Multi-hetero Acceleration by GPU and FPGA for Astrophysics Simulation on oneAPI Environment

Ryuta Kashino
Master's Program in Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba, Ibaraki, Japan
kashino@hpcs.cs.tsukuba.ac.jp

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

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

ABSTRACT

GPU (Graphics Processing Unit) computing is one of the most popular accelerating methods for various high-performance computing applications. For scientific computations based on multi-physical phenomena, however, a single device solution on a GPU is insufficient, where the single timescale or degree of parallelism is not simply supported by a simple GPU-only solution. We have been researching a combination of a GPU and FPGA (Field Programmable Gate Array) for such complex physical simulations. The most challenging issue is how to program these multiple devices using a single code.

OneAPI, recently provided by Intel, is a programming paradigm supporting such a solution on a single language platform using DPC++ based on SYCL 2020. However, there are no practical applications utilizing its full features or supporting heterogeneous multi-device programming to demonstrate its potential capability. In this study, we present the implementation and performance evaluation of our astrophysics code ARGOT used to apply the oneAPI solution with a GPU and an FPGA. To realize our concept of Cooperative Heterogeneous Acceleration by Reconfigurable Multidevices, also known as CHARM, as a type of next-generation accelerated supercomputing for complex multi-physical simulations, this study was conducted on our multi-heterogeneous accelerated cluster machine running at the University of Tsukuba.

Through the research, we found that current oneAPI framework is effective not only for its typical programming by DPC++ but also for utilizing traditionally developed applications coded by several other languages such as CUDA or OpenCL to support multiple types of accelerators. As an example of real application, we successfully implemented and executed an early stage universe simulation by fundamental astrophysics code to utilize both GPU and FPGA effectively. In this paper, we demonstrate the actual procedure for this method to program multi-device acceleration over oneAPI.

This work is licensed under a Creative Commons Attribution International 4.0 License.

HPC Asia2022, January 12–14, 2022, Virtual Event, Japan
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8498-8/22/01.
https://doi.org/10.1145/3492805.3492817

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**.

KEYWORDS

GPU, FPGA, Multi-hetero Acceleration, Intel oneAPI

ACM Reference Format:
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 (HPC Asia2022), January 12–14, 2022, Virtual Event, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3492805.3492817

1 INTRODUCTION

Accelerators are one of the most important technologies driving modern HPC (High-Performance Computing) from the perspective of performance and power-consumption. They are particularly important on large-scale systems with a number of computation nodes, and have been top machines on the TOP500 List [9]. Thus far, the GPU (Graphics Processing unit) has been the most representative accelerator owing to its SIMD base operation supporting a large FLOPS under a simple operation, as well as its high affinity to high-performance memory such as HBM2 based on its type of calculation operation. NVIDIA Tesla A100 [5] provides a theoretical peak performance of up to 9.7 TFLOPS with double precision (FP64) and 2.039 TB/s of theoretical memory bandwidth. Although recent GPUs have mainly focused on AI or deep learning applications, they are still the strongest accelerators used in traditional HPC applications such as climate, biomedical, astrophysics, and quantum physics modeling.

In addition to the excellent features for HPC, however, GPU computing is based on a relatively simple execution model with SIMD-style computations where a large number of regular computations are required to utilize its high performance potential. For instance, most of the Linpack benchmark results for large-scale GPU clusters have achieved approximately 50% of the theoretical

Table 1: Acceleration devices

Device	GPU	FPGA
Parallelism	SIMD	Pipeline
Memory	Strong (large HBM2)	Weak (DDR and small HBM)
Conditional branch	Weak (true/false part run sequentially)	Strong (true/false part run in parallel)
Lower Parallelism	Weak (most of core goes to rest)	Strong
Inter-node Communication	Weak (by CPU interconnect)	Strong (own optical link)

peak performance because the linear algebraic computations applied are quite suitable for the nature of GPU computing. However, achieving an actual sustained performance on real applications is difficult owing to various limitations, including the following:

- an insufficient number of floating-point operations in the complete codes,
- irregular computational patterns inhibited based on the branch conditions,
- or frequent communications between computational nodes, forcing a switching between the CPU and GPU execution.

In addition to their relatively low programmability, these are important issues in the application of GPUs for various applications.

By contrast, the FPGA (Field Programmable Gate Array) is attractive as a new type of accelerator for HPC applications, and is a fully reconfigurable processor allowing a rewriting of the inner circuit according to the application. The benefits of this device are as follows:

- It is optimizable for computational applications.
- It is vertically parallelizable in a pipeline manner.
- The optimized performance based on the power consumption can exceed that of a GPU.
- Recent high-end FPGAs enable a direct communication between FPGA devices using high-speed optical links.

However, an FPGA also has several disadvantages for HPC use, including the following:

- The HDL (Hardware Description Language) is difficult to program for general application users.
- A long compilation time (usually from several to more than 10 h) prevents the productivity of the codes.
- Hardware resources (logic elements and memory devices) are limited because they cannot be reused in different locations within a large code unlike with an ordinary CPU or GPU.

The recent high-end FPGA for HPC provides much larger hardware resources and memory capacity, which decreases some of the weaknesses described above.

Herein, we focus on the complementary characteristics of both a GPU and an FPGA (Table 1). Thus, we have been researching ways to couple these devices together on a computational node for use in a large-scale cluster system, achieving a 360-degree type system that will allow each device to compensate the other for various types of HPC computing. We have been studying this concept, which we call CHARM (Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices), as shown inFig. 1. Accelerators such as GPUs are used for coarse-grained parallel applications, whereas multiple FPGAs connected by a high-speed interconnect

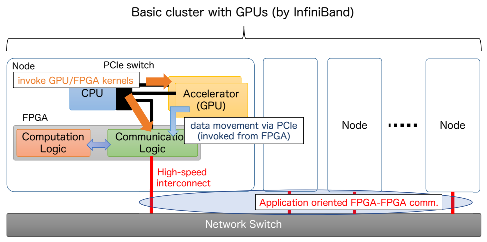


Figure 1: CHARM: Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices

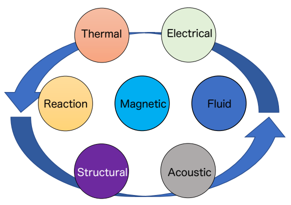


Figure 2: Complicated simulation with multi-physical phenomena

autonomously conduct communication and computations in areas where CPUs/GPUs are weak. We consider this concept to be particularly applicable to complicated computational simulations based on multi-physical phenomena. In advanced physics and chemistry, the target physics is not simple and we need to combine several physical phenomena in the target problem, as shown in Fig. 2.

The platform used for our concept is based on a complex computational node with CPUs, GPUs, and FPGAs connected by an intra-node PCIe switch [1]. Recent CPUs are equipped with a number of PCIe lanes (gen3 or gen4), where multiple devices can be easily connected. However, we need additional FPGAs along with multiple GPUs on a single node. Thus, we may need external PCIe switches to sufficiently enlarge the number of PCIe lanes and support the combined use of multiple GPUs and FPGAs. Of course, another set of PCIe lanes is required for an interconnection network such as InfiniBand.

Although the hardware construction is relatively easy owing to the generality of PCIe to connect all devices, the program coding is quite difficult for application users. We have been constructing a uniform programming environment to apply OpenACC [8] to cover both types of accelerators with multiple background compilers [20]. Another recent approach has been the use of oneAPI [6], provided by Intel, for supporting multiple accelerators in a single programming framework, including the operation control over multiple devices and the data transfer among such devices with interoperability. In the oneAPI framework, for applicability to various accelerating devices including a GPU or an FPGA, a code is written in DPC++ based on SYCL 2020.

However, there is a large stock of codes by different languages, particularly for GPUs, such as CUDA (C/C++), CUDA Fortran, OpenCL, or OpenMP. OneAPI is also designed to involve such codes described in different languages to invoke appropriate backend compilers. Moreover, for certain hardware platforms, it is possible to

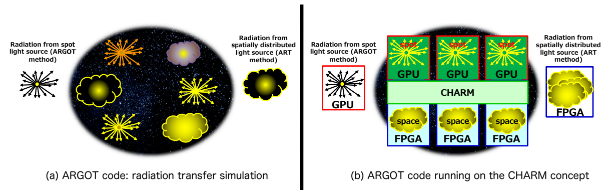

Radiation from spot light source (ARGOT method)
Radiation from spatially distributed light source (ART method)
(a) ARGOT code: radiation transfer simulation
Radiation from spot light source (ARGOT method)
GPU
spot
GPU
spot
GPU
spot
GPU
CHARM
space
FPGA
space
FPGA
space
FPGA
Radiation from spatially distributed light source (ART method)
FPGA
(b) ARGOT code running on the CHARM concept

Figure 3: (a) Overview of the ARGOT code and (b) how to accelerate it using the CHARM concept

even include compiled binary codes in the oneAPI framework. We focus on this oneAPI feature of utilizing traditional codes written for multiple devices to replace the complicated coding in a single manner for computation kernel execution, event queuing, data movement, and total control under the host CPU. In this study, we describe how to implement our CHARM concept on the oneAPI framework and apply it to our already running code for computational astrophysics.

2 ARGOT: RADIATIVE TRANSFER SIMULATION CODE FOR ASTROPHYSICS

ARGOT is an astrophysics simulation code developed in our organization for the simulation of how the first objects were generated in the early stage of the universe. As shown in Fig. 3 (a), two algorithms are combined to solve the radiative transfer problems: the ARGOT algorithm [17], which computes the radiative transfer from the point sources, and the ART algorithm [18], which computes the radiative transfer from sources spreading out within the target space. To accelerate the ARGOT code, we run the ARGOT algorithm on GPUs and run the ART algorithm on FPGAs separately, as shown in Fig. 3 (b). In the next section, we provide a brief description of the two algorithms.

2.1 ARGOT algorithm

To solve the radiative transfer from point radiation sources, a computation of the optical depth between each pair of a point radiation source and a target mesh grid, that is, the end point of each light ray (see Fig. 4 (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. 4 (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, the 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, for example, a target mesh grid in Fig. 4 (b), the photon flux coming from each radiation source at the target mesh grids is given by

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

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

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

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

Fig. 4 (c) shows how to parallelize the ARGOT algorithm. This parallelized technique is called "Node Parallelization." It decomposes the simulation volume evenly along each direction. Light rays are divided by boundaries of parallel domains into several "ray segments," and a computation of the optical depths of the assigned ray segments in each parallel domain is conducted. In the GPU simulations, this computation was conducted concurrently on the GPUs. Subsequently, a sum reduction of the optical depths of each ray segment to their target mesh grids was applied. However, in this study, the ARGOT code runs on a single node, and we do not use this parallelization technique. In this 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 each computation is then conducted in parallel.

2.2 ART algorithm

To solve the radiation transfer from spatially diffuse sources, the ART algorithm is used, which 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; thus, accelerating the ART algorithm directly results in the performance improvement of the ARGOT code. As shown in Fig. 5, multiple incident rays come from a boundary and move in a straight direction parallel to 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 the computational volume, using the following equation:

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

This calculation is conducted every time the ray passes through a mesh grid. For a given incoming radiation intensity I_ν^{in} along the direction $\hat{\boldsymbol{n}}$, 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 the absorption coefficient of the mesh grid, respectively. The direction (angle) of the ray was computed using the HEALPix algorithm [13]. 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. The number will be at least 768, for which the resolution parameter $N_{side} = 8$ in HEALPix.

Because the ART method uses ray tracing, the computational order within a 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 the ART method on a SIMD-like architecture is problematic in two ways.

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

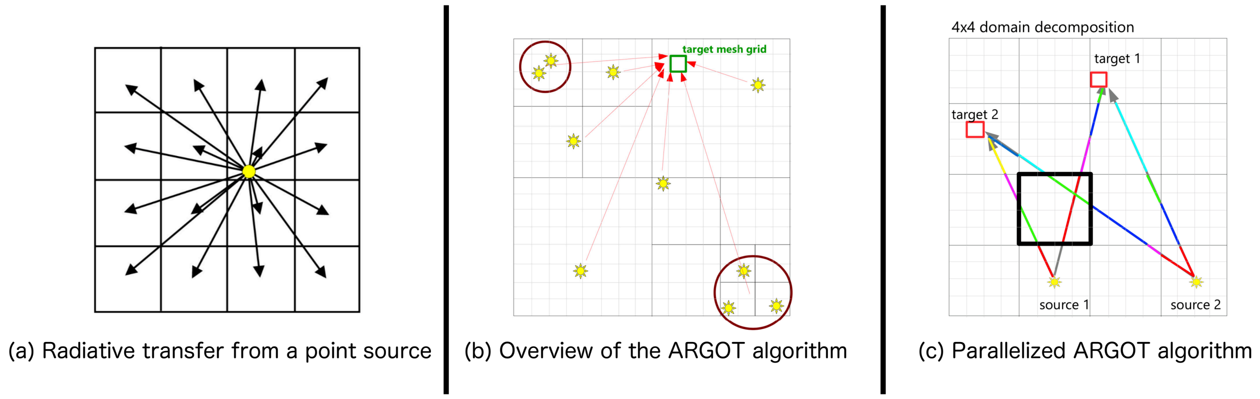

(a) Radiative transfer from a point source
target mesh grid
(b) Overview of the ARGOT algorithm
4x4 domain decomposition
target 1
target 2
source 1
source 2
(c) Parallelized ARGOT algorithm

Figure 4: (a) Schematic illustration of the ray-tracing method for the radiation emitted by a point radiation source in the two-dimensional mesh grids, (b) a way to solve it using the ARGOT algorithm, and (c) the parallelized ARGOT algorithm

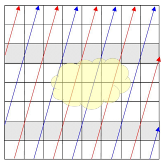


Figure 5: Ray tracing method used in the ART method. The arrows and yellow clouds show the rays and gas used to compute the reactions, respectively.

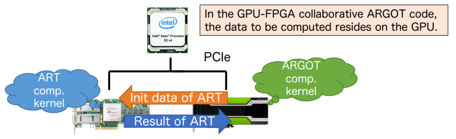

In the GPU-FPGA collaborative ARGOT code, the data to be computed resides on the GPU.
PCIe
ART comp. kernel
ARGOT comp. kernel
Init data of ART
Result of ART

Figure 6: Overview of the implementation of the GPU–FPGA-accelerated ARGOT code

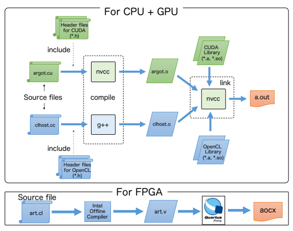


Figure 7: Compilation flow of the GPU–FPGA-accelerated ARGOT code

ray interactions in a 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 a long latency in the GPU.

Second, the integration of mesh data resulting from two rays that are close to each other will cause a conflict. When multiple light rays pass through shaded mesh grids, as shown in Fig. 5, 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 the processing speed may be significantly reduced. 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 [18] does not compute the neighboring rays simultaneously, which means that ray tracing along the red and blue light rays is separately performed as shown in Fig. 5. However, this method further exacerbates the memory access problems in the ART algorithm described for the first reason because this method causes the memory access patterns to become more scattered. This overhead is expected to be nearly cubically proportional, and close to the number of atomic operations.

Given the characteristics of the ART algorithm, we consider that SIMD-style processors such as CPUs and GPUs are unsuitable for this algorithm. By contrast, FPGAs can access on-chip internal memory with low latency, and high bandwidth for random access. In addition to its performance, we can program memory access patterns as part of the FPGA hardware. Therefore, we consider the use of the ART method on an FPGA to be suitable. A previous study implemented an FPGA-based hardware engine with an OpenCL programming framework [12], and we integrated the engine into the ARGOT code. Its implementation is described in the next section.

Please note that, although the ART algorithm is based on a ray-tracing algorithm, it is essentially different from that of CG (Computer Graphics), which can be accelerated through NVIDIA Turing architecture-based GPUs. The raytracing of the CG retroactively calculates the light reflection and transmission on the object surface from the viewpoint of the observer. By contrast, the ART algorithm calculates the radiation intensity every time a ray is passed through a mesh grid, and takes the average intensity by calculating the radiation intensity in each ray direction. In short, their only common point is a “raytracing.”

2.3 GPU–FPGA-accelerated ARGOT code

An overview of the implementation of the ARGOT code with the GPU and FPGA [14] is shown in Fig. 6. The existing GPU implementation of the ARGOT code enables both algorithms to execute on the GPU. Based on the GPU implementation of the ARGOT code, we replaced the implementation of the ART algorithm on the GPU by implementing the algorithm on the FPGA. This necessitated an appropriate data transfer between the GPU and FPGA because the ART algorithm initially generates data on the GPU, which are then sent to the FPGA. Ultimately, the resulting data processed by ART are sent back to the GPU.

The ARGOT code was implemented using multilingual programming composed of CUDA and OpenCL; therefore, a separate compilation was necessary. Fig. 7 shows the flow of the compilation. The CUDA code and OpenCL host code were compiled usingnvcc and g++, respectively, and the generated object files were linked using the `nvcc` to generate an ELF (Executable and Linkable Format) file. As previously described, the OpenCL kernel code for the ART algorithm was compiled offline using an Intel FPGA OpenCL compiler.

However, the implementation cost of such an application executing cooperative GPU-FPGA computations using APIs of various programming languages is extremely high and the source code is cumbersome, which is likely to compromise the maintainability and availability of the application code. Therefore, in this paper, we report the implementation and performance evaluation of a GPU–FPGA-accelerated ARGOT code using the Intel oneAPI as one of the methodologies for realizing a GPU-FPGA cooperative computation with as little effort as possible for application users.

3 GPU–FPGA-ACCELERATED ARGOT CODE WITH INTEL ONEAPI

3.1 Intel oneAPI

Intel oneAPI [6] is a cross-architecture programming framework proposed by Intel®. The purpose of oneAPI is to simplify the development across different architectures. As shown in Fig. 8, oneAPI includes the DPC++ programming language and a set of libraries, enabling development in a single language and API.

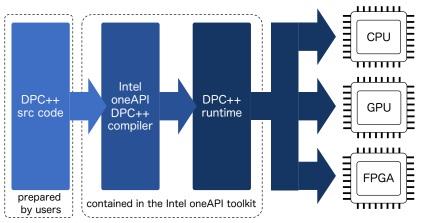


Figure 8: Intel oneAPI programming model.

In oneAPI programming, the `queue` class is used as an interface to conduct operations on accelerator devices. This operation is called an action, and typical actions include a kernel launch (`parallel_for` or `single_task`) and an explicit data transfer (`memcpy`). When a `queue` is created, it is always associated with a single accelerator device, and the actions submitted to the `queue` are executed on the associated device, such as a GPU or an FPGA. To handle both the GPU and FPGA simultaneously, it is necessary to prepare a queue for the GPU and a queue for the FPGA, and then submit actions to each queue. The devices officially supported by oneAPI are listed in [4]. Although NVIDIA GPUs can also be used, they are currently in the experimental stage[2].

3.2 CHARM on Intel oneAPI

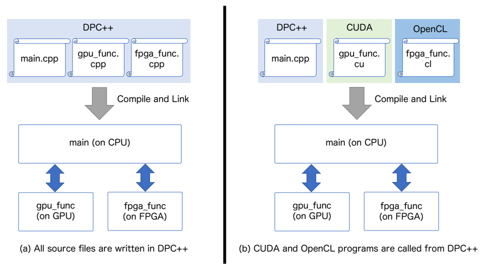


Figure 9: Two approaches to realizing CHARM with Intel oneAPI.

As shown in Fig. 9, there are two approaches to realizing CHARM with oneAPI: (a) writing all device programs in DPC++ or (b) invoking existing CUDA and OpenCL kernels from the DPC++ programming framework.

The former approach is recommended by Intel®, and we confirmed that it is possible to make an Intel CPU, NVIDIA GPU, and Intel FPGA work together using this approach. However, to execute existing applications with this approach, it is necessary to rewrite the entire source code in DPC++. This would impose a heavy burden on application users because most existing HPC applications, including the ARGOT code targeted in this study, are actually implemented in CUDA, being that they are intended to be executed on NVIDIA GPUs.

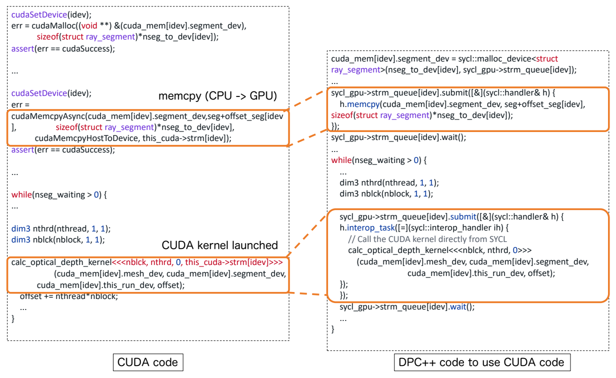
Therefore, we focus on approach (b), which reduces the programming effort as much as possible for application users by exploiting the existing CUDA and OpenCL code without modification, and by using DPC++ for their cooperative operation. It is possible to call CUDA and OpenCL kernels from DPC++ by utilizing `interop_task` (an extension of DPC++) for the CUDA kernels, and to construct DCP++ objects from OpenCL objects for OpenCL kernels. Using these features, we propose a methodology for realizing a GPU-FPGA-accelerated ARGOT code using oneAPI.

We believe that this study is valuable as a guideline for application users who are required to accelerate their already implemented HPC applications with oneAPI. This is because, by making the GPU and FPGA work collaboratively under oneAPI, it is possible to use a unified API (i.e., queue) to launch the kernel of each device and to synchronize and mediate between the two devices, thus maintaining a high maintainability and the availability of the application code. Although OpenCL is the same as traditional OpenCL programming in terms of managing different devices with a unified queue, OpenCL still requires application users to rewrite all applications they want to run on GPUs and FPGAs in OpenCL, which is a burden for application users. However, with oneAPI, as mentioned earlier, the CUDA and OpenCL codes developed thus far can be directly used and absorbed by DPC++. If the performance loss of the CUDA and OpenCL kernels absorbed by DPC++ is acceptable, it would be beneficial to application users.

In the following sections, to realize the CPU-FPGA-accelerated ARGOT code with oneAPI, we describe how to call the ARGOT and ART methods, which are major arithmetic components of the ARGOT code, as described in the previous section.

3.3 ARGOT method called in DPC++

The ARGOT method is implemented by the CUDA kernel and computed using a GPU. This CUDA kernel is called by SYCL API/DPC++. To invoke the CUDA kernel from DPC++, we use "DPC++ for CUDA" (see [3]), which provides support for Nvidia GPUs. An example of this code snippet is shown in Fig. 10, which is a part of the code used to calculate the optical depth of the ARGOT method (Equation (2)).



```
cudaSetDevice(idev);
err = cudaMalloc((void **) &(cuda_mem[idev].segment_dev),
        sizeof(struct ray_segment)*nseg_to_dev[idev]);
assert(err == cudaSuccess);

...

cudaSetDevice(idev);
err =
cudaMemcpyAsync(cuda_mem[idev].segment_dev,seg+offset_seg[idev
],              sizeof(struct ray_segment)*nseg_to_dev[idev],
         cudaMemcpyHostToDevice, this_cuda->strm[idev]);
assert(err == cudaSuccess);

...

while(nseg_waiting > 0) {
...

dim3 nthrd(nthread, 1, 1);
dim3 nblck(nblock, 1, 1);

calc_optical_depth_kernel<<<nblck, nthrd, 0, this_cuda->strm[idev]>>>
                (cuda_mem[idev].mesh_dev, cuda_mem[idev].segment_dev,
         cuda_mem[idev].this_run_dev, offset);
   offset += nthread*nblock;
   ...
}
```

```
cuda_mem[idev].segment_dev = sycl::malloc_device<struct
ray_segment>(nseg_to_dev[idev], sycl_gpu->strm_queue[idev]);
...
sycl_gpu->strm_queue[idev].submit([&](sycl::handler& h) {
   h.memcpy(cuda_mem[idev].segment_dev, seg+offset_seg[idev],
sizeof(struct ray_segment)*nseg_to_dev[idev]);
});
sycl_gpu->strm_queue[idev].wait();
...
while(nseg_waiting > 0) {
   ...
   dim3 nthrd(nthread, 1, 1);
   dim3 nblck(nblock, 1, 1);

   sycl_gpu->strm_queue[idev].submit([&](sycl::handler& h) {
   h.interop_task([=](sycl::interop_handler ih) {
     // Call the CUDA kernel directly from SYCL
     calc_optical_depth_kernel<<<nblck, nthrd, 0>>>
       (cuda_mem[idev].mesh_dev, cuda_mem[idev].segment_dev,
                    cuda_mem[idev].this_run_dev, offset);
   });
   });
   sycl_gpu->strm_queue[idev].wait();
   ...
}
```

Figure 10: Code snippet to calculate the optical depth of the ARGOT method (Equation (2).

First, to use DPC++ for CUDA, it is necessary to include the <CL/sycl/backend/cuda.hpp> header file. By reading this header file, it is possible to conduct operations equivalent to the CUDA API from oneAPI. In Fig. 10, the orange line in the upper part corresponds to the data transfer from the CPU and GPU. The CUDA kernel is called in the orange line at the bottom. To invoke the CUDA kernel, DPC++ for CUDA provides a function called `interop_task`. This function allows us to call the existing CUDA kernel directly.

3.4 ART method called in DPC++

In our previous study [14], we used an Intel Arria 10 FPGA as the accelerator to run the ART method; however, in this study, we used the more powerful Intel Stratix 10 FPGA. We also proposed an OpenCL implementation of the ART method for the Intel Stratix 10 FPGA [11], and used this OpenCL kernel code through the DPC++ APIs, as shown in Fig. 11.

As shown in the figure, the hardware accelerator for the ART method implemented in the FPGA consists of an engine that executes the calculation of the ART method and a handler that is responsible for loading mesh data from the external memory into the engine and writing the calculation results back to the external

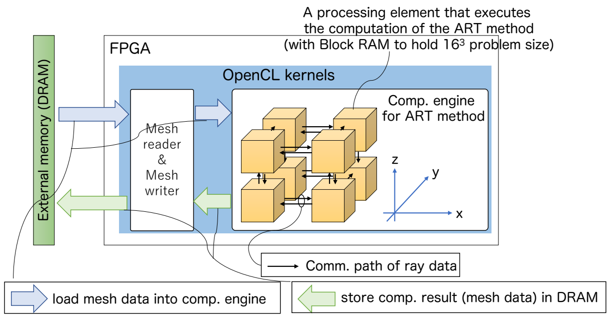
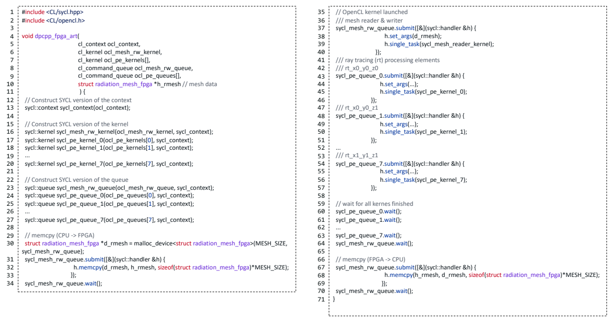

A processing element that executes the computation of the ART method (with Block RAM to hold 16^3 problem size)
FPGA
External memory (DRAM)
OpenCL kernels
Mesh reader & Mesh writer
Comp. engine for ART method
z
y
x
Comm. path of ray data
load mesh data into comp. engine
store comp. result (mesh data) in DRAM

Figure 11: OpenCL implementation of the ART method for Intel Stratix 10 FPGA [11].

memory, which are all implemented in OpenCL. The computing engine is composed of PEs (Processing Elements), which are computing cores connected in a three-dimensional ($2 \times 2 \times 2$) manner. In other words, the problem space for which a single FPGA is responsible is divided into smaller blocks and assigned to each PE to perform parallel computations. The divided problem space is stored in a working memory (such as a scratchpad memory) implemented using block BRAMs (Block RAMs) in each PE, for which each PE holds 16^3 problem sizes (32^3 problem sizes are held in a single FPGA). Each PE then communicates the ray data to the other PEs. The PE that receives the ray data then executes the arithmetic kernel of the ART method and sends the ray data reflecting the result of the arithmetic kernel to the PE in charge of the next problem space located in the ray direction, thereby realizing the ray tracing algorithm of the ART method. The execution result, which is the mesh data affected by the radiative transfer, is then written back to the external memory through the handler.



```
 1  #include <CL/sycl.hpp>
 2  #include <CL/opencl.h>
 3
 4  void dpcpp_fpga_art(
 5                      cl_context ocl_context,
 6                      cl_kernel ocl_mesh_rw_kernel,
 7                      cl_kernel ocl_pe_kernels[],
 8                      cl_command_queue ocl_mesh_rw_queue,
 9                      cl_command_queue ocl_pe_queues[],
10                      struct radiation_mesh_fpga *h_rmesh // mesh data
11                       ) {
12   // Construct SYCL version of the context
13   sycl::context sycl_context(ocl_context);
14
15   // Construct SYCL version of the kernel
16   sycl::kernel sycl_mesh_rw_kernel(ocl_mesh_rw_kernel, sycl_context);
17   sycl::kernel sycl_pe_kernel_0(ocl_pe_kernels[0], sycl_context);
18   sycl::kernel sycl_pe_kernel_1(ocl_pe_kernels[1], sycl_context);
19   ...
20   sycl::kernel sycl_pe_kernel_7(ocl_pe_kernels[7], sycl_context);
21
22   // Construct SYCL version of the queue
23   sycl::queue sycl_mesh_rw_queue(ocl_mesh_rw_queue, sycl_context);
24   sycl::queue sycl_pe_queue_0(ocl_pe_queues[0], sycl_context);
25   sycl::queue sycl_pe_queue_1(ocl_pe_queues[1], sycl_context);
26   ...
27   sycl::queue sycl_pe_queue_7(ocl_pe_queues[7], sycl_context);
28
29   // memcpy (CPU -> FPGA)
30   struct radiation_mesh_fpga *d_rmesh = malloc_device<struct radiation_mesh_fpga>(MESH_SIZE,
    sycl_mesh_rw_queue);
31   sycl_mesh_rw_queue.submit([&](sycl::handler &h) {
32                     h.memcpy(d_rmesh, h_rmesh, sizeof(struct radiation_mesh_fpga)*MESH_SIZE);
33                    });
34   sycl_mesh_rw_queue.wait();
```

```
35   // OpenCL kernel launched
36   /// mesh reader & writer
37   sycl_mesh_rw_queue.submit([&](sycl::handler &h) {
38                     h.set_args(d_rmesh);
39                     h.single_task(sycl_mesh_reader_kernel);
40                   });
41   /// ray tracing (rt) processing elements
42   /// rt_x0_y0_z0
43   sycl_pe_queue_0.submit([&](sycl::handler &h) {
44                    h.set_args(...);
45                    h.single_task(sycl_pe_kernel_0);
46                  });
47   /// rt_x0_y0_z1
48   sycl_pe_queue_1.submit([&](sycl::handler &h) {
49                    h.set_args(...);
50                    h.single_task(sycl_pe_kernel_1);
51                  });
52   ...
53   /// rt_x1_y1_z1
54   sycl_pe_queue_7.submit([&](sycl::handler &h) {
55                    h.set_args(...);
56                    h.single_task(sycl_pe_kernel_7);
57                  });
58
59   // wait for all kernes finished
60   sycl_pe_queue_0.wait();
61   sycl_pe_queue_1.wait();
62   ...
63   sycl_pe_queue_7.wait();
64   sycl_mesh_rw_queue.wait();
65
66   // memcpy (FPGA -> CPU)
67   sycl_mesh_rw_queue.submit([&](sycl::handler &h) {
68                     h.memcpy(h_rmesh, d_rmesh, sizeof(struct radiation_mesh_fpga)*MESH_SIZE);
69                    });
70   sycl_mesh_rw_queue.wait();
71  }
```

Figure 12: DPC++ code used to call the OpenCL implementation of the ART method from the host (CPU).

Fig. 12 also shows the DPC++ code snippet used to make this OpenCL implementation of the ART method available to oneAPI. The DPC++ is based on the C++ standard and the SYCL specifications developed by the Khronos Group, and we include <CL/sycl.hpp> and <CL/opencl.h> headers in lines 1 and 2 to allow OpenCL objects to be handled as DPC++ objects. The operations for this, that is, the conversion of OpenCL objects (`cl_context`, `cl_kernel`, and `cl_queue`) into DPC++ objects, correspond to lines 12–27. To send the mesh data (already copied from the GPU to the CPU) as the initial data to the FPGA from the host CPU, the device memory area of the FPGA is allocated in line 30, and the memory copy from the CPU to the FPGA is executed in lines 31–33. Subsequently, the FPGA-based ART method is executed by launching all OpenCL kernels (the external memory handler and eight PEs) with the operations in lines 35–57. When the ART method finishes, the operation in lines 67–69 writes the execution result from the external memory of the FPGA back to the CPU memory (these data are later copied from the CPU to the GPU).

3.5 Compilation flow

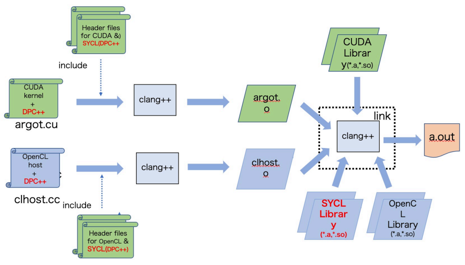


Figure 13: Compilation flow for GPU–FPGA-accelerated ARGOT code with oneAPI.

Fig. 13 shows the compilation flow for GPU–FPGA-accelerated ARGOT code with oneAPI. The compilation procedure itself is the same as in Fig. 7, with a separate compilation and linking of the ARGOT method source code and the ART method source code. The clang compiler used in this compiler flow is an open-source version of the oneAPI Data Parallel C++ compiler, which can input not only DPC++ and OpenCL source codes, but also CUDA code (CUDA Toolkit is used in the backend to generate NVPTX).

In this study, the source codes of CUDA and OpenCL kernels are used as they are, and the DPC++ code is added to the part to invoke the kernels (the part marked in red). These are compiled according to the compile flow shown in the figure, and finally a host binary that can control both the GPU and FPGA is generated. Note that the aocx for loading into the FPGA is generated by the offline compilation using the Intel FPGA SDK for the OpenCL tool, similar to Fig. 7.

4 EVALUATION

4.1 Experimental Settings

Table 2 shows our experimental machine configuration. This is a heterogeneous platform composed of three types of devices: two Intel® Xeon® Gold 6242 CPUs, a single NVIDIA V100 GPU for PCIe-based servers (Gen3 x16), and a single Intel® FPGA PAC D5005 board connected to the CPU through a PCIe Gen3 x16 interface. In this evaluation, we used a single CPU, GPU, and FPGA as located on the same CPU socket to avoid the performance degradation caused by a PCIe access over the Intel UPI (Ultra Path Interconnect).

CentOS 7.3 was used as the operating system of our experimental machine, and the ARGOT code with the oneAPI was compiled using the oneAPI Data Parallel C++ compiler (clang), as shown in the

Table 2: Our experiment environment.

Hardware specifications	
CPU	Intel® Xeon® Gold 6242 × 2
GPU	NVIDIA Tesla V100 (PCIe Gen3 x16 card version)
FPGA	Intel FPGA PAC D5005 (Intel Stratix 10 SX)
Software specifications	
Host OS	CentOS 7.3
Linux Kernel Version	3.10.0-1160.15.2.el7.x86_64
Compiler	oneAPI data parallel C++ compiler [7] (commitID: 6e9ddb6)
MPI	Open MPI 4.0.3
Accelerator Platforms	CUDA 10.2.89 Intel FPGA SDK for OpenCL 2021.2.0 Build 268.1 Pro edition

Table 3: Resource usage and clock frequency of ART applied to FPGA implementation.

ALMs	Registers	M20Ks	DSPs	fmax [MHz]
500,885 (54%)	1,054,972 (28%)	5,070 (43%)	1,916 (33%)	260

previous section. CUDA version 10.2.89 was used as the backend. The ART algorithm working on the FPGA was implemented in OpenCL kernel code and was compiled using the offline compiler provided by the Intel FPGA SDK for OpenCL, version 2021.2.0 Build 268.1 Pro edition. Because the ARGOT code was developed under the assumption that it would run on multiple nodes using MPI, we used OpenMPI 4.0.3, and made the ARGOT code run with a single process during this evaluation.

The problem size used for the evaluation was only 32^3 because the current FPGA implementation of the ART method [11] is not equipped with a feature [15] for dividing a large problem stored in DDR memory into small blocks that can be stored in BRAM, and owing to time constraints, executes the ART calculation through time division multiplexing in each block using the FPGA. The ART applied in the FPGA implementation uses a design with eight PEs (2^3), with each PE having BRAMs for 16^3 meshes. Therefore, 32^3 meshes are stored in an FPGA. Nside, which is a parameter used to determine the resolution in HEALpix, is set to 8, which generates 768 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 was measured and included the cost of launching and synchronizing the device, for both FPGA and GPU implementations. The time for the data transfer between the host and devices is also included.

4.2 Resource Consumption

Table 3 shows the FPGA resource utilization. The ALM (Adaptive Logic Module) is a term used by Intel and is a logic component that includes a logically partitionable LUT (Lookup Table) 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, which is called a block RAM, and internal buffers such as FIFOs are implemented using memory blocks. The DSP (Digital Signal Processor) is a built-in hardware component that is faster and offers more compact implementations of integer multiplications and floating-point operations than programmable logic components. Here, "fmax" means the maximum operating frequency in the clock domain for OpenCL kernels.

If we optimize the OpenCL code to decrease the resource usage in the design, implementing more PEs into the FPGA is possible, which directly leads to a reduction in the execution time. As listed in the table, the ALMs are the greatest resource users in this design, with more than half of the total ALMs used, which becomes a bottleneck when attempting to increase the performance. From the perspective of resource utilization, using all DSPs is the goal for optimization because floating additions and multiplications are implemented on them. However, we cannot double the number of PEs to increase the DPS utilization owing to an ALM overutilization. We consider the next possible number of 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 at a power of 2. In general, as the resource usage per PE is reduced, the operating frequency increases because place-and-routing is easy to apply. This also improves the performance of ART.

4.3 Performance Evaluation of the ARGOT code

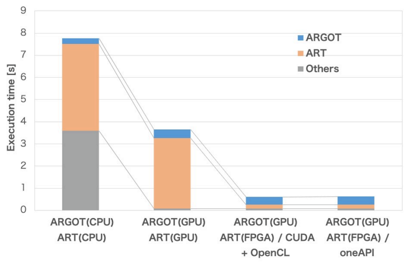


Figure 14: Performance comparison between FPGA, CPU, and GPU implementations. The problem size was 32^3.

Fig. 14 shows the performance comparison between the CPU, GPU, and FPGA implementations, applying a problem size of 32^3. These results indicate the execution time per simulation step. "ARGOT (CPU) / ART (CPU)" indicates that both algorithms are implemented in the CPU, 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 used a single Xeon CPU, and the CPU implementation was applied with 16 OpenMP cores (threads). The GPU implementation is based on the CPU implementation but written in CUDA.

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 the chemical reactions and radiative heating/cooling of each mesh, based on the execution results of the ARGOT and ART algorithms. Fortunately, solving the chemical reactions and radiative heating/cooling of each mesh is an independent task, which 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. 14, the execution of the ART algorithm is not accelerated. 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, their problem sizes are too small to sufficiently exploit the 5,120 CUDA cores of the GPU because the requisite parallelism is not achievable, and GPU kernel activation and CPU-GPU communication may also reduce the performance.

As reported in [15], on the GPU, the performance of the ART algorithm improves significantly when solving larger sized problems. 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 2.2, 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.

On the other hand, the FPGA-based ART implementation is better than the GPU-based implementation even when solving smaller sized problems. As reported in [12], this high performance comes from the pipelined ART algorithm implemented in the FPGA, and compared with the ART method applied on a GPU, we achieved a performance improvement of 18.7 and reduced the execution time of the entire ARGOT code to 5.42 times.

We then compared the CUDA+OpenCL implementation with the oneAPI implementation and found that there was almost no difference in performance between the two. In our implementation of the GPU–FPGA-accelerated ARGOT code using oneAPI, the ARGOT and ART methods used in the CUDA+OpenCL implementation are directly applied (in fact, the oneAPI implementation applied in this evaluation still uses the CUDA API instead of the DPC++ API for a small portion of the ARGOT method, such as the GPU device memory allocation, but it does not affect the performance evaluation). In other words, this result shows that the existing developed application codes can be used in oneAPI with almost zero performance loss. As previously described in Section 3.2, by making the GPU and FPGA work collaboratively under oneAPI, it is possible to use a unified API (i.e., `queue`) to launch the kernel of each device and to synchronize and mediate between the two devices, thus maintaining the high maintainability and availability of the application code.

When the ARGOT method is executed on the GPU, the execution time increases slightly compared to the CPU because the ARGOT algorithm includes code to allocate pinned memory for each iteration. This is because the ARGOT algorithm includes a code that allocates the pinned memory for each iteration. Although this reduces the computation time, the overhead of allocating the pinned memory hinders such a reduction. Therefore, in the future, we will modify the code to allocate the pinned memory outside of the iteration to reduce the overhead as much as possible. As reported in [15], the ARGOT and ART methods are inherently independent and can be concurrently performed on each device. As a result, achieving a higher simulation speed can be expected, which will be an area of future study.

5 RELATED WORK

For our investigation, research on cooperative computations between CPUs, GPUs, and FPGAs conducted for more than a 10-year period was considered. KH Tsoi et al. [19] proposed a heterogeneous computer cluster called Axel that contains 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 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; therefore, the authors of [19] developed their own framework for the Axel cluster based on MapReduce. This means that in order to run an already developed application on the Axel cluster, it is mandatory to re-implement the application to fit the MapReduce framework. To make matters worse, the FPGA implementation at the time had to be applied using a hardware description language owing to a lack of high-level synthesis tools. In addition, despite the fact that a framework was developed that could handle the CPU, GPU, and FPGA comprehensively, it is likely that there were few users who could actually use it to the fullest.

The last decade has seen dramatic advances in high-level synthesis tools, and many studies have proposed frameworks that make use of these tools. For example, similar to our research, María Angélica Dávila Guzmán et al. [10] proposed a framework to handle the CPU, GPU, and FPGA in a single programming language. They reported that they extended EngineCL, a runtime that acts as a wrapper for OpenCL C++, to support FPGAs with Intel FPGA SDK for OpenCL (an HLS tool offered by Intel), and that CPUs, GPUs, and FPGAs have achieved cooperative computations in such a framework. They also confirmed that the inter-device load-balancing algorithm provided by EngineCL improves the performance of benchmarks run on compute nodes equipped with CPUs, GPUs, and FPGAs. However, as previously described, most of the existing HPC applications are CUDA-based implementations, and it would be extremely burdensome for programmers to rewrite all of the code into OpenCL for compatibility with EngineCL.

Although the Intel oneAPI was recently released for users in 2020, results on the use of this toolkit have already been reported [16]. The authors of this study implemented the algorithm for higher-order (fourth-order) epistasis detection in DPC++ (SYCL) and evaluated its performance on an Intel Xeon Gold 6128 CPU (x2), Intel UHD P630, Iris Xe MAX, and NVIDIA Titan RTX (Turing). In the future, they plan to implement a heterogeneous cluster-based acceleration with custom hardware such as FPGAs in addition to CPUs and GPUs. Their implementation policy for multi-hybrid acceleration can be categorized using Fig. 9 and falls under approach (a). It is clear that if the development of a new code is required, it makes sense to adopt approach (a) from the viewpoint of maintainability and availability of the code, which we also plan to do. However, as we pointed out in [10], for users who have already developed code (most of which is implemented in CUDA) and are required to run it with Intel oneAPI, this is a huge burden because it is the same as being forced to re-implement their applications in DPC++.

Therefore, in this study, we focus on approach (b), which reduces the programming effort as much as possible for application users, by exploiting the existing CUDA and OpenCL code without modification, and by using DPC++ for their cooperative operation. We also show a GPU–FPGA-accelerated ARGOT code with Intel oneAPI as a proof of concept. We believe that this study will enlighten application users who are required to accelerate their already implemented HPC applications with oneAPI.

6 CONCLUSION

In this paper, we presented the implementation and performance evaluation of our astrophysics code ARGOT to apply a oneAPI solution using a GPU and an FPGA. To realize our CHARM concept for complex multi-physical simulations as a next generation of accelerated supercomputing, this study was conducted on our multi-heterogeneous accelerated cluster machine running at the University of Tsukuba.

To reduce the programming effort as much as possible for application users, we focus on an approach to exploit the existing CUDA and OpenCL codes without modification, and by using DPC++ for their cooperative operation. We evaluated the performance of the GPU-FPGA-accelerated ARGOT code implemented with oneAPI and found that it is comparable to the performance of the ARGOT code implemented with CUDA+OpenCL. This means that existing application codes can be used through oneAPI with almost zero performance loss. Furthermore, by making the GPU and FPGA work collaboratively under oneAPI, it is possible to use a unified API (i.e., `queue`) to launch the kernel of each device and to synchronize and mediate between the two devices, thus maintaining the high maintainability and availability of the application code.

We showed that the GPU and FPGA can be controlled under the oneAPI, and that a small part of the code (the part that allocates GPU device memory) uses the CUDA API. Therefore, in parallel with replacing this part with the DPC++ API, in future studies we will work on further optimization of the GPU–FPGA-accelerated ARGOT code, specifically, the concurrent execution of the ARGOT and ART methods, and optimization of the pinned memory allocation method used in the ARGOT approach.

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 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] 2020. Cygnus Supercomputer. https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#Cygnus.

[2] 2021. Build DPC++ toolchain with support for NVIDIA CUDA. https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-support-for-nvidia-cuda.

[3] 2021. DPC++ OSS Compiler Project. https://www.codeplay.com/solutions/oneapi/for-cuda/.

[4] 2021. Intel oneAPI Base Toolkit System Requirements. https://software.intel.com/content/www/us/en/develop/articles/intel-oneapi-base-toolkit-system-requirements.html.

[5] 2021. NVIDIA A100 GPU. https://www.nvidia.com/en-us/data-center/a100/.

[6] 2021. oneAPI. https://www.oneapi.com/.

[7] 2021. oneAPI Data Parallel C++ compiler (Open source version). https://github.com/intel/llvm.

[8] 2021. OpenACC. https://www.openacc.org/.

[9] 2021. TOP500 Supercomputer Sites. https://www.top500.org/lists/top500/2021/06/.

[10] 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. *J. Supercomput.* 75, 3 (March 2019), 1732–1746. https://doi.org/10.1007/s11227-019-02768-y

[11] 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)*. 48–55. https://doi.org/10.1109/H2RC51942.2020.00011

[12] 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

[13] 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

[14] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, and Masayuki Umemura. 2020. Accelerating Radiative Transfer Simulation with GPU-FPGA Cooperative Computation. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 9–16. https://doi.org/10.1109/ASAP49362.2020.00011

[15] 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

[16] Ricardo Nobre, Aleksandar Ilic, Sergio Santander-Jiménez, and Leonel Sousa. 2021. Fourth-Order Exhaustive Epistasis Detection for the XPU Era. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3472456.3472509

[17] 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 arXiv:http://oup.prod.sis.lan/mnras/article-pdf/419/4/2855/9505080/mnras0419-2855.pdf

[18] 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). https://doi.org/10.1093/pasj/psv027 arXiv:http://oup.prod.sis.lan/pasj/article-pdf/67/4/62/5165633/psv027.pdf 62.

[19] 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

[20] Ryuta Tsunashima, Ryohei Kobayashi, Norihisa Fujita, Taisuke Boku, Seyong Lee, Jeffrey Vetter, Hitoshi Murai, Masahiro Nakao, and Mitsuhisa Sato. 2020. OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration. In *13th International Symposium on High-level Parallel Programming and Applications* (Porto, Portugal) *(HLPP 2020)*.