The figure shows nine boxes, each of which shows a small block containing meshes. The central box with solid lines (A) shows the current small block computation (shown as variable b in Figure 6). Rays are read from two blue input buffers (1 and 2) and written into two red output buffers (3 and 4). The input and output buffers are changed depending on which small box is computed. If we compute on block B, buffers 4 and 5 are inputs, and buffer 6 is an output. There is no output buffer on the right hand side of block B because rays go outside of the domain and are discarded.

6 PERFORMANCE EVALUATION

6.1 Evaluation Environment

We used the Pre-PACS version X (PPX) system for the performance evaluation in this study. This system is a prototype

```
for dir in ray_directions[] {
 for b in small_blocks[] {
  A: mesh_load(b)
  for i in ipix(dir) {
   for iray in rays(ipix) {
    r = ray_load(iray)
    for m in compute_path(r) {
     C: compute reaction between r and m
    }
    ray_store(iray, r) }}
  B: mesh_store(b) }}
```

Figure 6: Pseudo code of DDR implementation.

Table 1: Evaluation Environment

CPU	Intel Xeon E5-2660 v4 × 2
CPU Memory	DDR4 2400 MHz 64 GB (8 GB × 8)
GPU	NVIDIA Tesla P100 (PCIe)
Host OS	CentOS 7.3
Host Compiler	gcc 4.8.5
OpenCL SDK	Intel FPGA SDK for OpenCL 16.1.2.203
CUDA	CUDA 8.0.61
FPGA	BittWare A10PL4 (10AX115N3F40E2SG)
FPGA Memory	DDR4 2133 MHz 8 GB (4 GB × 2)

system for the next generation of PACS series supercomputers at University of Tsukuba. Table 1 lists the environmental specifications of PPX. A CPU and an FPGA are connected by PCIe Gen.3 x8 lanes, while a CPU and a GPU are connected by PCIe Gen.3 x16 lanes. To avoid the performance degradation caused by a PCIe access over a Quick Path Interconnect (QPI), the closest CPU to the device is used for the FPGA and GPU evaluation.

We used a benchmark program that contained the core computation of the ART method extracted from the ARGOT program for the performance evaluation. The mesh data input to the benchmark are generated by a pseudo random generator, and the input ray data are generated by the HEALpix library in the same way as the ARGOT program because the computation cost of the ART method does not depend on mesh data but depends on the ray data.

The problem size used for the evaluation ranged from 16^3 to 128^3. The current 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. We do not measure the time for the data transfer between the host and the devices because they are the same for the FPGA and GPU (or advantageous for the GPU because of the larger lane size on PCIe). We use "mesh/s" as a unit for the performance evaluation. This indicates how many meshes are traced by rays in a second.

6.2 Resource Usage

Table 2 lists the resource usage values of FPGA implementations. ALMs and registers are fundamental elements used to represent logic circuits, M20K and MLAB are the distributed BRAM, and DSP is a block used for floating point number operations. "freq." means the operation frequency in the clock domain for OpenCL kernels.

There is a large difference in resource usage between the 16^3 implementation and the 32^3 implementation. This difference is caused by the DDR control kernels described in section 5.3. They are not necessary in the 16^3 implementation because the problem size is small, and all of the meshes can be stored in an FPGA's BRAM.

As listed in the table, ALMs and registers are the greatest resource users in this design, with 40% of the total ALMs and registers used, which becomes a bottleneck when attempting to increase performance. To achieve the best performance, it is desirable to use all of the DSP blocks in an FPGA to compute the ART method by increasing the number of PEs. From the DSP point of view, we expect to implement 32 PEs using 84% of the DPSs. However, because 40% of the ALMs and registers are used with an 8 PE design, we cannot implement enough PEs to utilize all of the DPSs. We have to optimize the OpenCL code to decrease the resource usage in the design.

The frequency differs between the 32^3, 64^3, and 128^3 cases by the OpenCL compiler. These source codes are almost the same, but there are slight differences in the constants such as the loop count or problem size. Verilog HDL codes generated by the OpenCL compiler is not human readable, and it is too difficult for us to understand how kernels are implemented as circuits.

6.3 Performance Evaluation

Table 3 and Figure 7 show the performance comparison between the CPU, GPU, and FPGA implementations. The CPU implementation is written in C and uses OpenMP for the thread parallelization. "CPU(14C)" represents the CPU implementation running with 14 OpenMP threads and "CPU(28C)" represents the double count of threads. The evaluation environment has two Xeon CPUs, and each CPU has 14 cores. where these two cases correspond to single or double CPU sockets. The GPU implementation is based on the CPU implementation but written in CUDA.

The FPGA implementation achieves 1283 M mesh/s, 1165 M mesh/s, 1111 M mesh/s and 1134 M mesh/s on 16^3, 32^3, 64^3 and 128^3, respectively. The CPU implementation is the slowest and 6.8 times slower than the GPU implementation and 6.9 times slower than the FPGA implementation. Although the fastest problem size for a CPU is 64^3, it is still slower than either the FPGA or GPU. We believe that the performance degradation from 64^3 to 128^3 is caused by decreasing the cache hit rate.

The 16^3 case on the GPU is more than 10 times slower than the 64^3 and 128^3 sizes. Their problem sizes are too small

Table 2: Resource usage and clock frequency of implementation.

size	# of PEs	ALMs	(%)	Registers	(%)	M20K	(%)	MLAB	DSP	(%)	Freq. [MHz]
(16, 16, 16)	(2, 2, 2)	132,283	31%	267,828	31%	739	27%	14,310	312	21%	193.2
(32, 32, 32)	(2, 2, 2)	169,882	40%	344,447	40%	796	29%	21,100	312	21%	173.8
(64, 64, 64)	(2, 2, 2)	169,549	40%	344,512	40%	796	29%	21,250	312	21%	167.0
(128, 128, 128)	(2, 2, 2)	169,662	40%	344,505	40%	796	29%	21,250	312	21%	170.4

Table 3: Performance comparison between FPGA, CPU, and GPU implementations. The unit is M mesh/s.

Size	CPU(14C)	CPU(28C)	P100	FPGA
(16,16,16)	112.4	77.2	105.3	1282.8
(32,32,32)	158.9	183.4	490.4	1165.2
(64,64,64)	175.0	227.2	1041.4	1111.0
(128,128,128)	95.4	165.0	1116.1	1133.5

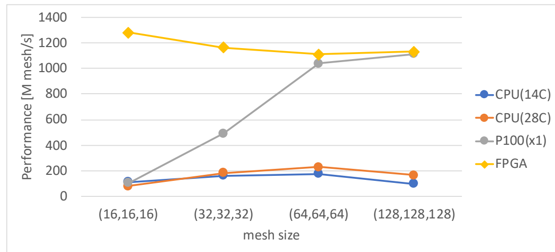

Figure 7: Performance comparison between FPGA, CPU, and GPU implementations.

for a GPU because of the insufficient parallelism for its 3584 CUDA cores.

Unlike the GPU implementation, the FPGA implementation's performance is high for all problem sizes. We believe that this characteristic of the FPGA comes from its pipeline design. Because the performances per frequency for each problem size are almost the same ($6.64 \sim 6.70$ mesh per cycle), the performance differences on the 16^3, 32^3, 64^3, and 128^3 problems are caused by the frequency differences. Therefore, optimization to increase the kernel frequency is important. The mesh data input and output are the dominant memory loads of the ART method. The current implementation (8 PEs) is computation bound. However, if more PEs are used, it will be memory bound. Even on the large problem sizes, the FPGA implementation has almost the same performance as GPU. Therefore, we consider the fundamental performance of the FPGA implementation to be adequate compared to a GPU.

7 NEXT STEP: PARALLELIZED FPGA

We confirmed that the FPGA performance was adequate compared to a GPU. To increase the problem size and improve the performance, we are planning to add networking functionality to our ART implementation on the FPGA for parallelization. In the GPU programming model, the GPU is a sort of slave device that can only be controlled by the host CPU, including the inter-node communication. Because inter-node communications are also initiated by the CPU, the CPU and the GPU must be synchronized before the communication starts. If the NVIDIA GPU is combined with the Mellanox HCA in a node, GPUDirect for RDMA (GDR) [5] can be used as an improved communication protocol between nodes. This would reduce the communication latency and increase its throughput. However, even if we can use GDR, communications are initiated by the CPU, and the communication overhead is not negligible when it occurs frequently.

On the other hand, recent advanced FPGAs such as Intel's Stratix 10 are equipped with multiple very high speed (up to 100 Gbps per link) interconnection links (up to 4 channels). Additionally, an HLS such as the OpenCL programming environment is provided, and there are several types of research to involve them in FPGA computing. In [3], we showed the basic feature needed to utilize a high speed interconnection over an FPGA driven by OpenCL kernels. Therefore, even though the performance of our implementation is almost the same as that of the NVIDIA P100 GPU, the overall performance with multiple computation nodes and a directly connected FPGAs can easily overcome the deficiencies of the GPU implementation, which requires host CPU control and kernel switching for inter-node communication. The networking overhead on FPGAs is much lower than that on GPUs. Improving the current ART method implementation with such an interconnection feature on an FPGA is our next step toward high-performance parallel FPGA computing.

8 CONCLUSION

In this study, we optimized the ART method used in an ARGOT program to solve a fundamental calculation in the early stage universe with a space radiative transfer phenomenon, on an FPGA using Intel's FPGA SDK for OpenCL. We parallelized the algorithm using the SDK's channel extension in an FPGA. We achieved a performance that was 4.89 times faster than the CPU implementation using OpenMP, and almost the same performance as the GPU implementation using CUDA. It was important to take the FPGA architecture into account, even though we used OpenCL, which was the language for the software. Our implementation consisted of multiple kernels (PEs, BEs, and so on), which came from the FPGA architecture with distributed block memories in a chip.

Although the performance of the FPGA implementation was comparable to that of the NVIDIA P100 GPU, there is room to improve its performance. The most important optimization is resource optimization. The performance could be improved by implementing a larger number of PEs than

in the current implementation. However, it would be difficult to reduce the use of ALMs and registers because they are not directly described in the OpenCL code. Both the resources and frequency are important. We suppose that Arria 10 with an OpenCL design can run at a frequency of 200 MHz or higher. The frequency is restricted by the loop structure in OpenCL. However, we have no method for fine tuning because the OpenCL compiler completely hides the logic used by loops from the user.

We will implement the network functionality in the ART design to parallelize it among multiple FPGAs. We consider networking using FPGAs to be an important feature for parallel applications using FPGAs. Although GPUs have higher computation performance FLOPS and higher memory bandwidth than FPGAs, I/O, including networking, is a weak point for GPUs because they are connected to NICs through a PCIe bus. In addition to networking, we will attempt to run our code on a Stratix 10 FPGA, which is the next generation Intel FPGA. We expect to be able to implement more PEs than the Arria 10 FPGA because it has 2.2 times more ALM blocks and 3.8 times more DPS blocks.

ACKNOWLEDGEMENTS

This research is a part of the project titled “Development of Computing-Communication Unified Supercomputer in Next Generation” under the program of “Research and Development for Next-Generation Supercomputing Technology” by MEXT. We thank Intel University Program for providing us both of hardware and software.

REFERENCES

[1] K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. Healpix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

[2] K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193, July 2015.

[3] R. Kobayashi, Y. Oobata, N. Fujita, Y. Yamaguchi, and T. Boku. Opencl-ready high speed fpga network for reconfigurable high performance computing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 192–201, New York, NY, USA, 2018. ACM.

[4] Y. Luo, X. Wen, K. Yoshii, S. Ogrenci-Memik, G. Memik, H. Finkel, and F. Cappello. Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017.

[5] NVIDIA Corporation. GPUDirect for RDMA, https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[6] T. Okamoto, K. Yoshikawa, and M. Umemura. argot: accelerated radiative transfer on grids using oct-tree. *Monthly Notices of the Royal Astronomical Society*, 419(4):2855–2866, 2012.

[7] S. Tanaka, K. Yoshikawa, T. Okamoto, and K. Hasegawa. A new ray-tracing scheme for 3d diffuse radiation transfer on highly parallel architectures. *Publications of the Astronomical Society of Japan*, 67(4):62, 2015.

[8] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’16, pages 35:1–35:12, Piscataway, NJ, USA, 2016. IEEE Press.