OpenACC Unified Programming Environment for Multi-hybrid Acceleration with GPU and FPGA

Taisuke Boku[1,2(✉)], Ryuta Tsunashima[2], Ryohei Kobayashi[1,2], Norihisa Fujita[1,2], Seyong Lee[3], Jeffrey S. Vetter[3], Hitoshi Murai[4], Masahiro Nakao[4], Miwako Tsuji[4], and Mitsuhisa Sato[4]

[1] Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan
{taisuke,rkobayashi,fujita}@ccs.tsukuba.ac.jp
[2] Degree Programs in Systems and Information Engineering, University of Tsukuba, Tsukuba, Japan
tsunashima@hpcs.cs.tsukuba.ac.jp
[3] Oak Ridge National Laboratory, Oak Ridge, USA
{lees2,vetter}@ornl.gov
[4] RIKEN Center for Computational Science, Kobe, Japan
{h-murai,masahiro.nakao,miwako.tsuji,msato}@riken.jp
https://www.ccs.tsukuba.ac.jp/

Abstract. Accelerated computing in HPC such as with GPU, plays a central role in HPC nowadays. However, in some complicated applications with partially different performance behavior is hard to solve with a single type of accelerator where GPU is not the perfect solution in these cases. We are developing a framework and transpiler allowing the users to program the codes with a single notation of OpenACC to be compiled for multi-hybrid accelerators, named MHOAT (Multi-Hybrid OpenACC Translator) for HPC applications. MHOAT parses the original code with directives to identify the target accelerating devices, currently supporting NVIDIA GPU and Intel FPGA, dispatching these specific partial codes to background compilers such as NVIDIA HPC SDK for GPU and OpenARC research compiler for FPGA, then assembles binaries for the final object with FPGA bitstream file. In this paper, we present the concept, design, implementation, and performance evaluation of a practical astrophysics simulation code where we successfully enhanced the performance up to 10 times faster than the GPU-only solution.

Keywords: GPU · FPGA · Programming framework · OpenACC · MHOAT

1 Introduction

GPU is the main player as a powerful accelerator on supercomputers, especially for ultra-large scale systems to achieve a high performance/power ratio. Over half of the world's top-10 machines in TOP500 List are equipped with GPUs. However, GPU is not a perfect accelerating device in such cases with:

A. Bienz et al. (Eds.): ISC High Performance 2023 Workshops, LNCS 13999, pp. 1–13, 2023.
https://doi.org/10.1007/978-3-031-40843-4_49

– poor degree of uniform parallelization lower than core count
– frequent conditional branches
– frequent internode communication, etc.

On the other hand, FPGA (Field Programmable Gate Array) becomes attractive as another candidate for accelerator [6,14,15]. The advantages of introducing FPGAs in HPC applications are:

– true codesigning system to be specialized for the target application
– pipelined parallel execution not in SIMD-manner
– high-end FPGAs are equipped with their own high-speed optical links for parallel FPGA environment
– relatively low power consumption compared with GPU

However, in most traditional research to employ FPGA in HPC applications, it is shown that the absolute performance of FPGA implementation is lower than GPU. Therefore, we should employ FPGA in problems where some parts of the entire computation are unsuitable for GPU. Even if a small fraction of the application cannot be improved, it makes the limit of performance even with a high performance of GPU according to the Amdahl's Law.

We have been researching the coupling of GPU and FPGA together toward highly efficient accelerated computing under the concept of CHARM (Cooperative Heterogeneous Acceleration with Reconfigurable Multidevices) [4,8] where both devices compensate with each other by different performance characteristics (Fig. 1). In this concept, we name such a computing framework with multiple types of accelerators as *multi-hybrid* computing.

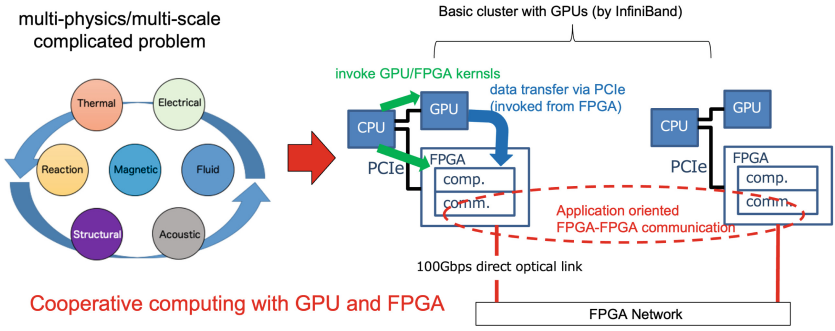

Fig. 1. Concept of CHARM

In previous works, we have researched the programming method and framework to apply the CHARM concept and developed the first practical application code. In [5], we presented an astrophysical application that implies heavy computation part where GPU acceleration does not work well, and we ported that part to Intel Arria10 FPGA to achieve up to 10 times faster performance with the

programming of OpenCL High Level Synthesis (HLS). Then in [8] we successfully combined that OpenCL code for FPGA and CUDA code for GPU to assign two devices appropriately where each one achieves higher performance than the other. In the latter work, we found that it is possible to combine partial binary modules compiled by CUDA compiler for NVIDIA GPU and OpenCL compiler for Intel FPGA (Intel FPGA SDK for OpenCL [1]) without any conflict on symbols and modules. The overall performance of the entire code improves up to 17 times faster than GPU-only code. This multi-hybrid code was also ported to Intel oneAPI [3] environment for more sophisticated device controlling and data management without any performance loss, as reported in [7].

However, these works are complicated for general HPC users, requiring a mixture of CUDA and OpenCL programming. To solve this problem, we have been developing a comprehensive framework for a single language solution to offload appropriate code parts to multi-hybrid devices based on OpenACC. We initially considered using OpenCL both for GPU and FPGA as the basic notation of accelerator offloading since both devices support this language framework. However, OpenCL is still relatively low for application users and too complicated to mix in multi-hybrid accelerator programming by them. Therefore, we decided to use OpenACC as the higher level of the basic framework.

2 MHOAT - Single Language Coding for CHARM

The language processing environment we developed is named MHOAT (Multi-Hybrid OpenACC Translator) to hire several backend compilers for GPU and FPGA to allow users to code simply in OpenACC with a small directive expansion. The preliminary work was reported in [13] with the prototype implementation on tiny sample codes. In this paper, we report a practical example of actual code compiled by MHOAT and its functionality enhancement of MHOAT itself. We also apply a programming method to increase spatial parallelism to improve the performance of FPGA.

As described in the previous section, we have confirmed that CHARM programming is possible and practical with GPU and FPGA to apply CUDA and OpenCL for each device, respectively. However, both languages are relatively in low level for general HPC users, although CUDA is popular for NVIDIA GPU, and it is ideal for them to program with higher level and easy-to-understand language such as OpenMP. As a language for accelerators, OpenACC inherits many concepts and ideas from OpenMP, and it is relatively easy to port OpenMP codes to OpenACC toward easy GPU acceleration for general users. Although the language is available only for NVIDIA GPUs, these GPU families dominate the market, and many applications have been developed. If a user can program only with OpenACC both for GPU and FPGA, it is ideal for the CHARM programming framework.

Since creating a single compiler to cover every type of accelerating device is challenging, we need several backend compilers for all supported devices and a top-level language processing system to analyze the program modules assigned

to appropriate devices based on user definition. Therefore, in the basic compilation framework of multi-hybrid OpenACC programming, we need the following process flow as shown in Fig. 2.

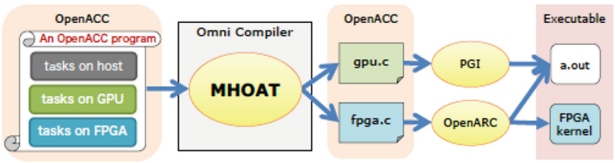

Fig. 2. Basic flow of single OpenACC code to process by backend compilers

The essential job of the MHOAT process is to separate tasks described in OpenACC pragma, such as *kernels* or *loop* to offload the target loop block to the accelerators. Here, we introduce an original pragma extension named **target_dev** to identify which accelerator is the target of that part, as shown in Fig. 3. We use **accomn** pragma instead of **acc** to identify that feature is our original extension (*omn* comes from our project name Omni for our compiler development). Currently, two device families, *GPU* and *FPGA*, can be specified as target devices.

```
#pragma accomn target_dev(GPU)
#pragma acc kernels
for(i=0; i<N; i++)   // this loop is offloaded to GPU
...
#pragma accomn target_dev(FPGA)
#pragma acc kernels
for(i=0; i<M; i++)   // this loop is offloaded to FPGA
...
```

Fig. 3. Extended directive **accomn target_dev** to target the accelerators

Then, MHOAT splits the source code into several files to be processed by the backend compilers, as shown below.

- Reading the program file and parsing the OpenACC directives for offloading to devices, especially with the specification of target accelerators,
- Separating the program fragments dispatched to the target backend compilers according to the target devices, and
- Assembling partially compiled binary objects created by these compilers into a final object file with several supportive run-time routines.

Currently, there is no commercial compiler for OpenACC on FPGA, and we only have HLS compilers by Intel or Xilinx for OpenCL, standard C, or C++. One of the solutions for OpenACC compilation is OpenARC [10] by Oak Ridge National Laboratory. It is a research compiler for multiple target devices, including GPUs and FPGAs. However, the device handling and data management policy is limited, and we like to extend more aggressive features on the system, for example, implying the fast DMA transfer mechanism between GPU and FPGA, which we originally developed. Therefore, we use the function of OpenARC to translate OpenACC to OpenCL only for FPGA. For GPU compilation, we can use the traditional compiler, NVIDIA SDK for HPC by PGI and NVIDIA [2].

Based on this design, we developed a prototype of a meta compiler for multi-hybrid OpenACC compilation, named MHOAT (Multi-Hybrid OpenACC Translator). Figure 4 shows the entire construction of MHOAT. Here, two backend compilers, NVIDIA SDK for HPC for NVIDIA GPU and OpenARC compiler for Intel FPGA, are invoked in the process flow. We have confirmed that the symbols and objects do not conflict between these compilers, so the binary of x86 host CPU can be easily assembled. For the partial compilation of OpenACC kernels dispatching to FPGA, the OpenARC compiler generates an OpenCL source code as the target file. We compile it by Intel FPGA SDK for OpenCL to create the target bitstream file (aocx file) to download to FPGA.

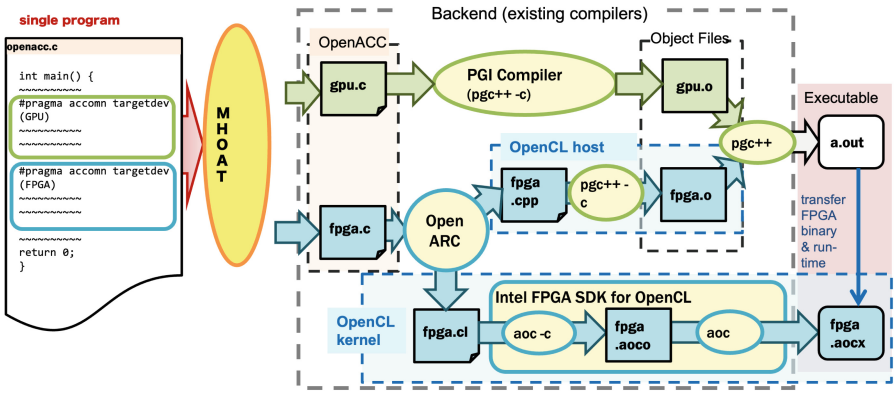

Fig. 4. Compilation flow of MHOAT with NVIDIA and OpenARC backend compilers

The current version of MHOAT does not support device-to-device direct data movement. Data handling should be treated as ordinary data directive in OpenACC as the relationship between the host CPU and the accelerating device. Therefore, when the data created by GPU is referred to on FPGA, it should first be synchronized with CPU memory, then synchronized with CPU and FPGA device memory. Introducing a feature to handle device-to-device data transfer is our future work.

3 OpenACC Coding for High Performance on FPGA

While pipeline parallelism is the base of performance gain on FPGA, there is a limit to the number of operations in the pipeline according to the target application. It is required to enlarge the number of operations per clock by exploiting spatial parallelism. A primary and efficient method is loop unrolling. However, the applicability is limited, especially due to the memory bandwidth bottleneck if high Byte/FLOP is required. To achieve a higher level of parallelization in coarse grain, we introduce multiple kernels to apply domain decomposition, like MPI programming on distributed memory architecture. That is a simple solution to enhance the performance to exploit a large amount of computing elements in the same manner as traditional HPC programs. The difference is how to connect multiple computation kernels within an FPGA. Intel FPGA SDK for OpenCL provides a feature named Channel to create a communication pipeline buffer for any pair of contact points of two kernels. The user can define an arbitrary number of Channels.

OpenARC provides a feature to program Intel Channel, which can be directly transformed into OpenCL code with Channel function. However, the current version of OpenARC supports only the default parameters on Channel attributes, and we need more flexibility for performance tuning (described later).

In Intel FPGA SDK for OpenCL, it is recommended to use Single Work-Item kernel to exploit pipeline parallelism rather than NDRange for spatial parallelism used for GPU usually. To exploit high performance with pipelining, we implement multiple kernels (in this study, eight identical kernels), and connect them by Channels. To program it, the code should be written as shown in Fig. 5.

```
    void fpga(...) {
#pragma accomn target_dev(FPGA)
    int channel1[1];
    int channel2[1];
    ...
#pragma acc data copy(a[:N], b[:N]) pipe(channel1[:1], channel2[:1])
{
#pragma acc serial pipein(channel1) pipeout(channel2) async(0)
    for(i=0; i < N/2; i++)
    ...
#pragma acc serial pipein(channel2) pipeout(channel1) async(1)
    for(i=N/2; i < N; i++)
    ...
} // acc data end
#pragma acc wait
...
}
```

Fig. 5. Multiple kernels by Single Work-Item, connected by Channel

Here, two kernels taking the first and latter half of data run asynchronously, and data are copied through two Channels, *channel1* and *channel2*. **serial** pragma is specified, but actually, these loops are pipelined by the FPGA compiler.

As described before, the overall performance of an FPGA is limited by the number of actual operations per clock and the frequency. In several cases, the compiler cannot estimate the behavior of the loop, especially on the dependency between operations, and it makes very conservative solutions where setting the pipeline pitch (single-stage latency) is too long. It is reflected in low clock frequency to limit the performance. In HPC applications, a typical case is a sort of reduction operation where a number of data are finally accumulated to finalize a scalar number or vector. If the calculation is complicated with several elements, it may happen more than expected.

Currently, we could not find a concrete solution to tell a reasonable estimation of pipeline latency to the compiler, so the final solution is to go back to the old-fashioned low-level description - Verilog HDL. It is similar to using *asm* construct in C language to escape to assembler coding to solve the problem partially. In our case of the target practical application shown in the next section, we could not solve this problem and introduce a function written in Verilog HDL to avoid too much long latency. However, we also did it in the previous works with CUDA and OpenCL descriptions, so this is not a fundamental problem of OpenACC coding. In our MHOAT solution, invoking functions written in Verilog HDL is possible.

4 Practical Application Example - ARGOT

As the first target application, we implement ARGOT (Accelerated Radiative transfer on Grids using Oct-Tree) [11] code developed in the Center for Computational Sciences, University of Tsukuba. It is a fundamental astrophysics simulation code to analyze how the first objects such as stars and galaxies were created in the universe about 500,000 years after the Big Bang. That is quite a short term in more than 13,000,000,000 years of its history. The most important source of these objects' creation is the radiation and the collection of tiny dust clouds. Radiation spread from a baby star or spatially spread in the universe affects other small objects to make their growth. There are two kinds of radiation calculated in the ARGOT code.

- Radiation spread from each Point Source (object) as shown in the left hand of Fig. 6. It can be calculated similarly to the gravity calculation where tree-code is applied like ordinary computation in astrophysics. This computing is named the ARGOT method.
- Radiation spread from long distance objects to be treated as potentially existing ones to go across the space field as shown in the right hand of Fig. 6. The calculation is similar to Ray-Tracing, where many light arrows cross the field and hit objects. This computing is named the ART (Authentic Radiation Transfer) method [12].

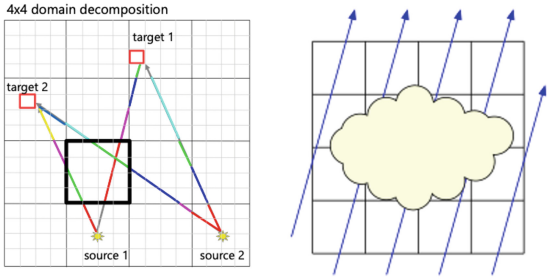

Fig. 6. Two methods of radiation transfer: ARGOT (left) and ART (right) methods

Remind that ARGOT *code* is for the entire application program while ARGOT *method* is a part of the computation in the code. So that ARGOT *code* mainly consists of ARGOT *method* and ART *method* computations as well as other additional physical phenomena.

Through the past work [5], we found the GPU acceleration does not work effectively on ART method calculation while it is suitable for ARGOT method like tree-code on GPUs. There are two main reasons. 1) Memory access pattern in ART method is almost random where the HBM memory of GPU does not work effectively, and 2) each radiation arrow to pass the target space is too short for the SIMD operation of the GPU core. Both features are critical to GPU performance, so GPU cannot improve ART method calculation. The serious problem is that approximately 90% of the computation cost on the GPU-only version of ARGOT code is consumed for ART method. That means we cannot accelerate the entire ARGOT code with GPUs only. We implemented the ART method part to Intel Aria10 FPGA, and achieved about ten times faster performance than GPU (NVIDIA P100). That is an excellent achievement of FPGA applied to HPC applications. Since GPUs are still applicable for ARGOT method much better than FPGA, we decided to apply the CHARM concept to the ARGOT code, as shown in Fig. 7.

5 Performance Evaluation

5.1 ARGOT Code Implementation for MHOAT

There are several versions of ARGOT code for CPU (with OpenMP) and GPU (with CUDA and OpenACC). All of them are parallelized by MPI for large scale computing in domain decomposition manner. However, this paper evaluates only a single node without MPI.

As described in the previous section, we applied a multi-kernel solution for domain decomposition within an FPGA on ART method calculation to exploit the coarse grain parallelism. Eight kernels are running in parallel and connected by Intel Channel with each other. We define a long macro description to implement a kernel and use it eight times to duplicate them so that the effort for the

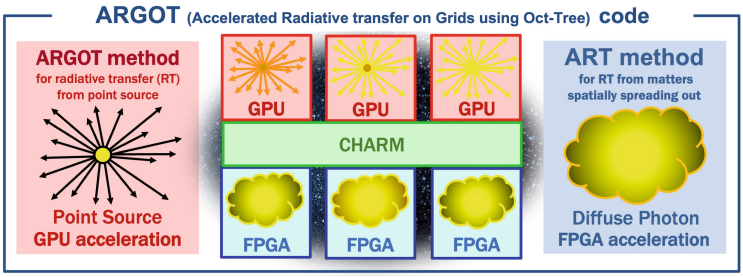

Fig. 7. ARGOT method and ART method mapping to GPU and FPGA in CHARM concept for ARGOT code

multi-kernel solution is easy. For inter-kernel communication, we need to apply a much deeper pipeline buffer on Channel communication than the default parameter set. However current version of the OpenARC compiler has no feature for optional Channel parameter settings. Therefore, we modify this part of OpenCL code generated from OpenACC by OpenARC compiler with a sort of 'patch script' after compilation.

Since the memory access pattern of ART method is almost random, we allocate all temporal data for target space in BRAM (Block RAM), which is a kind of SRAM implemented with the same calculation logic circuit. However, since the capacity of BRAM is limited to just 28 MByte, we keep all the data in DDR DRAM outside of FPGA, and transfer them according to the calculation progress. It means that we use BRAM as an addressable cache. That is another coding technique to enhance the FPGA performance.

As described before, the current MHOAT does not support the device-to-device data synchronization feature. However, in the ARGOT code, the data amount transferred between GPU and FPGA is negligible, with less than 1% of execution time, and it does not impact the performance.

5.2 Performance Evaluation

The target platform consists of two sockets of Intel Xeon E5-2690 v4 as host CPUs, NVIDIA Tesla V100 (32 GiB HBM2, PCIe Gen3×16), and Intel Stratix10 GX2800 FPGA (BittWare 520N card, PCIe Gen3×16).

Figure 8 shows the overall performance comparison of the MHOAT solution on ARGOT code compared with GPU-only code in OpenACC, GPU-only code in CUDA, and GPU + FPGA code with CUDA + OpenCL. That is the case with $32 \times 32 \times 32$ grid, a relatively small size problem of ARGOT code where the GPU performance bottlenecks by ART method calculation. The floating point operation is in single precision. Each bar implies the computation time for ARGOT method (GPU), ART method (GPU or FPGA), and Others (miscellaneous tasks mainly working on CPU with OpenMP).

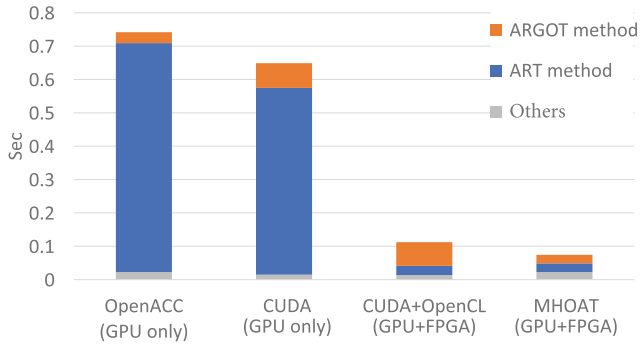

Fig. 8. ARGOT code execution time (1 step) with four styles of programming

At first, we confirmed that the performance gain by CHARM programming is significantly higher than GPU-only cases, and MHOAT achieves the best performance. The ART method calculation performance on GPU is poor because the excellence of GPU architecture does not fit, so the ART method part dominates the computation time. For the GPU-only cases on ART method, the efficiency of OpenACC is slightly higher than CUDA, and it still shows the CUDA coding with detailed tuning is better than OpenACC. On the other hand, the ARGOT method calculation on GPU is improved in OpenACC. We have yet to analyze the reason, but it has a good effect on the MHOAT solution.

Table 1. Breakdown of execution time (sec./step)

	OpenACC (GPU-only)	CUDA (GPU-only)	CUDA+OpenCL (CHARM)	MHOAT (CHARM)
ARGOT method	0.033	0.074	0.071	0.026
ART method	0.686	0.559	0.028	0.026
Others	0.023	0.015	0.014	0.023
Total	0.742	0.649	0.113	0.075

Table 1 shows the detailed execution time for each computation part in all cases. In the main computation for ARGOT method and ART method, the MHOAT solution is the best. However, the miscellaneous tasks take a long time on MHOAT. In total, MHOAT still keeps the advantage over the CUDA+OpenCL coding. When we see the two bars in the right hand, it is clearly saying that the CHARM solution dramatically improves the performance of this application. Especially by MHOAT, 1) ART method on FPGA has no performance difference between OpenCL and OpenACC thanks to an efficient compilation on OpenARC, and 2) ARGOT method calculated on GPU is better in MHOAT because that performance is better in OpenACC than CUDA as

shown in two bars in the left hand. In conclusion, these results show the excellence of the MHOAT solution where the performance of ARGOT code execution is 9.9 times faster than OpenACC with GPU-only.

Table 2. Source Line Of Count (SLOC) for ART method on OpenCL and OpenACC

	OpenCL host	OpenCL kernel	OpenACC
SLOC	263	945	900

Finally, we examined the source code line counts. The difference between the naive CUDA+OpenCL coding and MHOAT OpenACC coding for multi-hybrid computing is shown in Table 2. Here, SLOC (Source Line Of Code) for OpenCL is 1208 lines, with 263 for the host (CPU) and 945 for the kernel (FPGA). On the other hand, OpenACC in MHOAT counts just 900 lines. Simply the source line count is reduced to 75%. Moreover, the user has to understand OpenACC notation only for easy programming, even for CHARM.

6 Discussion

The concept of CHARM is to provide multi-hetero accelerating environment on supercomputers where a single kind of accelerator, such as GPU, does not effectively work partially in the code, while other parts have no performance problem. That problematic parts bottleneck the total execution performance according to the Amdahl's Law. On multi-physics applications especially, that is a practical problem. Thus, a different kind of accelerator, such as FPGA, could support and compensate these parts. If the entire code can be accelerated only by GPU, there is no problem, and GPUs have been used for such cases so far.

MHOAT requires users to specify the computation parts to offload to GPU or FPGA. On the other hand, Intel oneAPI allows any computation part to offload to GPU and FPGA. This is because oneAPI assumes offloading any kernel dynamically for load balancing over multiple devices. However, running the same kernel code on both is impractical because the performance characteristics are wholly different depending on the devices. In our concept, a user must choose the devices how to run each kernel according to its computation behavior. The performance result of ARGOT code clearly proves it. Therefore, it is unnecessary to compile all loops (kernels) for FPGA, and perform it only for them to run on that device.

Another issue is the footprint of the logic circuit of an FPGA device. Since it is critically limited, there is only room to compile some of the loops for FPGA in a complicated code. In this viewpoint, we should limit the kernels on FPGA to only the necessary ones. In conclusion, explicitly limiting the target kernels for FPGA is a necessary and sufficient condition in the CHARM concept.

7 Conclusions

In this paper, we described our concept of CHARM, where multi-hybrid heterogeneous computing is a powerful solution for multi-physics simulations such as complicated astrophysics ones. While GPU is the most powerful and efficient solution as a representative accelerating computing device, it is imperfect. On the other hand, FPGA is relatively weak on HPC applications compared with GPU, but the performance characteristics and architecture behavior are almost opposite from GPUs. Thus, we combine both devices to compensate for each other in performance. We developed MHOAT for a single notation programming framework with OpenACC to support easy programming. In the first practical application of astrophysics, we achieved approximately ten times higher performance on MHOAT-coded astrophysical simulation driven by GPU+FPGA than traditional GPU-only solution with OpenACC.

Our future works imply the node-parallel extension of ARGOT code with MPI. It is easy to work because the original OpenACC version of ARGOT code is already written for MPI, and nothing by MHOAT inhibits it. We have already done with parallelized CHARM code with CUDA+OpenCL [9] where we showed a good performance improvement with a parallel environment with GPU+FPGA. The current version of OpenARC used in MHOAT needs several enhancements, such as more flexible Channel usage, automatic spatial parallelism improvement such as loop unrolling, etc. Our final goal for ARGOT on CHARM is to run the program on the full scale of our CHARM-base cluster Cygnus [4].

Acknowledgements. This work is supported by JSPS KAKENHI (Grant Number 21H04869). The Cygnus utilization is supported by the MCRP 2022 Program by the Center for Computational Sciences, University of Tsukuba.

References

1. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
2. Nvidia HPC SDK: A comprehensive suite of compilers, libraries and tools for HPC. https://developer.nvidia.com/hpc-sdk
3. oneAPI: A new era of accelerated computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.smg356
4. Boku, T., Fujita, N., Kobayashi, R., Tatebe, O.: Cygnus - world first multi-hybrid accelerated cluster with GPU and FPGA coupling. In: 2nd International Workshop on Deployment and Use of Accelerators (DUAC2022) (2022)
5. Fujita, N., et al.: Accelerating space radiative transfer on FPGA using OpenCL. In: 2018 International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018) (2018). https://doi.org/10.1145/3241793.3241799
6. Hill, K., Craciun, S., George, A., Lam, H.: 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 (ASAP2015), pp. 189–193 (2015)

7. Kashino, R., Kobayashi, R., Fujita, N., Boku, T.: Multi-hetero acceleration by GPU and FPGA for astrophysics simulation on intel oneAPI environment. In: Proceedings of International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2022) (2022)
8. Kobayashi, R., et al.: Multi-hybrid accelerated simulation by GPU and FPGA on radiative transfer simulation in astrophysics. J. Inf. Process. **28**, 1073–1089 (2020). https://doi.org/10.2197/ipsjjip.28.1073
9. Kobayashi, R., et al.: GPU-FPGA-accelerated radiative transfer simulation with inter-FPGA communication. In: 2023 International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2023) (2023)
10. Lee, S., Kim, J., Vetter, J.S.: OpenACC to FPGA: a framework for directive-based high-performance reconfigurable computing. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS2016), pp. 544–554 (2016)
11. Okamoto, T., Yoshikawa, K., Umemura, M.: ARGOT: accelerated radiative transfer on grids using oct-tree. Monthly Not. Roy. Astron. Soc. **419**(4), 2855–2866 (2012)
12. Tanaka, S., Yoshikawa, K., Okamoto, T., Hasegawa, K.: A new ray-tracing scheme for 3D diffuse radiation transfer on highly parallel architectures. Publ. Astron. Soc. Jpn. **67**(4), 1–16 (2015)
13. Tsunashima, R., et al.: OpenACC unified programming environment for GPU and FPGA multi-hybrid acceleration. In: 13th International Symposium on High-level Parallel Programming and Applications (HLPP2020) (2020)
14. Tsuruta, C., Miki, Y., Kuhara, T., Amano, H., Umemura, M.: Off-loading let generation to peach2: a switching hub for high performance GPU clusters. ACM SIGARCH Comput. Archit. News **43**(4), 3–8 (2016)
15. Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M., Matsuoka, S.: 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 2016), pp. 35:1–35:12 (2016)