Automatika
Journal for Control, Measurement, Electronics, Computing and Communications

ISSN: (Print) (Online) Journal homepage: www.tandfonline.com/journals/taut20

Study and evaluation of automatic offloading for function blocks of applications

Yoji Yamato

To cite this article: Yoji Yamato (2024) Study and evaluation of automatic offloading for function blocks of applications, Automatika, 65:1, 387-400, DOI: 10.1080/00051144.2024.2301888

To link to this article: https://doi.org/10.1080/00051144.2024.2301888

© 2024 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.

View supplementary material

Published online: 09 Jan 2024.

Submit your article to this journal

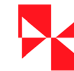
Article views: 275

View related articles

View Crossmark data

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=taut20

AUTOMATIKA
2024, VOL. 65, NO. 1, 387–400
https://doi.org/10.1080/00051144.2024.2301888



Taylor & Francis
Taylor & Francis Group

RESEARCH ARTICLE

OPEN ACCESS Check for updates

Study and evaluation of automatic offloading for function blocks of applications

Yoji Yamato

Network Service Systems Laboratories, NTT Corporation, Tokyo, Japan

ABSTRACT
Systems using graphical processing units (GPUs) and field-programmable gate arrays (FPGAs) have increased due to their advantages over central processing units (CPUs). However, such systems require the understanding of hardware-specific technical specifications such as Hardware Description Language (HDL) and compute unified device architecture (CUDA), which is a high hurdle. Based on this background, we previously proposed environment-adaptive software that enables automatic conversion, configuration and high-performance operation of existing code according to the hardware to be placed. As an element of this concept, we also proposed a method of automatically offloading loop statements of application source code for CPUs to GPUs and FPGAs. In this paper, we propose a method for offloading a function block, which is a larger unit, instead of individual loop statements in an application to achieve higher speed by automatically offloading to GPUs and FPGAs. We implemented the proposed method and evaluated it using current applications offloading to GPUs and FPGAs.

ARTICLE HISTORY
Received 13 August 2021
Accepted 29 December 2023

KEYWORDS
Environment adaptive software; GPGPU; automatic offloading; performance; function block

1. Introduction

It is mentioned in [1] that Moore's law will be slow down and a central processing unit's (CPU's) transistor density cannot be expected to double in 1.5 years. Based on this situation, systems with heterogeneous hardware, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), are increasing. For example, Microsoft's search engine Bing uses FPGAs [2], and Amazon Web Services (AWS) provides GPUs and FPGAs using cloud technologies [3–6].

However, to achieve high application performance using heterogeneous hardware, developers need to appropriately program and configure such hardware and use high skill technologies such as compute unified device architecture (CUDA) [7] and open computing language (OpenCL) [8]. This is a high barrier to using GPUs or FPGAs for offloading.

Along with the progress in Internet of Things (IoT) technology (e.g. Industrie 4.0 [9] and so on [10–15]), connected IoT devices are increasing rapidly. Gartner, Inc. forecasts the number of IoT devices will reach several tens of billions in 2020 and will be connected to networks. There are various application fields of IoT such as manufacturing, distribution, medical, and agriculture, and many applications are developed. In IoT applications, the knowledge of embedded software and assembly is required for precise control of IoT devices. In many applications, one gateway (GW) controls several IoT devices, but design studies according to the environments are necessary.

The expectation of applications using various heterogeneous hardware and many IoT devices is getting higher; however, the hurdles are currently high for using them. To surmount such hurdles, it will be required that application programmers will only need to write common logics to be processed, then software will adapt to the environments with heterogeneous hardware to make it easy to use such hardware and IoT devices. Java, which appeared in 1995, caused a paradigm shift in environment adaptation that allows software written once to run on another CPU machine. However, no consideration was given to the application performance at the migration porting destination.

Thus we previously proposed environment-adaptive software that effectively runs once-written applications by automatically executing code conversion and configurations so that GPUs, FPGAs, IoT devices and so on can be appropriately used on deployment environments [16]. To enable low-skilled users to take advantage of heterogeneous hardware, environment-adaptive software concept extracts appropriate offloadable parts from existing applications and offloads to heterogeneous hardware automatically. The features are "offloading existing applications", "automatic extraction of appropriate offloadable parts" and "automatic conversion suitable for hardware of deployment environment".

Basically, there are manual, semi-automatic, and automatic modification methods when offloading to heterogeneous devices, however, there are few examples

CONTACT Yoji Yamato yoji.yamato.wa@hco.ntt.co.jp Network Service Systems Laboratories, NTT Corporation, 3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585, Japan

© 2024 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. The terms on which this article has been published allow the posting of the Accepted Manuscript in a repository by the author(s) or with their consent.

of automatic modification offloading because of its difficulty. Our research aims at full automation by searching for an appropriate offload pattern through actual performance measurement. As an element technology of environment-adaptive software concept, we have developed a method for fully automatically offloading loop statements of application source code to GPUs or FPGAs [17–19]. These research can be automated but the performance improvements are not sufficient. Therefore, in this paper, the motivation and challenge are proposing a method for offloading function blocks, which are larger units, rather than individual loop statements in applications to achieve higher performance by automatic offloading to GPUs or FPGAs. We implemented the proposed method and evaluated its effectiveness of offloading several applications to GPUs or FPGAs. Of course, some applications which are not suitable for offloading to GPUs or FPGAs to improve performances are out of scope.

The rest of this paper is organized as follows. In Section 2, we review technologies in the market and our previous proposals. In Section 3, we also describe recent research. In Section 4, we present the proposed automatic offloading method for function blocks to GPUs and FPGAs. In Section 5, we explain its implementation. In Section 6, we discuss its performance evaluation and the results. In Section 7, we conclude this paper.

2. Existing technologies

2.1. Technologies in the market

Java is one example of environment-adaptive software. In Java, using a virtual execution environment called Java Virtual Machine, written software makes it possible to run even on machines of different OSes without more compiling (Write Once, Run Anywhere). However, it was not considered whether the expected performance could be attained at the porting destination, and there was too much effort involved in performance tuning and debugging at the porting destination (Write Once, Debug Everywhere). If software uses heterogeneous hardware, such as GPUs or FPGAs, the tuning effort increases.

CUDA is a major development environment for general-purpose GPUs (GPGPUs) that use GPU computational power for more than just for graphics processing. To control heterogeneous hardware and many core CPUs uniformly, OpenCL specification and its software development kit (SDK) are widely used. CUDA and OpenCL require not only C language extension but also additional descriptions such as memory copy between GPU or FPGA devices and CPUs. Because of these programming difficulties, there are few CUDA and OpenCL programmers.

For easy heterogeneous hardware programming, there are technologies that specify parallel processing areas by specified directives, and compilers transform these directives into device-oriented codes basis on specified directives. Open accelerators (OpenACC) [20] is one directive-based specification, and the Portland Group Inc.(PGI) compiler [21] is one directive-based compiler. For example, users specify OpenACC directives on C/C++ codes to process them in parallel, and the PGI compiler checks the possibility of parallel processing and outputs and deploys execution binary files to run on GPUs and CPUs. IBM JDK supports GPU offloading based on Java lambda expression [22].

In this way, CUDA, OpenCL, OpenACC, and others support GPU or FPGA offload processing. Although processing on a GPU or FPGA can be done, sufficient application performance is difficult to attain. For example, when users use an automatic parallelization technology, such as the Intel compiler [23] for multi-core CPUs, possible areas of parallel processing such as "for" loop statements are extracted. However, naive parallel execution performances with GPUs or FPGAs are not high because of overheads of CPU and GPU/FPGA memory data transfer. To achieve high application performance with GPUs/FPGAs, CUDA/OpenCL needs to be tuned by highly skilled programmers or an appropriate offloading area needs to be searched for by using the PGI compiler or other technologies.

Therefore, it is difficult to attain high application performance for users without GPU or FPGA skills. Moreover, if users use automatic parallelization technologies to obtain high performance, it takes many efforts to determine if each loop statement is parallelized, and there are many applications that cannot be improved.

This paragraph shows a basic taxonomy. There are manual, semi-automatic and automatic modification methods when offloading to heterogeneous devices. Manual modification is a method of modifying using detail languages such as CUDA and Hardware Description Language (HDL), and semi-automatic modification is a method of specifying the offload part using an instruction directive and the compiler generates an offload binary, using simple languages such as OpenMP and languages of High Level Synthesis (HLS) tools. Automatic modification is a method that analyses a normal CPU program and the tool automatically determines the offload part and generates an offload binary. However, basically, even if a machine can analyse whether a loop statement can be processed in parallel, it cannot determine whether it is suitable for parallel processing, so there are few examples of automatic modification offloading. This research challenges automatic modification.

2.2. Previous our proposals

Based on the background, to adapt software to an environment, we previously proposed environment-adaptive software [16], the processing flow of which is shown in Figure 1.

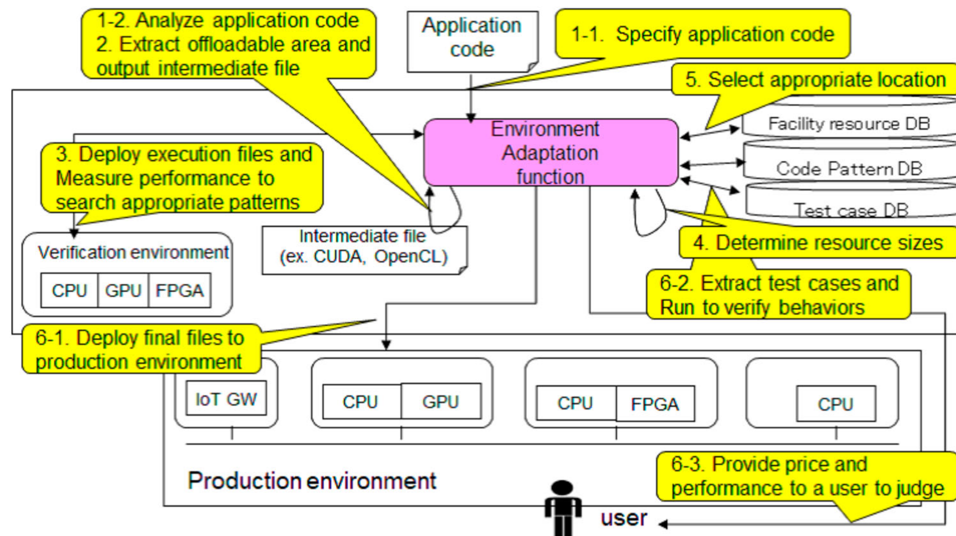


Figure 1. Processing flow of environment adaptive software.

Step 1: Code analysis
Step 2: Offloadable-part extraction
Step 3: Search for suitable offload parts
Step 4: Resource-amount adjustment
Step 5: Placement-location adjustment
Step 6: Execution-file placement and operation verification
Step 7: In-operation reconfiguration

Then, we explain our previous automatic loop-statement-offloading method for GPUs. It regarded as a elemental technology to achieve an environmental adaptation concept. We proposed using the genetic algorithms (GA) [24] to automatically find an appropriate loop statement to be offloaded to a GPU [16]. First, a parallel loop statement is checked from a general-purpose program that is not supposed to be parallelized, and loop-statement offload patterns are mapped to genes with a value of 1 for GPU execution and 0 for CPU execution. Then the performance verifications are repeated in the verification environment to search for an appropriate offloading area.

For FPGAs, since it takes more than hours to implement codes to be processed on FPGAs unlike GPUs, we carried out actual FPGA measurements after narrowing down the offload candidate loop statements [19]. For detected loop statements, a loop statement having a high arithmetic intensity is extracted using an arithmetic intensity analysis tool. Then this method precompiles the generated OpenCL codes and finds a loop statement with high resource efficiency. For narrowed-down loop statements, it generates OpenCL codes that offload each loop statement or combination of those loop statements, implements codes to be processed on FPGAs, measures application performance, and selects the highest-performance OpenCL code.

In the case of GPUs or FPGAs acceleration, however, it is often the case that an algorithm for CPUs is changed to one suitable for hardware processing. For this reason, simple offloading of loop statements is often insufficient in application performance compared to manually changing algorithms. However, it is very difficult for machines to automatically extract the hardware-oriented algorithms appropriate for each application. Therefore, this paper targets automatic offloading to GPUs or FPGAs with sufficient performances compared to previous our loop statement offloading.

3. Recent research

Wuhib et al. studied resource management and effective allocation [25] on the OpenStack cloud. e target appropriate offloading on heterogeneous hardware servers including the cloud. We previously proposed methods for selecting appropriate servers from heterogeneous hardware servers. The methods of this paper improve previous methods with automatically function block offloading.

The papers of [26–32] study offloading mobile devices processing to edge computing or other servers. Research are done to offload the processing of mobile devices to edge computing to distribute the load. The paper [33] also aims at comfortable application operation by performing processing offload from mobile devices to edge servers and reducing the processing of mobile devices with small resources based on Neuro-Fuzzy approach. The issue is how to distribute the load and it is assumed that the calculation will be processed mainly by CPUs. On the other hand, the author's research aims to offload processing that takes a long time in CPUs to heterogeneous devices such as GPUs and FPGAs, and the main issue is code conversion in different devices.

Regarding offloading to GPUs, Chen et al. [34] used metaprogramming and just-in-time (JIT) compilation for GPU offloading of C++ expression tem-

plates, Bertolli et al. [35] and Lee et al. [36] are working on offloading to GPU using OpenMP. There have been few studies on automatically converting existing code to the GPU without manually inserting new targeted directives or a new development model.

We use an OpenACC PGI compiler for C/C++ application offloading evaluations. In addition to C/C++ language, Java is often used for OSS applications. From Java 8, parallel processing can be specified by lambda expression. IBM provides a JIT compiler that offloads processing with lambda expressions to a GPU [22]. In the case of Java, we can extract an appropriate offloading area by using this JIT compiler and the proposed method checks whether function blocks and loop statements require GPU processing with a lambda expression.

Regarding FPGA offloading, Liu et al. [37] proposed a technology that offloads nested loops to FPGAs. The nested loops can be offloaded with an additional 20 minutes of tools run time. Alias et al. [38] proposed a technology in which an HLS configures an FPGA by specifying C language code, loop tiling, and so on using Altera HLS C2H. Sommer et al. [39] proposed a technology that can interpret OpenMP code and execute FPGA offloading. Putnum et al. [40] used a CPU-FPGA hybrid machine to speed up a program with a slightly modified standard C language. For FPGA offloading, instructions needed to be manually added such as which parts to parallelize using OpenMP or other specifications. There have been few studies on automatically offloading existing codes to FPGAs.

Generally, CUDA and OpenCL control intra-node parallel processing, and message passing interface (MPI) controls inter-node or multi-node parallel processing. However, MPI also requires high technical skills of parallel processing. Thus MPI concealment technology has been developed that virtualizes devices of outer nodes as local devices and enables such devices to be controlled by only OpenCL [41]. When we select multi-nodes for offloading destinations in the future, we plan to use this MPI concealment technology.

Even if an extraction of an offloading area is appropriate, application performance may not be high when the resource balance of a CPU and devices is not appropriate. For example, a CPU takes 100 seconds and GPU 1 second when one task is processed, so a CPU slows down processing. Shirahata et al . [42] attempted to improve total application performance by distributing map tasks with the same execution times of a CPU and GPU in MapReduce processing. The study by Kaleem et al. [43] is also related to task scheduling when the CPU and GPU are integrated chips of the same die. Referring to their papers, we also investigate how to deploy functions on appropriate locations and resource amounts to avoid bottlenecking CPUs or GPUs.

The papers of [44–46] are the recent papers such as for time critical applications. The work of [44] controls load balancing of CPU, GPU and FPGA devices by EngineCL which is a high-level framework based on OpenCL. The work of [45] studies a method for accelerating Web search engines using FPGA. In the search process, various pipeline processes are performed. By implementing hardware processing based on the clock timing, performance is improved by 2 orders of magnitude. The work of [46] proposes SWITCH, a middleware service for infrastructure planning and provisioning for time critical applications. According to the programming model described in the abstract layer, the cloud infrastructure is coordinated to satisfy the constraints. Although the [44]'s target of controlling heterogeneous hardware is close to our research, programming with EngineCL is needed and automatic adaptation of existing applications like this paper is not considered. As for [45], we will consider using this idea for FPGA offloading of function blocks that require real-time processing such as Web search processing. In consideration of [46], automatic conversion for each hardware has not been sufficiently studied.

Figure 2 shows a comparison of related work of heterogeneous device offloading. There are many reports on using GPUs and FPGAs to achieve high performance of applications manually with much effort such as design space exploration (DSE). Our approach is novel because our method achieves high performance with GPUs and FPGAs automatically by detecting offloadable function blocks with name matching and a similarity detection tool. Particularly, using a similarity detection tool for offloadable function detection is a new approach because similarity detection tools

	This research and previous papers [18][19]	Chen, et al., [44]	Bertolli, et al., [45]	Lee, et al., [46]	Liu, et al., [47]	Alias, et al., [48]	Sommer, et al., [49]	Putnum, et al., [50]	Guzman, et al., [54]	Owaida, et al., [55]
offload device	GPU, FPGA	GPU	GPU	GPU	FPGA	FPGA	FPGA	FPGA	GPU, FPGA	FPGA
approach	automatic	manual	semi-automatic	semi-automatic	semi-automatic	semi-automatic	semi-automatic	manual	semi-automatic	manual
target	function blocks (and loop statements with other papers)	C++ expression templates	Clang of the control loop	existing OpenMP applications	specified nested loops	loop tiling and so on	exisitng LLVM	generic ANSI-C code	load balancing of CPU, GPU and FPGA	Web search engine
methods, tools	code similarity detection, Deckard	metaprogramming and JIT compilation	OpenMP	OpenMP	soft coarse-grained reconfigurable array (SCGRA) overlay	Altera HLS C2H	OpenMP	hybrid CPU-FPGA computing platform	OpenCL	decision tree ensemble on FPGA

Figure 2. Comparison of related work of heterogeneous devices offloading.

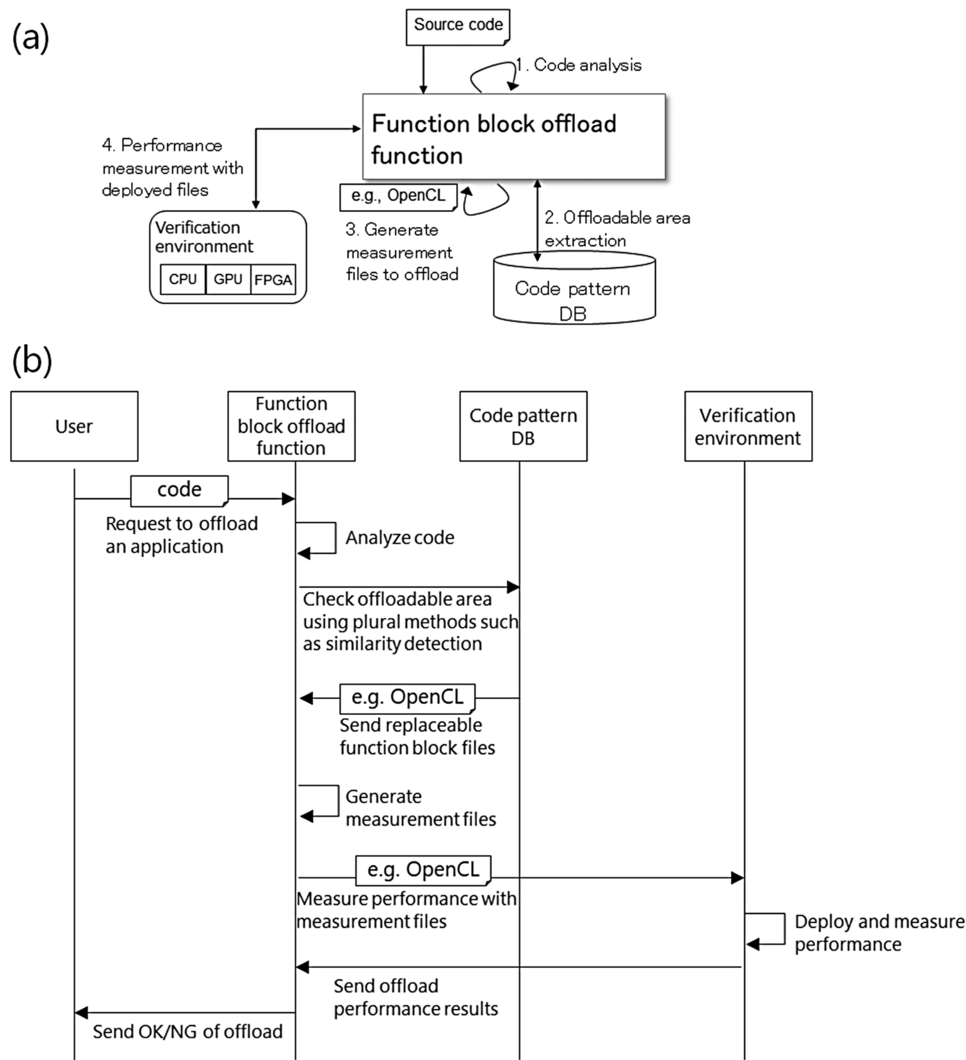


Figure 3. (a) Image of function-block offloading and (b) sequence of function-block offloading.

are basically used to detect code clones in software maintenance.

4. Proposed automatic offloading method for function blocks

We aimed to improve application performance by replacing function blocks implemented with hardware-oriented algorithms, with large units such as matrix manipulation and FT in CPU codes. This is because it is difficult to automatically extract hardware-oriented algorithms. In other words, we use the existing know-how of developers. This function block offloading is regarded as another approach to achieve an environmental adaptation concept from loop offloading because focus points of offloading small loops and offloading large function blocks are opposite.

4.1. Outline of function-block offloading and considerations

Because designing hardware circuits requires a large amount of time regarding FPGAs, it is often possible to use circuit design in the form of an "IP core" for functions already designed. Typical examples of IP cores are encryption/decryption processing, arithmetic calculations such as FFT, image processing and voice processing. Many IP cores have licensing fees, but some are offered free of charge. Since an IP core can be said to involve the existing know-how of developers, we consider using IP cores for automatic offloading to FPGAs.

For GPUs, fast FT (FFT) and matrix calculation are frequently used, and cuFFT and cuSOLVER are implemented by CUDA and provided free as GPU libraries. We considered using these libraries (not IP core) for GPUs.

If existing source code created for CPU includes function blocks that can be accelerated by offloading to GPUs or FPGAs, such as FFT processing, GPU libraries or FPGA IP cores are replaced with the function blocks to increase offloading performance.

An overview of function-block offloading is illustrated in Figure 3(a,b). Figure 3(a) shows an image of offloading behaviour and (b) shows a sequence of offloading behaviour. In Step 1, source codes are

analysed using a parse tool, such as Clang, and outer library calls and function processing are analysed with the loop-statement structure. For the library calls and function processing analysed in Step 1, function blocks that can be offloaded to GPUs or FPGAs are found by checking with the code-pattern DB in step 2. In Step 3, offloadable function blocks are replaced with libraries for GPUs or IP cores for FPGAs by creating interfaces with CPU programs. At this time, since it is not known whether function-block offloading to GPUs or FPGAs will lead to immediate increasing performance, performance measurements are repeated in a verification environment to extract faster offloading patterns with or without offloading of certain function blocks. With function block offload, the devices that are offloaded are limited to those in the same node. This is because offloading to another node device needs network processing time, so there is no merit in speeding up.

With regard to offloading of loop statements, which was done in previous studies, individual loop statement detection is carried out with a parse tool, and loop statements can be offloaded to GPUs or FPGAs using OpenACC's #pragma or OpenCL. However, with regard to function-block offloading, we need to consider the following three points; discovering function blocks in source codes, checking whether the function blocks have offloadable GPU libraries or FPGA IP cores, and matching interfaces between replaced libraries or IP cores and the host CPU program.

4.2. Function-block-offloading method

Based on the three points in the previous subsection, we developed a function-block-offloading method for this study.

A. Discovering function blocks in source codes

A-1: In parsing, the proposed method detects that external libraries are called from source codes. For parsing, parsing tools such as Clang are used. A parsing tool can detect method calls, then the proposed method checks registered external libraries list with the code-pattern DB. For example, FFT library calls are detected. The code-pattern DB holds the external libraries list beforehand.

A-2: To detect function processing other than registered library calls, classes and structures are detected from source-code-definition description by using parse tools.

B. Checking whether the function blocks have offloadable GPU libraries or FPGA IP cores

B-1: The code-pattern DB holds GPU libraries, FPGA IP cores, and related information which improve specific libraries or function-block processing. For replacement source libraries and function blocks, codes and executable files with function names are registered. For library calls detected in A-1, the proposed method searches for GPU libraries or FPGA IP cores that can be accelerated using the library name as a key.

B-2: The information registered in the code-pattern DB in B-1 is used. A similarity-detection tool detects whether there are libraries or IP cores that can be accelerated for the function processing of the classes and structures detected in A-2. A similarity-detection tool, such as Deckard [47], detects a copy code or a changed code after copying. There are many types of similarity-detection tools such as line-based detection, lexical unit-based detection, abstract syntax tree-based detection, program-dependent graph-based detection, metric and fingerprint-based detection and so on. However, detection accuracy is not 100% of all tools. Among them, Deckard uses an abstract syntax tree for detection, and it is unlikely that multiple function blocks will have the same abstract syntax tree characteristics. Such a tool can detect some codes that have similar descriptions when calculated by a CPU such as matrix manipulation, and changed descriptions after copying from other codes. It cannot detect newly created classes; thus these classes are out of the scope of this study. For functions with libraries or IP cores registered in the code-pattern DB that accelerate specific function blocks, a similarity detection tool determines whether the similarity is high based on the tool threshold. Since it is clear that the detection range with this tool is not 100%, artificial intelligence (AI) pattern recognition will also considered for detection in future. A support vector machine (SVM), which is frequently used for pattern recognition in supervised learning, deep learning and unsupervised learning, can be applied. However, even in the case of AI processing, it is difficult to make the machine understand the intention of codes, so the newly created class and structure may be difficult to detect.

C. Matching interfaces between replaced libraries or IP cores and host CPU program

C-1: Since the corresponding library or IP core is searched in B-1 for the library call detected in A-1, the replacement library or IP core is installed in the GPU or FPGA, and a host (CPU) program is connected. A library, such as CUDA, is assumed as a library for GPUs. Since methods of using CUDA libraries from C language codes are open with libraries, the code-pattern DB holds such methods as well. When GPU libraries are used, these libraries and the host program are connected referring to these methods. In an FPGA IP core, hardware description language (HDL) is assumed. The code-pattern DB also holds OpenCL code as IP-core-related information. From the OpenCL code, the connection between a CPU and FPGA using the OpenCL interface and implementation of an IP core on an FPGA can be done using high-level synthesis tools of FPGA vendors such as Xilinx and Intel (Xilinx Vivado, Intel HLS Compiler, etc.).

C-2: For classes and structures detected in A-2, we search for libraries and IP cores that can be accelerated in B-2 and implement the corresponding libraries and IP cores on the GPU and FPGA. In C-1, because it is a library or IP core that speeds up specific library calls, it is necessary to generate an interface, but the number and type of arguments and return of library or IP core and offloadable functions are matched. However, since B-2 is determined based on similarity, there is no guarantee that the basic items, such as the number and type of arguments and return, match.

If they do not match, because libraries and IP cores are existing know-how and cannot be changed frequently, we will confirm with a user on whether to change them. After receiving confirmation, we will proceed with performance tests. Regarding to differences in variable types, if we only need to cast such as float and double, we can proceed with performance tests without user confirmation. Also, when the numbers of arguments and returns differ between the source programs and libraries or IP cores, but if there is no problem even it is omitted, the proposed method does need not to notify the user and proceeds with the performance tests. For example, if arguments A and B are required and C is optional in the source program and arguments A and B are required in the library, we may treat to omit the option argument automatically. Note that if the numbers of arguments and returns are the same, we proceed the same as in C-1.

Here, we assume that cloud operators or carrier operators such as NTT who provide heterogeneous hardware for the cloud or so on prepare code-pattern DB with GPU libraries and FPGA IP cores that are useful to many users. To help users' applications speed up automatically by using the code pattern DB, the merit of operator is that the use of services such as the cloud will increase and profits will increase.

Regarding to GPU libraries, NVIDIA provides many CUDA libraries for free such as cuFFT [48] and cuSOLVER [49], and it is assumed that the operator will register them in the DB. Regarding to IP cores of FPGA, many IP cores need license fee. Therefore, it is assumed that IP cores implemented by the operator are registered in the DB. For example, the signal processing function implemented by the carrier operator is registered. Regarding to OpenCL for FPGA, there are some open releases, and it is possible to register them.

5. Implementation

5.1. Tools used

In this section, we explain the implementation of the proposed method. To confirm the method's effectiveness in function-block offloading, we used C/C++ language applications, NVIDIA Quadro P4000 as the GPU, and Intel PAC with Intel Arria10 GX FPGA as the FPGA. We also carried out compiling to the FPGA on DELL EMC PowerEdge R740.

GPU processing uses PGI compiler 19.4. This PGI compiler is an OpenACC compiler for C/C++/Fortran languages. The bytecode for the GPU can be extracted by specifying parallel processable parts, such as loop statements, by OpenACC directive #pragma acc kernels, #pragma acc parallel loop and executed on the GPU. This PGI compiler can also use CUDA libraries such as cuFFT or cuRAND.

To control the FPGA, we used Intel Acceleration Stack Version 1.2 (Intel FPGA SDK for OpenCL 17.1.1, Quartus Prime Version 17.1.1). The Intel FPGA SDK for OpenCL is a high-level synthesis tool (HLS) that compiles #pragma directives in addition to the standard OpenCL. It compiles OpenCL code that describes the kernel program processed by the FPGA and the host program processed by the CPU, outputs information such as the amount of resources, performs FPGA wiring and so on to operate the code using the FPGA. Time of implementing codes to be processed on an FPGA depends on many factors such as code size, resource amount and so on. When we use Intel FPGA SDK, even a small program of about 100 lines takes about 3 hours to be able to operate on an actual FPGA, but an error occurs early when the amount of used FPGA resources is exceeded a limit of available resources. By including the existing OpenCL codes of the FPGA into kernel codes, they can be offloaded to the FPGA after OpenCL program compiling.

We used MySQL8.0 as the code-pattern DB. Since the code-pattern DB searches for offloadable GPU library or FPGA IP core using the called library name as a key, the source library name and the corresponding destination GPU library and/or FPGA IP core names are linked and managed. The names of the libraries and IP cores that are registered are linked with the executable files of the libraries, IP cores and the codes such as CUDA and OpenCL. Since the execution file is replaced and called automatically after being detected, the usage procedure such as method name, calling order, protocol to use is also registered. Next, when the offloadable function block is detected by the similarity detection tool, the code that performs the function is also linked to the source library name. The similarity detection tool compares the registered code with the code specified by the user.

We used Deckard v2.0 [47] as the similarity-detection tool. Deckard is used to expand function blocks for offloading. It determines the similarity between the partial code to be verified and the code for comparison registered in the code-pattern DB to detect functions. Deckard can detect function blocks even if there are comments and little modification. We think Deckard is suitable for detecting offloadable function blocks. Though detecting function blocks

using abstract syntax tree similarity is a language independent method, Deckard can analyse C and Java applications.

We implemented the method with C language and Python 2.7.

5.2. Implementation behaviour

When a C/C++ application is specified, this implementation parses C/C++ code and detects loop statements for loop offloading of previous studies called libraries (A-1) and defined classes and structures (A-2). For parsing, the implementation uses parsing libraries of Clang. When the implementation searches if there is an external library call, it checks the external library list in the code-pattern DB.

Next, the implementation detects GPU libraries and FPGA IP cores that can speed up the called library (B-1). Using the called library name as a key, it obtains an executable file or OpenCL code that can be accelerated from the registered record in the code-pattern DB. If a replacement function that can be accelerated is found, the implementation then generates an executable file. In a GPU library, the implementation deletes the source part and replaces it with the found GPU library call in the C/C++ code so that the replaced CUDA library is called. In an IP core of FPGA, the implementation deletes the source part and replaces the acquired OpenCL code with the kernel code. After completing the replacements, the PGI compiler compiles for GPU and Intel Acceleration Stack for FPGA (C-1). Based on the OpenCL code, the CPU and FPGA are connected using Intel's high-level synthesis tool.

The above description is the case of a library call, and detection processing is carried out in parallel when using a similarity detection tool. In this implementation, Deckard detects the similarity between the detected partial codes such as classes and the comparison code registered in the code-pattern DB, and the comparison codes exceeding the threshold are detected (B-2). Detected codes are associated with the corresponding GPU library or FPGA IP cores. Then the implementation acquires executable files and OpenCL codes, as the same way as in B-1. Next, it generates executable files, as the same way as in C-1. However, if the interface of the source code and the replacement library or IP core arguments differ, the interface that matches the replacement library or IP core is notified to the user who requested the offload, and the user can confirm whether it can be changed. If the user accepts, the implementation generates executable files.

At this point, execution files are created that can be used to measure application performance on GPUs or FPGAs in a verification environment. For function-block offloading, if there is only one functional block to be replaced, it only considered whether that one is offloaded. However, if there are several function blocks, the implementation generates a verification pattern that offloads a certain function block to find a fast solution. This is because even if it is possible to increase the performance based on existing know-how, it will not be clear whether the speed will be increased under the deployed environment condition until application performance is actually measured. For example, if there are five function blocks that can be offloaded and the measurement results show that the performances of offloading of #2 and #4 can be improved, the implementation measures this again with the pattern of offloading both #2 and #4. If it is faster than offloading #2 and #4 separately, it selects the offloading of both as the solution.

Therefore, focusing on the processing of A, B and C as a whole, source codes are parsed, function blocks are detected, replaceable functions are found, interfaces with CPU sides are created, the performances of several offload patterns are measured in a verification environment, and high performance patterns are searched.

The degree of performance improvement by offloading function blocks is verified. The processing of A-1, B-1 and C-1 is fundamental, but to increase the number of offloadable targets, the processing of A-2, B-2 and C-2 is also verified. Using Deckard, the implementation detects codes which can be replaced to offloadable libraries or IP cores when the codes with comments and modification are similar to registered codes.

In the performance measurement, along with the processing time, the implementation checks whether the calculation result is valid or not. For example, the PCAST function of the PGI compiler can check the difference in calculation results. If the difference is large and not allowable, the implementation sets the offload pattern invalid. If no function block can be offloaded, the implementation moves on to the trial of loop statement offload which is reported in other papers [19].

6. Evaluation

6.1. Evaluation method

6.1.1. Evaluated applications

Based on the implementation, we evaluated the application performance improvement of offloading. We evaluated three applications for GPU offloading, FT, matrix calculation and random number generation, which are used in many areas such as IoT. We evaluated one application for FPGA offloading, finite-impulse response filter of signal processing. It is evaluated that not only the processing time performance but also the electric power usage can be reduced by offload. The evaluation targets are Himeno benchmark for fluid calculation for GPU offload and MRI-Q for MRI image processing for FPGA offload.

FT processing is used in various types of monitoring, such as vibration frequency analysis. When considering an IoT application that transfers data from a device to the network, it is assumed that the device side performs

primary analysis such as FFT processing to reduce network cost. To speed up FFT processing, CUDA's existing library cuFFT [48] is automatically replaced to original codes.

Matrix calculation is used in many types of analysis such as machine-learning analysis. Because matrix calculation is used not only on cloud sides but also device sides due to the spread of IoT and AI, automatic performance improvements for various applications are needed. We used matrix calculation of lower–upper (LU) decomposition. To speed up LU decomposition, CUDA's existing library cuSOLVER [49] is automatically replaced to original codes.

Random number generation is used in many areas. For example, option pricing simulation often uses random number generation. To speed up random number generation, CUDA's existing library cuRAND [50] is automatically replaced to original codes.

The time-domain finite-impulse response filter performs processing in a finite time on the output when an impulse function is input to a system. We used MIT Lincoln laboratory's high-performance embedded computing (HPEC) Challenge Benchmark Suite C code and sample tests with it for offloading performance measurement, and Intel sample OpenCL [51] is automatically replaced to original C codes. When considering applications that transfer signal data from devices over the network, to reduce network costs, it is assumed that signal processing such as filters are conducted on devices sides, thus signal processing offloading to FPGA is important, we think.

The original codes of calculation applications for GPU are from Numerical Recipes in C [52].

Himeno benchmark [53] is a performance measurement benchmark software for incompressible fluid analysis and solves Poisson's equation by the Jacobi iterative method. Himeno benchmark has C language and Fortran, but this time we decided to use Python for power measurement and described the processing logic in Python. The data is calculated in a large size of 512∗256∗256 grid. CPU processing is processed by Python's Numpy, and GPU processing is processed via the Cupy library [54] that offloads the Numpy Interface to the GPU.

MRI-Q [55] computes the matrix Q representing the scanner configuration used in the 3D MRI reconstruction algorithm in non-Cartesian space. MRI-Q is written in C language, and 3D MRI image processing is performed during performance measurement, and the processing time is measured with Large data of 64∗64∗64 size. CPU processing is in C language, and FPGA processing is processed based on OpenCL.

6.1.2. Experiment conditions

Function-block offloading to GPUs and FPGAs is combined with loop-statement offloading for actual production use. However, since loop-statement offloading has been evaluated previously [17], we only evaluate offloading of function blocks for this paper. For the target applications, we prepared function blocks that can be offloaded in the code-pattern DB beforehand and measured application performance when original codes are automatically replaced.

The experimental conditions are as follows.

Offload sources: FT, Matrix calculation, Random number generation and Time-domain finite-impulse response filter.

Offload targets: cuFFT, cuSOLVER, cuRAND and Intel OpenCL of Time-domain finite-impulse response filter.

Offload-source-discovery method: The code of the offload source application calls the external library on the code side and discovered by DB name matching. To discover by Deckard, we prepared target applications which include library codes with added comments and slight modifications to the original codes. We prepared both two patterns for this experiment.

Methods to be compared: All-CPU-processing method, Proposed function-block-offloading method and Loop-statement-offloading method.

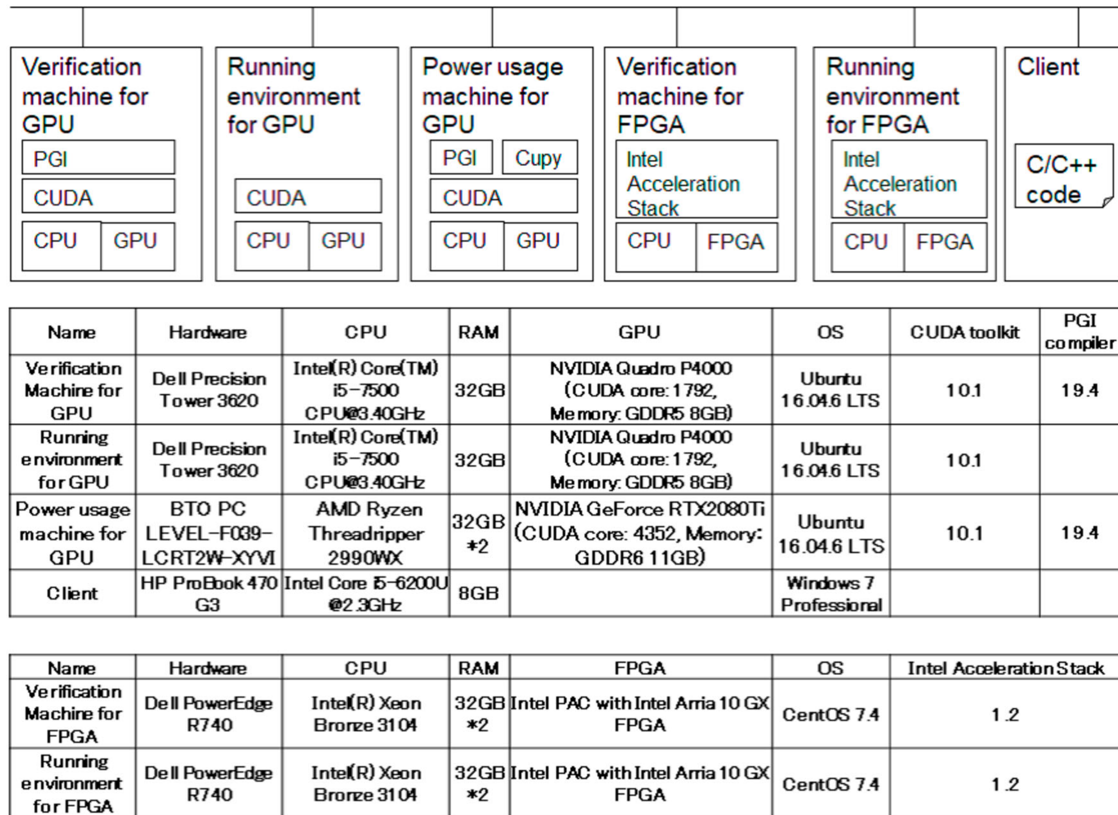
Our previous loop-statement-offloading method [17] involves using the GA to search for appropriate loop-offloading patterns in a verification environment.

Performance measurement: In FT, sample test processing is carried out with a grid size of 2048 × 2048, and the processing time is measured. In matrix calculation, the processing time of the 2048 × 2048 orthogonal matrix LU decomposition is measured. In random number generation, 2048 × 2048 × 2 float-type random numbers are generated. In time-domain finite-impulse response filter, HPEC's sample test processing is carried out with 64 filters and 4096 length of input/output vectors.

In the power usage evaluation, the processing time and power usage are measured when processing is executed after offload. The time change of the power usage is acquired and the power reduction compared to the case where all processing is executed by only CPU is confirmed.

6.1.3. Experimental environment

We used physical machines with NVIDIA Quadro P4000 for GPU offloading evaluation. The CUDA core number of NVIDIA Quadro P4000 is 1792. To control GPU, PGI compiler community edition v19.4 and CUDA toolkit v10.1 were used. We also used physical machines with Intel Arria 10 GX FPGA for FPGA offloading evaluation. To control FPGA, Intel Acceleration Stack v1.2 was used. Figure 4 shows the experimental environment and environment specifications. A client note PC specifies the C/C++ application codes, which are converted and vivificated on verification machines, and the final codes are deployed in running environments for users after verification.



Name	Hardware	CPU	RAM	GPU	OS	CUDA toolkit	PGI compiler
Verification Machine for GPU	Dell Precision Tower 3620	Intel(R) Core(TM) i5-7500 CPU@3.40GHz	32GB	NVIDIA Quadro P4000 (CUDA core: 1792, Memory: GDDR5 8GB)	Ubuntu 16.04.6 LTS	10.1	19.4
Running environment for GPU	Dell Precision Tower 3620	Intel(R) Core(TM) i5-7500 CPU@3.40GHz	32GB	NVIDIA Quadro P4000 (CUDA core: 1792, Memory: GDDR5 8GB)	Ubuntu 16.04.6 LTS	10.1	
Power usage machine for GPU	BTO PC LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX	32GB *2	NVIDIA GeForce RTX2080Ti (CUDA core: 4352, Memory: GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	19.4
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 7 Professional		

Name	Hardware	CPU	RAM	FPGA	OS	Intel Acceleration Stack
Verification Machine for FPGA	Dell PowerEdge R740	Intel(R) Xeon Bronze 3104	32GB *2	Intel PAC with Intel Arria 10 GX FPGA	CentOS 7.4	1.2
Running environment for FPGA	Dell PowerEdge R740	Intel(R) Xeon Bronze 3104	32GB *2	Intel PAC with Intel Arria 10 GX FPGA	CentOS 7.4	1.2

Figure 4. Experimental environment.

For power usage measurement, LEVEL-F039-LCRT2W-XYVI of BTO PC was used as GPU, and Dell PowerEdge R740 of verification machine was used as FPGA. GPU power usage is measured by NVIDIA's nvidia-smi and CPU power usage is measured by s-tui [56]. For the FPGA, Intelligent Platform Management Interface (IPMI) tool of the Dell server measured the power of the entire server and compares power usage when using the FPGA and when using only the CPU.

6.2. Performance results

We confirmed the performance improvements of three applications to GPU and one application to FPGA that are expected to be used by many users.

Figure 5 shows an example of FT performance improvement on our previous study using GA [17]. It shows maximum performance change of FT in each generation with GA generation transitions (the vertical axis shows how many times faster GPU offloading was than using only a CPU). FT performance improved and GPU offloading was about 5.4 times faster. During GA processing, it took more than several hours to search for appropriate offloading loop statements.

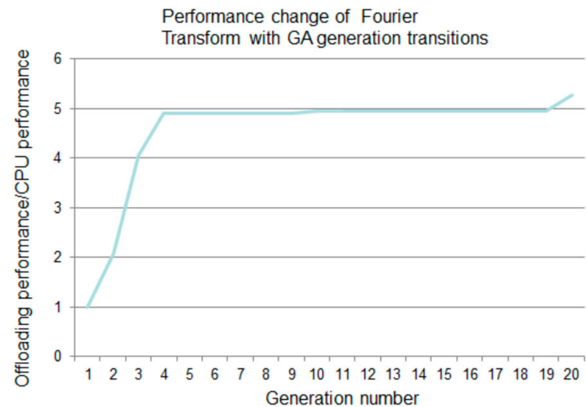


Figure 5. Reference graph: performance change of FT with GA generation transitions [17].

Based on these previous results, we present the measurement results of how much these applications improved with the proposed method. The offload-source-discovery method can be replaced with the DB name matching and similarity detection tool depending on whether the library is called or the code is copied. Table 1 shows how much the four applications improved in performance with the proposed function-block offloading method compared to the all-CPU-processing method. 1 means that the applications performed the same with both methods. The performance improvements with the loop-statement-offloading method are also shown. FT improved in performance 730 fold with the proposed method, which was only 5.4 times that with the loop-statement-offloading method. Matrix calculation improved 130,000 fold with the proposed method compared to 38 fold with the loop-statement-offloading method. Random number generation, it was found that

Table 1. Comparison of application performance improvement between loop-statement-offloading and proposed function-block-offloading methods.

	Performance improvement of loop statement offloading	Performance improvement of function block offloading
Fourier transform (to GPU)	5.4	730
Matrix calculation (to GPU)	38	130,000
Random number generation (to GPU)	19	27
Time domain finite impulse response filter (to FPGA)	4.0	21

the improved 27 fold with the proposed method compared to 19 fold with the loop-statement-offloading method. Time-domain finite-impulse response filter execution time was shortened from 0.298 to 0.0139 seconds, thus the processing performance improved 21 fold with the proposed method compared to 4.0 fold with the loop-statement-offloading method. From these four measurements, function block offload performance is superior to loop statement offload performance. Because CUDA libraries are implemented suitable algorithms for specific calculations, they are much faster than simply offloading loop statements when function block offloading can be conducted. In addition, GPU library and IP core are built in when using, so there is no delay in external calls when function block offloading can be conducted.

For these four cases, the offloading of function blocks was completed in a few seconds because our implementation only checked DB name matching and similarity detection. When code size is larger, Deckard may take more time, but it will complete in 1 minute.

Figure 6(a) shows watt and time when the Himeno benchmark was offloaded to the GPU. Compared to all CPU processing, the processing time is shortened from 153 to 19 seconds, but it can be seen that the power usage is about 27–109 watts. As a result, watt ∗ sec is about 1/2 of 2070 watts ∗ sec from 4080 watt ∗ sec in the case of all CPU processing.

Figure 6(b) shows watt and time when MRI-Q was offloaded to the FPGA. Compared to all CPU processing, the processing time has been shortened from 14 to 2 seconds, and it can be seen that the power usage of the entire server is about 121–111 watts. As a result, watt ∗ sec has changed from 1690 watt ∗ sec for all CPU processing to 223 watt ∗ sec, which is about 1/8.

We confirmed power usage reduction in GPU and FPGA offload applications. At the time of GPU offload of Himeno benchmark, watt was increased, but the power usage could be reduced as a whole due to the effect of shortening the processing time. At the time of FPGA offload of MRI-Q, watt was reduced, which was combined with the shortening of time, and the power usage could be greatly reduced. It is generally said that FPGAs have good power efficiency, and this time it was confirmed that the power usage of FPGAs is low in the MRI-Q experiment. Therefore, if the GPU and FPGA have similar performance when offloaded in a mixed environment, it is conceivable to select the FPGA by looking at the power usage.

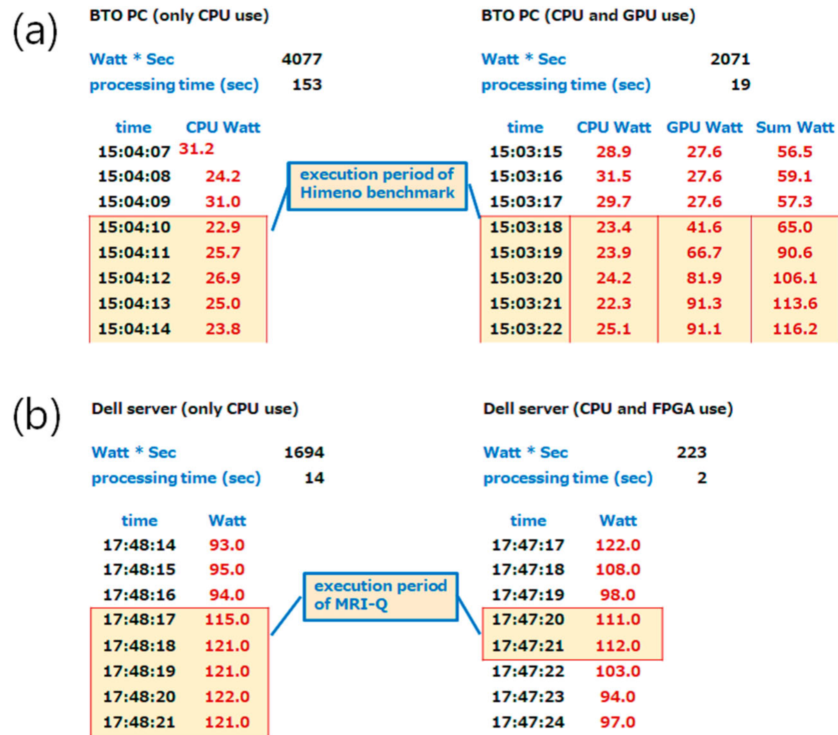
6.3. Discussion

In our previous study on loop-statement offloading to GPUs and FPGAs, we used a method of measuring the performance of multiple offloading patterns in a verification environment and searching high-performance patterns automatically. For example, even large applications with more than 100 loop statements, such as Darknet, are automatically offloaded to GPUs and triple in performance. However, there have been many cases in which the performance improved by more than 10 fold through manual program development using CUDA for GPUs and HDL for FPGAs and in some cases automatic performance improvement was insufficient. Therefore, the proposed method can achieve high performance with specific processing, such as FFT and matrix calculation, by offloading them to the CUDA library or IP core.

Function-block offloading can greatly contribute to performance improvement, particularly for FPGAs because FPGAs often require algorithms that are suitable for hardware processing compared to GPUs.

Regarding the offloading effect with costs of hardware, GPU boards, such as NVIDIA Quadro, cost about 2 000 USD and FPGA boards, such as Intel Arria, cost about 3 000 USD. Therefore, hardware for GPUs or FPGAs costs about twice to three times as much as that for only CPUs. In data centres, hardware, development, and verification costs of systems, such as a cloud systems, are about a 1/3 the total cost, electricity and operation/maintenance cost is more than 1/3, and other expenses, such as service orders, is the other 1/3. Regarding AWS, a GPU instance with one GPU costs about 650 USD/month, which is the same as hosting a general dedicated server. Therefore, we believe improving application performance more than 10 times that take much time will have a sufficiently positive cost effect even though the hardware cost is about twice to three times. For example in this paper's experiment, the GPU server was 6 000 USD the FPGA server was 9 000 USD, and the CPU server was about 3 000 USD. All applications offloaded to GPU or FPGA has more than three times the performance improvement, therefore, all applications can be said cost-effective with automatic offloading.

Regarding the time to start production services, in only function-block offloading, it is assumed that processing will be completed in a few minutes. When there is no function block that can be offloaded, application performance improves using the loop-statement-offloading method, and the application performance is increased from several hours to half day through



(a)

BTO PC (only CPU use)

Watt * Sec 4077
processing time (sec) 153

time	CPU Watt
15:04:07	31.2
15:04:08	24.2
15:04:09	31.0
15:04:10	22.9
15:04:11	25.7
15:04:12	26.9
15:04:13	25.0
15:04:14	23.8

execution period of Himeno benchmark

BTO PC (CPU and GPU use)

Watt * Sec 2071
processing time (sec) 19

time	CPU Watt	GPU Watt	Sum Watt
15:03:15	28.9	27.6	56.5
15:03:16	31.5	27.6	59.1
15:03:17	29.7	27.6	57.3
15:03:18	23.4	41.6	65.0
15:03:19	23.9	66.7	90.6
15:03:20	24.2	81.9	106.1
15:03:21	22.3	91.3	113.6
15:03:22	25.1	91.1	116.2

(b)

Dell server (only CPU use)

Watt * Sec 1694
processing time (sec) 14

time	Watt
17:48:14	93.0
17:48:15	95.0
17:48:16	94.0
17:48:17	115.0
17:48:18	121.0
17:48:19	121.0
17:48:20	122.0
17:48:21	121.0

execution period of MRI-Q

Dell server (CPU and FPGA use)

Watt * Sec 223
processing time (sec) 2

time	Watt
17:47:17	122.0
17:47:18	108.0
17:47:19	98.0
17:47:20	111.0
17:47:21	112.0
17:47:22	103.0
17:47:23	94.0
17:47:24	97.0

Figure 6. (a) Power usage of Himeno benchmark after GPU offload and (b) power usage of MRI-Q after FPGA offload.

repeated verification. When we provide production services, we provide the first day for free and try to speed up the verification environment during the first day, and from the second day we provide the production service using GPUs and FPGAs. Therefore, we believe the tuning time is acceptable.

To offload with the proposed method, it is necessary to pre-register the library and IP core that are commonly used in multiple applications to the code-pattern DB. In addition to the processing used in various applications, such as FFT and matrix manipulation, it is assumed that registration will be focused on specific fields such as machine learning and signal processing.

In this evaluation, it was confirmed that offloadable functions of copied codes with comments can be found using a similarity detection tool Deckard. Similarity detection tools are tools for discovering code clones in software maintenance phase originally, therefore, using a similarity detection tool for automatic offloading is a new approach. In the software engineering field, similarity detection is a hot topic and new methods are proposed frequently. Therefore, we will study to detect more function blocks that can be offloaded using recent studies such as applying Artificial Intelligence (AI) methods of SVM and deep learning in the future.

Though it is difficult to detect offloadable function block rather than offloadable individual loop, expectable effect is high because function block offloading can use hardware oriented algorithms. Therefore, for actual service phase, providers need to provide both approaches of function blocks and loop statements for automatic offloading to GPUs or FPGAs.

7. Conclusion

We proposed an automatic offloading method for function blocks of applications as a new element of our environment-adaptive software. Environment adaptive software adapts applications to the environments to use heterogeneous hardware such as GPUs and FPGAs appropriately.

The proposed method starts with source code analysis. It analyses the source code, detects offloadable library calls by checking a DB, and replaces them with replaceable GPU libraries or FPGA IP cores registered in the DB. The performance is measured in a verification environment, including the functions of the replaced GPU and FPGA, and took the pattern with the highest performance as the solution. To search for more replaceable function blocks in source-code analysis, offloadable function blocks are also searched for using similarity detection technology. Replacement and performance measurement are carried out as the same way. However, even if it is determined that the function block can be replaced, if the interface is different, the user is asked whether it can be changed with the interface of the replaceable function. We implemented the proposed method, evaluated its automatic offloading of several applications to GPUs or FPGAs, and confirmed its effectiveness. Compared to the loop statement offload of the previous research, the function block offload of this research is 40% or more for the GPU offload of random number generation, 100 times or more for the GPU offload of Fourier transform, and 5 times or more for the FPGA offload of time-domain finite-impulse response filter have been improved.

For future work, we will investigate a common method for appropriately offloading existing CPU applications including offloadable function blocks and loop statements in an environment where GPUs, FPGAs and many-core CPUs are mixed. We will also study ways to improve cost-effectiveness by adjusting the amount of processing resources of CPU, GPU and FPGA when the migration destination environment is mixed.

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

[1] Shahidi G. Slow-down in power scaling and the end of Moore's law? International Symposium on VLSI Design, Automation and Test, 2019.
[2] Putnam A, Caulfield AM, Chung ES, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In: Proceedings of the 41th Annual International Symposium on Computer Architecture (ISCA'14); 2014. p. 13–24.
[3] Yamato Y. Use case study of HDD–SSD hybrid storage, distributed storage and HDD storage on openStack. In: 19th International Database Engineering & Applications Symposium (IDEAS'15); 2015. p. 228–229.
[4] Yamato Y, Nishizawa Y, Nagao S. Fast restoration method of virtual resources on OpenStack. In: IEEE Consumer Communications and Networking Conference (CCNC2015); 2015. p. 607–608.
[5] Yamato Y, et al. Development of resource management server for production IaaS services based on OpenStack. J Inf Process. 2015;23(1):58–66.
[6] Yamato Y. Proposal of optimum application deployment technology for heterogeneous IaaS cloud. In: 2016 6th International Workshop on Computer Science and Engineering (WCSE 2016); 2016. p. 34–37.
[7] Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Boston: Addison-Wesley; 2011.
[8] Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng. 2010;12(3):66–73. doi:10.1109/MCSE.2010.69
[9] Hermann M, Pentek T, Otto B. Design principles for industrie 4.0 scenarios. Dortmund: Rechnische Universitat Dortmund; 2015.
[10] Yamato Y, Fukumoto Y, Kumazaki H. Proposal of shoplifting prevention service using image analysis and ERP check. IEEJ Trans Electr Electron Eng. 2017;12(S1):141–145. doi:10.1002/tee.2017.12.issue-S1
[11] Yamato Y, Takemoto M, Shimamoto N. Method of service template generation on a service coordination framework. In: 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004); 2004.
[12] Noguchi H, Demizu T, Hoshikawa N, et al. Autonomous device identification architecture for internet of things. In: 2018 IEEE 4th World Forum on Internet of Things (WF-IoT 2018); 2018. p. 407–411.
[13] Noguchi H, Kataoka M, Yamato Y. Device identification based on communication analysis for the internet of things. IEEE Access. 2019;7:52903–52912. doi:10.1109/Access.6287639
[14] Noguchi H, Demizu T, Kataoka M, et al. Distributed search architecture for object tracking in the internet of things. IEEE Access. 2018;6:60152–60159. doi:10.1109/ACCESS.2018.2875734
[15] Yamato Y, Fukumoto Y, Kumazaki H. Proposal of real time predictive maintenance platform with 3D printer for business vehicles. Int J Inf Electron Eng. 2016;6(5):289–293.
[16] Yamato Y. Study of parallel processing area extraction and data transfer number reduction for automatic GPU offloading of IoT applications. J Intell Inf Syst. 2019;54:567–584. doi:10.1007/s10844-019-00575-8
[17] Yamato Y. Study and evaluation of improved automatic GPU offloading method. Int J Parallel Emergent Distrib Syst. 2021;36(6):594–608. doi:10.1080/17445760.2021.1941010
[18] Yamato Y. Study and evaluation of automatic GPU offloading method from various language applications. Int J Parallel Emergent Distrib Syst. 2021;37(1):22–39. doi:10.1080/17445760.2021.1971666
[19] Yamato Y. Automatic offloading method of loop statements of software to FPGA. Int J Parallel Emergent Distrib Syst. 2021;36(5):482–494. doi:10.1080/17445760.2021.1916020
[20] Wienke S, Springer P, Terboven C, et al. OpenACC-first experiences with real-world applications. In: Euro-Par Parallel Processing; 2012.
[21] Wolfe M. Implementing the PGI accelerator model. In: ACM the 3rd Workshop on General-purpose Computation on Graphics Processing Units; 2010. p. 43–50.
[22] Ishizaki K. Transparent GPU exploitation for Java. In: The Fourth International Symposium on Computing and Networking (CANDAR 2016); 2016.
[23] Su E, Tian X, Girkar M, et al. Compiler support of the workqueuing execution model for Intel SMP architectures. In: Fourth European Workshop on OpenMP; 2002.
[24] Holland JH. Genetic algorithms. Sci Am. 1992;267(1):66–73. doi:10.1038/scientificamerican0792-66
[25] Wuhib F, Stadler R, Lindgren H. Dynamic resource allocation with management objectives – implementation for an OpenStack cloud. In: Proceedings of network and service management; 2012 8th International Conference and 2012 Workshop on Systems Virtualization Management; 2012. p. 309–315.
[26] Shakarami A, Shahidinejad A, Ghobaei-Arani M. An autonomous computation offloading strategy in mobile edge computing: a deep learning-based hybrid approach. J Netw Comput Appl. 2021;178:Article ID 102974. doi:10.1016/j.jnca.2021.102974
[27] Jazayeri F, Shahidinejad A, Ghobaei-Arani M. Autonomous computation offloading and auto-scaling the in the mobile fog computing: a deep reinforcement learning-based approach. J Ambient Intell Humaniz Comput. 2021;12:8265–8284. doi:10.1007/s12652-020-02561-3
[28] Shakarami A, Ghobaei-Arani M, Shahidinejad A. A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective. Comput Netw. 2020;182:Article ID 107496. doi:10.1016/j.comnet.2020.107496
[29] Shakarami A, Ghobaei-Arani M, Masdari M, et al. A survey on the computation offloading approaches in mobile edge/cloud computing environment: a stochastic-based perspective. J Grid Comput. 2020;18:639–671. doi:10.1007/s10723-020-09530-2

[30] Shakarami A, Shahidinejad A, Ghobaei-Arani M. A review on the computation offloading approaches in mobile edge computing: a game-theoretic perspective. Softw Pract Exper. 2020;50:1719–1759. doi:10.1002/spe.v50.9
[31] Shahidinejad A, Ghobaei-Arani M, Masdari M. Resource provisioning using workload clustering in cloud computing environment: a hybrid approach. Cluster Comput. 2021;24:319–342. doi:10.1007/s10586-020-03107-0
[32] Aslanpour MS, Dashti SE, Ghobaei-Arani M, et al. Resource provisioning for cloud applications: a 3-D, provident and flexible approach. J Supercomput. 2018;74:6470–6501. doi:10.1007/s11227-017-2156-x.
[33] Anitha S, Padma T. A Neuro-Fuzzy hybrid framework for augmenting resources of mobile device. Int J Inf Technol Decis Mak. 2021;20:1519–1555. doi:10.1142/S0219622021500413
[34] Chen J, Joo B, Watson III W, et al. Automatic offloading C++ expression templates to CUDA enabled GPUs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum; 2012. p. 2359–2368.
[35] Bertolli C, Antao SF, Bercea GT, et al. Integrating GPU support for OpenMP offloading directives into Clang. In: ACM Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM'15); 2015.
[36] Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'09); 2009.
[37] Liu C, Ng H-C, So HK-H. Automatic nested loop acceleration on fpgas using soft CGRA overlay. In: Second International Workshop on FPGAs for Software Programmers (FSP 2015); 2015.
[38] Alias C, Darte A, Plesco A. Optimizing remote accesses for offloaded kernels: application to high-level synthesis for FPGA. In: 2013 Design, automation and test in Europe (DATE); 2013. p. 575–580.
[39] Sommer L, Korinth J, Koch A. OpenMP device offloading to FPGA accelerators. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017); 2017. p. 201–205.
[40] Putnam A, Bennett D, Dellinger E, et al. CHiMPS: a C-level compilation flow for hybrid CPU-FPGA architectures. In: IEEE 2008 International Conference on Field Programmable Logic and Applications; 2008. p. 173–178.
[41] Shitara A, Nakahama T, Yamada M, et al. Vegeta: an implementation and evaluation of development-support middleware on multiple opencl platform. In: IEEE Second International Conference on Networking and Computing (ICNC 2011); 2011. p. 141–147.
[42] Shirahata K, Sato H, Matsuoka S. Hybrid map task scheduling for GPU-based heterogeneous clusters. In: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom); 2010. p. 733–740.
[43] Kaleem R, Barik R, Shpeisman T, et al. Adaptive heterogeneous scheduling for integrated GPUs. In: 2014 IEEE 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT); 2014. p. 151–162.
[44] Guzman MAD, Nozal R, Tejero RG, et al. FPGA heterogeneous execution with EngineCL. J Supercomput. 2019;75(3):1732–1746. doi:10.1007/s11227-019-02768-y
[45] Owaida M, Alonso G, Fogliarini L, et al. Lowering the latency of data processing pipelines through FPGA based hardware acceleration. Proc VLDB Endow. 2019;13(1):71–85. doi:10.14778/3357377.3357383
[46] Stefanic P, Cigale M, Jones AC, et al. SWITCH workbench: a novel approach for the development and deployment of time-critical microservice-based cloud-native applications. Future Gener Comput Syst. 2019;99:197–212. doi:10.1016/j.future.2019.04.008
[47] Deckard web site. Available from: http://github.com/skyhover/Deckard
[48] cuFFT web site. Available from: https://docs.nvidia.com/cuda/cufft/index.html
[49] cuSOLVER web site. Available from: https://docs.nvidia.com/cuda/cusolver/index.html
[50] cuRAND web site. Available from: https://docs.nvidia.com/cuda/curand/index.html
[51] Time domain finite impulse response filter web site. Available from: https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/td-fir.html
[52] Numerical Recipes in C. Available from: https://www.cec.uchile.cl/cinetica/pcordero/MC_libros/NumericalRecipesinC.pdf
[53] Himeno benchmark website. Available from: http://accc.riken.jp/en/supercom/
[54] Cupy website. Available from: https://cupy.dev/
[55] MRI-Q website. Available from: http://impact.crhc.illinois.edu/parboil/
[56] s-tui website. Available from: https://github.com/amanusk/s-tui