Title	Study and Evaluation of Automatic GPU Offloading Method from Various Language Applications
Author(s)	Yoji Yamato
Citation	Journal title(Repository name etc.),Volume,Issue, Pages(Article number) etc. ・ジャーナル名（刊行物・サイト名）・巻号・ページ（その他論文番号等）： International Journal of Parallel, Emergent and Distributed Systems ・DOI (URL) https://doi.org/10.1080/17445760.2021.1971666 Publication Date: yyyy/mm/dd ・出版日：2021 年 9 月 6 日 Publisher ・出版者：Taylor and Francis
Declaration	This preprint is the Author Accepted Manuscript(AAM) of the above. ・本プレプリントは、上記論文の (受理済)著者最終稿(AAM) である。 All necessary permissions from the publisher have ・ジャーナル（出版者）から必要な許諾を been obtained / not been obtained ◉ 得ている ○ 得ていない
Notes	"This is an Accepted Manuscript of an article published by Taylor & Francis in International Journal of Parallel, Emergent and Distributed Systems, on 6th Sep., 2021, available online: https://doi.org/10.1080/17445760.2021.1971666."

ORIGINAL PAPER

Study and Evaluation of Automatic GPU Offloading Method from Various Language Applications

Yoji Yamato[a]

[a]Network Service Systems Laboratories, NTT Corporation, 3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585, Japan

ARTICLE HISTORY
Compiled August 10, 2021

ABSTRACT
Heterogeneous hardware other than a small-core central processing unit (CPU) is increasingly being used, such as a graphics processing unit (GPU), field-programmable gate array (FPGA), or many-core CPU. However, to use heterogeneous hardware, programmers must have sufficient technical skills to utilize OpenMP, CUDA, and OpenCL. On the basis of this, we previously proposed environment-adaptive software that enables automatic conversion, configuration, and high performance operation of once-written code, in accordance with the hardware to be placed. However, the source language for offloading was mainly C/C++ language applications, and there was no research into common offloading for various language applications. In this paper, for a new challenge, we study a common method for automatically offloading for various language applications in not only C language but also Python and Java. We evaluate the effectiveness of the proposed method in multiple applications of various languages.

KEYWORDS
Environment Adaptive Software; GPGPU; Automatic Offloading; Various Languages; Performance

1. Introduction

As Moore's Law slows down, the transistor density of a central processing unit (CPU) cannot be expected to double every 1.5 years. To compensate for this, more systems are using heterogeneous hardware, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs). For example, Microsoft's search engine Bing uses FPGAs [1], and Amazon Web Services (AWS) provides GPU and FPGA instances using cloud technologies (e.g., [2]-[7]).

However, to properly utilize devices other than CPUs in these systems, configurations and programs must be made that consider device characteristics, such as Open Multi-Processing (OpenMP) [8], Open Computing Language (OpenCL) [9], and Compute Unified Device Architecture (CUDA) [10]. Therefore, for most programmers, skill barriers are high. In addition, Internet of Things (IoT) devices (e.g., [11]-[16]) are increasingly being used in the system, so required skills are also increasing.

The expectations are becoming higher for applications using heterogeneous hardware and Internet of Things (IoT) devices; however, the skill hurdles for using them

CONTACT Yoji Yamato Email: yoji.yamato.wa@hco.ntt.co.jp

are currently high. To surmount these barriers and use heterogeneous hardware easily and effectively, a platform is needed on which developers only write logics to be processed so that software can adapt to the deployed environments with heterogeneous hardware such as GPU and FPGA by automatic conversion and configuration.

Java, which appeared in 1995, caused a paradigm shift in environment adaptation that enables software written once to run on another CPU machine. However, the application performance at the porting destination was not considered. Therefore, we previously proposed environment-adaptive software that effectively runs once-written applications by automatically converts and configures code so that GPUs, FPGAs, many-core CPUs, and so on can be appropriately used in deployment environments [17]. For elemental technologies for environment-adaptive software, we also proposed methods for automatically offloading loop statements and function blocks of C language applications to GPUs or FPGAs [18][19].

The purpose of this paper is to automatically offload the application regardless of whether the migration source language is C language, Python, or Java. First, we propose the same common way of automatic offloading flow for all languages. Next, we cleary design language-dependent processing when the source language is C language, Python, or Java. We evaluate the effectiveness of the proposed method in existing plural applications with three languages.

The rest of this paper is organized as follows. In Section 2, we review technologies on the market and our previous proposals. In Section 3, we present the proposed automatic offloading method for various language applications. In Section 4, we explain its implementation. In Section 5, we discuss its performance evaluation and the results. In Section 6, we describe related work, and in Section 7, we conclude the paper.

2. Existing technologies

2.1. Technologies on the market

Java is one example of environment-adaptive software. In Java, using a virtual execution environment called Java Virtual Machine, written software can run even on machines that use different operating systems (OSes) without more compiling (Write Once, Run Anywhere). However, whether the expected performance could be attained at the porting destination was not considered, and too much effort was involved in performance tuning and debugging at the porting destination (Write Once, Debug Everywhere).

CUDA is a major development environment for general-purpose GPUs (GPGPUs) (e.g., [20]) that use GPU computational power for more than just graphics processing. To control heterogeneous hardware uniformly, the 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.

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 on the basis of specified directives. Open accelerators (OpenACC) [21] and OpenMP are examples of directive-based specifications, and the Portland Group Inc. (PGI) compiler [22] and gcc are examples of compilers that support these directives.

In this way, CUDA, OpenCL, OpenACC, OpenMP, and others support GPU,

FPGA, or many-core CPU offload processing. Although processing on devices 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 devices are not high because of overheads of CPU and device memory data transfer. To achieve high application performance with devices, CUDA, OpenCL, or so on needs to be tuned by highly skilled programmers, or an appropriate offloading area needs to be searched for by using the OpenACC compiler or other technologies.

Therefore, users without skills in using GPU, FPGA, or many-core CPU will have difficulty attaining high application performance. Moreover, if users use automatic parallelization technologies to obtain high performance, much effort is needed to determine whether each loop statement is parallelized. As an effort to automate trial and error in a parallel processing area search, we previously proposed automatic GPU offloading using an evolutionary computation method.

2.2. Previous proposals

To adapt software to an environment, we previously proposed environment-adaptive software, the processing flow of which is shown in Figure 1. The environment-adaptive software is achieved with an environment-adaptation function, test-case database (DB), code-pattern DB, facility-resource DB, verification environment, and production environment.

Step 1: Analyze code
Step 2: Extract offloadable part
Step 3: Search for suitable offload parts
Step 4: Adjust resource amount
Step 5: Adjust placement location
Step 6: Place execution-file and verify operation
Step 7: Reconfigure in-operation

In Steps 1-7 of the processing flow, code is converted, the resource amount is adjusted, the placement location is adjusted, and in-operation reconfiguration is conducted for environment adaptation. However, only some steps can be selected. For example, if we only want to convert code for a GPU, FPGA, or many-core CPU, we only need to conduct Steps 1-3.

In short, manual offloading for heterogeneous devices is currently the mainstream. We proposed the concept of environment-adaptive software and was considering automatic offloading of C language software to GPU and FPGA, but various migration source languages such as Python and Java were not considered. Therefore, in this paper, we will focus on automatic offload when the source language is diverse.

3. Study of automatic GPU offloading from various source languages

To embody the concept of environment-adaptive software, we have proposed automatic GPU and FPGA offload of loop statement [17][18] and automatic offload of function block [24] for C language program so far. Moreover, we have also proposed the offload method when the migration destination environment is mixed.

On the basis of these elemental technologies, we define the target for the source

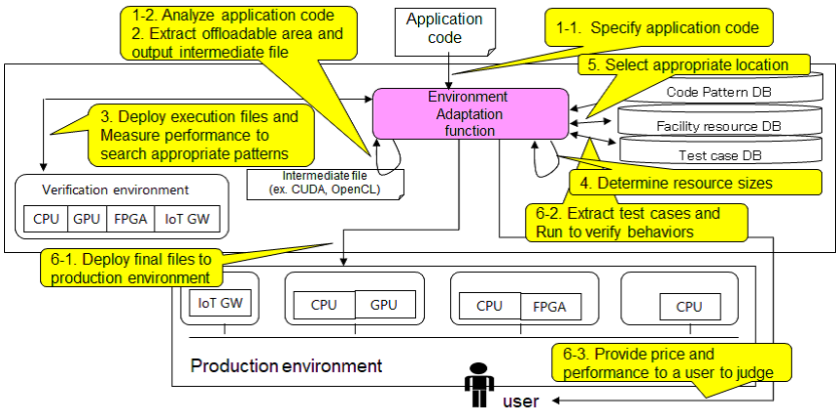


Figure 1. Processing flow of environment-adaptive software

language and describe the basic idea to follow even if the target diversifies in 3.1. In 3.2, we propose an automatic offload method that can be used even when the source language diversifies. In 3.3, we propose applying the common method to each source language.

3.1. Basic ideas for offloading from various languages

The various source languages covered in this paper are C language, Python, and Java. These are the top three in the popularity ranking of programming languages announced by TIOBE every month [25] and have large numbers of dedicated programmers. In addition, C language is a compiled language, Python is an interpreted language, and Java is an intermediate language, so they cover all the various processing methods. Therefore, if the method can be used in common with these three languages, more languages will be easy to support.

To automatically offload at high-speed using heterogeneous devices such as GPUs from various languages, we gradually search for high-speed offload patterns with evolutionary calculation by measuring the performance on a physical machine in a verification environment. The reason for this is that the performance varies greatly with not only the code structure but also the actual processing contents such as the specifications of the processing hardware, compiler or interpreter, the data size, and the number of loops. Therefore, the performance is difficult to predict statically, and dynamic measurements should be conducted. There is already an automatic parallelizing compiler on the market such as [23] that finds loop statements and parallelizes them at the compile stage. However, performance often needs to be measured because parallelization of parallelizable loop statements often results in low speed.

As for the objects to be offloaded, we focus on the loop statement and function block of the program. Loop statements are the first target for offloading because most processing of programs that take a long time is spent in loops. On the other hand, with regard to function blocks, when speeding up specific processing, an algorithm suitable for the processing content and processing hardware is often used, so there is a case where processing can be greatly speeded up compared with offloading individual loop statements [24]. The performance of programs are improved by replacing frequently used function blocks such as matrix calculation and Fourier transform with CUDA

libraries implemented with algorithms suitable to GPU.

We have considered three migration destination environments (GPU, FPGA, and many-core CPU) and have also considered offloading C language programs to a mixed environment. Since the migration source language will be diversified this time, only GPU is set as the migration destination to be evaluated. If we can validate the common method on GPU from various languages, FPGA and many-core CPU adaptation will be extensions.

3.2. Common automatic GPU offloading method

In this subsection, we describe a common automatic GPU offloading method of loop statements in 3.2.1 and function blocks in 3.2.2. Both two approaches are conducted in automatic GPU offloading.

3.2.1. Automatic GPU offloading of loop statements

For automatic GPU offloading of loop statements, several methods [17][18] have been proposed for C language.

First, as a basic problem, the compiler can find the limitation that this loop statement cannot be processed in parallel on the GPU, but it is difficult to find out whether this loop statement is suitable for parallel processing on the GPU. Loops with a large number of iterations are generally said to be more suitable, but it is difficult to predict how much performance will be achieved by offloading to the GPU without actually measuring it. Therefore, it is often the case that the instruction to offload this loop to the GPU is manually given and the performance measurement is tried. On the basis of that, [17] proposes automatically finding an appropriate loop statement that is offloaded to the GPU with a genetic algorithm (GA), which is an evolutionary computation method. From a general-purpose program for normal CPUs that does not assume GPU processing, the proposed method first checks the parallelizable loop statements. Then for the parallelizable loop statements, it sets 1 for GPU execution and 0 for CPU execution. The value is set and geneticized, and the performance verification trial is repeated in the verification environment to search for an appropriate area. By narrowing down to parallel processing loop statements and holding and recombining parallel processing patterns that can be accelerated in the form of gene parts, patterns that can be efficiently accelerated are explored from the huge number of parallel processing patterns.

The work of [18] proposes transferring variables efficiently. Regarding the variables used in the nested loop statement, when the loop statement is offloaded to the GPU, the variables that do not have any problem even if CPU-GPU transfer is performed at the upper level are summarized at the upper level. This is because if CPU-GPU transfer is performed at the lower level of the nest, the transfer is performed at each lower loop, which is inefficient. We also proposed a method to further reduce CPU-GPU transfers. Specifically, for not only nesting but also variables defined in multiple files, GPU processing and CPU processing are not nested, and for variables where CPU processing and GPU processing are separated, the proposed method specifies to transfer them in a batch.

Common methods of loop statement offload include code analysis, understanding of loops and variables, geneticization of loops with GPU processing and CPU-GPU transfer instructions, coding of genes, compilation, performance measurement in verification environment, and creation of next-generation genes. These processes are iteratively ex-

ecuted, and the final solution is determined.

3.2.2. Automatic GPU offloading of function blocks

The automatic offload method for function blocks for C language was proposed in [24]. The outline of the function block offload process is explained in Fig. 2. In Step 1, the source code is analyzed. By using a parsing tool of syntax analysis such as Clang, the proposed method analyzes the library call and function processing included in the code along with the loop statement structure. For the library call and function processing parsed in Step 1, the proposed method discovers the processing that can be offloaded to GPU by checking with the code-pattern DB in Step 2. In Step 3, the proposed method replaces the process that can be offloaded with a library for GPU and offloads while creating an interface with the CPU program. At this time, it is not known whether the processing that can be offloaded to GPU libraries actually improves performance, but by trying to determine whether or not the function block is offloaded, a faster pattern is automatically extracted. In addition to matching the names of libraries, the search for a library that speeds up function processing is also performed by a similarity-detection tool. The similarity-detection tool detects copy code and code changed slightly after copying, such as Deckard [26] and CloneDigger [27].

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 and Clone Digger use abstract syntax trees for detection because we think offloadable multiple function blocks will have the same abstract syntax tree characteristics. Of course, grammar and description policy are different for each language such as comments and variable definitions. Since abstract syntax trees are used by both Deckard and CloneDigger, the differences in comments or so on are handled with the same abstract syntax trees. In addition, both Deckard and CloneDigger judge the similarity with the comparison code registered in the DB with specified logic for each language. For example, even if the variable type is not defined in the DB code because it is dynamically typed in Python, it can be compared if the type is not defined in the input code of Python.

As evaluated in [24], the function block offload is faster than the offload of individual loop statements because it tunes for specified processing including specialized algorithms.

Common methods of function block offload include code analysis, function block understanding, search for offloadable function blocks by name matching and similarity detection, replacement with offloadable function blocks, compilation, and performance measurement in a verification environment. These processes are iteratively executed, and the final solution is determined.

3.3. Study of adapting each source language

In this subsection, we examine the automatic offload to GPU described in the previous subsection to adapt each source language of C language, Python, and Java.

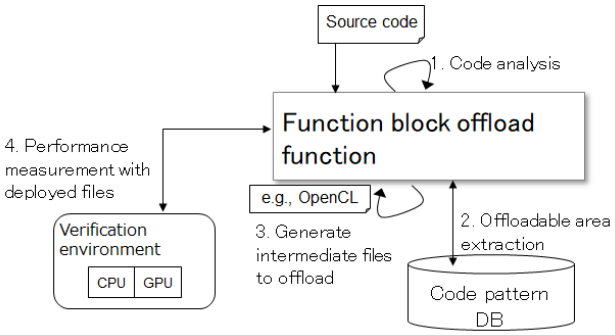


Figure 2. Image of function block offloading

3.3.1. Method for C language

Since the GPU automatic offload evaluation so far has been performed by the C language program, there is no new consideration for the C language. In this subsection, the basic flow can be language-independent in a common method, but the types of processing that depend on and do not depend on C language are described.

In the loop statement offload code analysis, parsing is performed using a parsing library such as Clang that analyzes C language. As for understanding loops and variables, when managing the results of parsing tools, it can be managed abstractly and language-independently. The geneticization of the loop with and without GPU processing is also language-independent. In the coding of genes, to create the code to be executed on the GPU in accordance with the gene patterns, GPU processing and variable transfer are specified by OpenACC, which is an extended grammar of C language. Regarding compile process, the proposed method compiles the OpenACC code with an OpenACC compiler such as a PGI compiler. Performance is measured by using an automatic test tool for C language. In the next-generation gene creation, the goodness of fit is set in accordance with the performance measurement result, and processing such as crossover and mutation is performed, but these are language-independent. Iterative execution and final solution determination are also language-independent.

As described above, in the loop statement offload, it is considered that the loop and variable management and the gene processing of GA can be applied language-independently.

In the code analysis of function block offload, parsing is performed using a parsing library such as Clang that analyzes C language. The result is managed as an abstract function block because it is used for the matching search. In the search for offloadable function blocks, the search is performed by matching by the name of a library and by detecting similarities by using a similarity-detection tool of a C language function block such as Deckard. It is necessary to replace the offloadable function block with processing such as calling the CUDA library. Regarding compile process, the proposed method compiles C language code with CUDA library call by using a PGI compiler. Performance is also measured by using an automatic test tool for C language. When there are multiple offloadable function blocks, they are executed repeatedly, and the pattern with the highest performance is determined as the final solution.

As described above, in the function block offload, it is considered that the management of the function block and the checking by the name matching of the function block can be applied language-independently.

3.3.2. Method for Python

In this subsection, the basic flow can be language-independent in a common method, but the types of processing that depend on and do not depend on Python are described.

In the code analysis, parsing is performed by using a parsing library such as ast that analyzes Python.

In the coding of genes, the processing part is converted into the Numpy interface [28] first, and variable transfer and calculations on GPU are specified by using the Cupy [29] library. We use pyCUDA [30] as the interpreter, which interprets and executes CUDA, and also use the Cupy library, which issues CUDA code itself.

In the search for offloadable function blocks by similarity detection, we use a similarity-detection tool of CloneDigger [27]. It is necessary to replace the offloadable function block with processing such as calling the CUDA library. Regarding interpretation, the proposed method interprets Python by using pyCUDA.

3.3.3. Method for Java

In this subsection, the basic flow can be language-independent in a common method, but the types of processing that depend on and do not depend on Java are described.

In the code analysis, parsing is performed using a parsing library such as Java Parser that analyzes Java.

In the coding of genes, GPU processing and variable transfer are specified by CUDA expression. Regarding the running environment, the proposed method uses JCUDA [31], which is Java bindings for CUDA and can execute GPU processing on the basis of CUDA expression.

In the search for offloadable function blocks by similarity detection, we use a similarity-detection tool of Deckard [26]. It is necessary to replace the offloadable function block with processing such as calling the CUDA library. Regarding the running environment, the proposed method uses JCUDA.

4. Implementation

4.1. Tools to use

In this section, we explain the implementation of the proposed method. To evaluate the method's effectiveness, we use C language, Python, and Java applications.

Regarding GPU, we use NVIDIA GeForce RTX 2080 Ti and NVIDIA Quadro K5200. To control the GPU, CUDA Toolkit 10.1 is commonly used. We use PGI compiler 19.10, which is an OpenACC compiler for C language, pyCUDA 2019.1.2 and Cupy 7.8 for Python (which are CUDA interpreter and CUDA calling library), and JCuda 10.1 for Java, which binds Java with the CUDA runtime libraries.

For language program parsing, we use parsing libraries of LLVM/Clang 6 libClang Python binding for C language, ast for Python, and Java Parser for Java.

We use Deckard v2.0 [26] as the similarity-detection tool for C language and Java. It determines the similarity between the partial code to be verified and the code for comparison registered in the code-pattern DB to detect offloadable functions by using abstract syntax tree similarity. We use CloneDigger [27] for Python. CloneDigger checks an abstract syntax tree extracted by ast and determines similarity on the basis of anti-unification.

We use MySQL8.0 as the code-pattern DB. It holds the record for searching a library

or IP core that can be speeded up, using the calling library name as a key. At the same time, the correspondence with the comparison code for detecting the library by the similarity-detection tool is also retained.

We implemented the proposed method with Perl 5 and Python 2.7. Perl controls GA processing, and Python controls other processing such as parsing.

4.2. Implementation behavior

The operation outline of the implementation is explained here. The implementation analyzes the code by using parsing libraries when there is a request to offload the applications. Next, offload verifications for function blocks offload and loop statement offload are tried. This is because the function block offloading, which offloads in accordance with the processing content including the algorithm, can be faster than loop statement offloading. If function block offloading is possible, latter loop statement offloading is verified for the code without the function block that is offloadable. The implementation selects the best performance from all verifications.

4.2.1. Function block offload verifications

The implementation parses the program structure such as the library being called, the defined classes, and structures.

Next, the implementation detects the GPU library that can speed up the called library. With the library being called a key, the executable library files that can be speeded up are acquired from the records registered in the code-pattern DB. When a library file that can speed up is found, the implementation creates the executable file. The original part is deleted and replaced so that the replacement CUDA library is called. After replacement, the implementation compiles.

The previous paragraph described the case of calling the library. The processing is also performed in parallel when the similarity-detection tool is used. The implementation uses Deckard or CloneDigger to detect the similarity between the partial code such as the detected class and the comparison code registered in the DB, and it discovers the function block whose similarity exceeds the threshold and the corresponding GPU library. In particular, if the replacement source code and the interface of the replacement library such as arguments, return values, and types are different, the implementation changes the interface in accordance with the replacement destination library after confirming that the user does not mind. After replacement, the implementation creates an execution file.

Here, an execution file is created that can measure performance with the GPU in the verification environment. For offloading multiple function blocks, we will measure the performance of each replacement function block and determine whether it can be speeded up.

4.2.2. Loop statement offload verifications

The implementation analyzes the code of the program, discovers the loop statement, and grasps the program structure such as the variable data used in the loop statement.

Since loop statements that cannot be processed by the offload device itself need to be eliminated, the implementation tries inserting instructions to be processed by the GPU for each loop statement and excludes loop statements that generate errors from the GA process. Here, if the number of loop statements with no error is A, then A is

the gene length.

Next, as an initial value, the implementation prepares genes with a specified number of individuals. Each value of the gene is created by randomly assigning 0 and 1. Depending on the prepared gene value, if the value is 1, an instruction is inserted into the code that specifies GPU processing for corresponding loop statements.

Data transfers between CPU and GPU are also specified. On the basis of the reference relation of variable data, the implementation instructs the data transfer. If the variables set and defined on the CPU program side and the variables referenced on the GPU program side overlap, the variables need to be transferred from the CPU to the GPU. Moreover, if the variables set on the GPU program side and the variables referenced, set, and defined on the CPU program side overlap, the variables need to be transferred from the GPU to the CPU. Among these variables, if variables can be transferred in batches before and after GPU processing, the implementation inserts an instruction that specifies that variables be transferred in batches.

The implementation compiles the code in which the instructions are inserted. It deploys the compiled executable files and measures the performances. 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 processing time to a huge value.

After measuring the performance of all individuals, the goodness of fit of each individual is set in accordance with the processing time. The individuals to be retained are selected in accordance with the set values. GA processing such as crossover, mutation, and copy is performed on the selected individuals to create next-generation individuals.

For the next-generation individuals, instruction insertion, compilation, performance measurement, goodness-of-fit setting, selection, crossover, and mutation processing are performed. After completing the specified number of generations of GA, the code with instructions that correspond to the highest performance gene is taken as the solution.

4.3. Common part and individual part for various languages

As described in Section 3, function block offload has a common part for function block management and matching by function block name, and loop statement offload has a common part for loop and variable management and GA gene processing.

4.3.1. Implementation for C language

Tool usage and directive settings for the compiler are implemented depending on C language. Clang is used for parsing to understand the structure of loop statements and function blocks. For offloading function blocks, replacement functions are searched for by using Deckard, and if offloading is possible, the library for the corresponding CUDA is called, changed codes are compiled by using the PGI compiler, and performance is measured by using Jenkins.

Next, for offloading the loop statement, the loop pattern is geneticized. GPU processing is specified by #pragma acc kernels and #pragma acc parallel loop of OpenACC, and #pragma acc data copy and #pragma acc data present specify whether to transfer data or not. OpenACC code corresponding to the gene pattern is compiled by using the PGI compiler, and performance is measured by using Jenkins. Creation and repetition of next-generation gene patterns in GA are performed by the common

part.

4.3.2. Implementation for Python

Ast is used for parsing to understand the structure of loop statements and function blocks. For offloading functional blocks, replacement function search is performed using CloneDigger, and if offloading is possible, the library for the corresponding CUDA is called, changed codes are interpreted by using pyCUDA, and performance is measured by using Jenkins.

Next, when offloading the loop statement, the loop pattern is geneticized and GPU processing is controlled by CUDA. The content calculated on the GPU is first converted from the loop statement into the Numpy interface of matrix representation. For CUDA command issuing, Cupy, which is a CUDA processing library with a Numpy compatible interface, is used. The matrix representation is converted into a CUDA command via Cupy, it is specified to PyCUDA, and PyCUDA executes GPU processing. Moreover, performance is measured by using Jenkins.

4.3.3. Implementation for Java

Java Parser is used for parsing to understand the structure of loop statements and function blocks. For offloading function blocks, the replacement function is searched for by using Deckard, and if offloading is possible, the corresponding GPU library is called and processed by JCUDA, and performance is measured by using Jenkins.

Next, for offloading the loop statement, the loop pattern is geneticized and parallel processing is specified by CUDA. Loop statements corresponding to 1 of gene values are converted into CUDA expression and executed by JCUDA. When converting CUDA expressions, detail memory control is not specified, but simple calculating on GPU with global memory copy is. Performance is measured by using Jenkins.

5. Evaluation

Since the automatic offload of loop statements to GPU are evaluated by [18], and the automatic offload of function blocks to GPU is evaluated by [24], in this paper, we demonstrate that the application can be properly offloaded even if the source language is diverse, not the effect of GPU offload.

5.1. Evaluation method

5.1.1. Evaluated applications

The evaluation targets are Fourier transform, fluid calculation, and matrix multiplication, which are expected to be used by many users. For Python and Java, we use applications in which the C language codes are modified for Python and Java with the same processing logic.

In an IoT environment, fast Fourier transform (FFT) is often necessary for monitoring such as vibration of sensors. NAS.FT [32] calculates three-dimensional FFT. When considering an application that transfers data from a device to a cloud via a network, automatic offloading is expected in which the device side executes primary analysis such as FFT processing to reduce network traffic. We used a sample test of

Fourier transform, which is equipped in NAS.FT. Sample test parameters of grid size are 256*256*128, and the number of iterations is 6.

Himeno benchmark [33] is a benchmark for incompressible fluid analysis, which solves the Poisson equation by using the Jacobi iteration method. It is frequently used for manual GPU acceleration including CUDA; thus, we used Himeno benchmark to verify whether it can be accelerated with our automatic offloading method. Data size for comparison was LARGE (512*256*256). We also pre-checked SSMALL, SMALL, MIDDLE size data to confirm size effect.

Simple matrix multiplication is used in many types of analysis such as machine-learning analysis. Because matrix calculation is used on not only cloud sides but also device sides due to the spread of artificial intelligence (AI), various applications need automatic performance improvements. For an experiment, we used polybench 3mm (3 matrix multiplications) in which three matrix multiplications were calculated with a size of 4000*4000 for compparison. (EXTRALARGE_DATASET: NI=4000, NJ=4000, NK=4000, NL=4000, NM=4000) [34]. We also pre-checked MINI, SMALL, STANDARD, LARGE size data to confirm size effect.

5.1.2. Experimental conditions

Applications with codes of three languages are input into the implemented automatic offload function, and the implementation tries offloading to GPU and measures each application performance. The experiments achieve a degree of improvement over the case where every processing is conducted by a normal CPU without offloading. Experiments also show that the implemented function can offload to the GPU even in different languages.

The experimental conditions are as follows.

Offload applications and loop statements number: Fourier transform block NAS.FT that has 65 loops, fluid calculation Himeno benchmark that has 10 loops, and matrix multiplications 3mm that has 10 loops.

The experimental conditions of the GA for loop statement offload are as follows. Number of individuals M: No more than the gene length. 20 for NAS.FT, 10 for Himeno, and 10 for 3mm.

Number of generations T: No more than the gene length. 20 for NAS.FT, 10 for Himeno, and 10 for 3mm.

Goodness of fit: $(\text{Processing time})^{-1/2}$. When processing time becomes shorter, the goodness of fit becomes larger. By setting the power of (-1/2), we prevent the search range narrowing due to the goodness of fit being too high for specific individuals with short processing times. If the performance measurement does not complete in 3 minutes, a timeout is issued, and processing time is set to 1000 seconds to calculate goodness of fit. If the calculation result is largely different from the result of the original codes, the processing time is also set to 1,000 seconds.

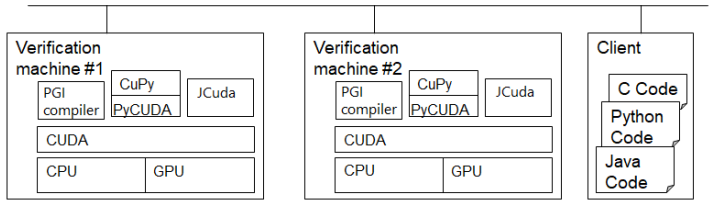
Selection algorithm: Roulette selection and Elite selection. Elite selection means that one gene with maximum goodness of fit must be reserved for the next generation without crossover or mutation.

Crossover rate Pc: 0.9

Mutation rate Pm: 0.05

The experimental conditions of function block offload are as follows.

Offload targets: Nothing in this experiment. There is a GPU library that speeds up Fourier transform and matrix calculation such as cuFFT and cuSOLVER, but this time we will not prepare a function block offload destination because we want



Name	Hardware	CPU	RAM	GPU	OS	CUDA toolkit	Compiler, etc
Verification Machine #1	LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX (32 Core)	32GB *2	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory: GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	PGI compiler 19.10, pyCUDA 2019.1.2, Cupy 7.8, JCuda 10.1
Verification Machine #2	Dell New Vostro 15 5000	Intel Xeon E5-2620 v3@2.40GHz	32GB	NVIDIA Quadro K5200 (CUDA core: 2304, Memory: GDDR5 8GB)	Ubuntu 16.04.6 LTS	10.1	PGI compiler 19.10, pyCUDA 2019.1.2, Cupy 7.8, JCuda 10.1
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 10 Pro		

Figure 3. Experiment environment

to validate offloading from various migration source languages. Before experiments, codes and libraries of random number generation and time domain impulse response filter were registered in MySQL DB but these were not used in this experiments.

5.1.3. Experimental environment

Regarding GPU, we use NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory: GDDR6 11GB) and NVIDIA Quadro K5200 (CUDA core: 2304, Memory: GDDR5 8GB). We use CUDA Toolkit 10.1 for GPU control. We use PGI compiler 19.10 for C language, pyCUDA 2019.1.2 and Cupy 7.8 for Python, and JCuda 10.1 for Java. Fig. 3 shows the evaluation environment and specifications. Here, the application code used by the user is specified from the client notebook PC, tuned using the verification machine, and then deployed to the running environment for the actual use.

5.2. Performance results

As an application that is expected to be used by many users, we evaluated automatic GPU offload from three languages of Fourier transform, fluid calculation, and matrix multiplication.

Fig. 4 shows an example of speeding up the Fourier transform of C language using verification machine #2. Regarding NAS.FT, since the function block offload could not be performed, the loop statement offload was tried and the performance change at that time is shown. Fig. 4 shows the maximum performance of each generation and the number of GA generations. Y axis shows Offloading performance / CPU performance and 1 means the same performance of only CPU processing. In Fig. 4, the CPU processing took 31.3 seconds, but the 10th generation processed in 5.8 seconds and achieved more than 5 times the performance compared to only CPU processing in final results. In addition, the same gene pattern with high fitness often occurs in GA, and the offload extraction process can be completed within 4 hours.

Before three language comparison, we pre-checked the size effect of data size change. The larger the data size, the greater the effect of improvement by automatic GPU offloading. This is because the positive effect of shortening the calculation time by increasing the amount of calculation calculated by the GPU rather than the nega-

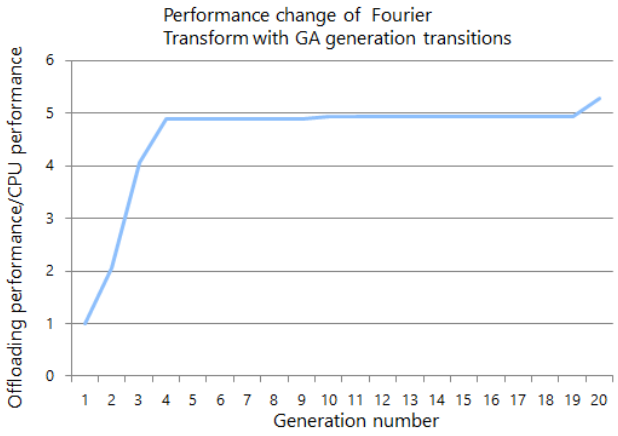


Figure 4. Reference graph: performance change of NAS.FT with GA generation [18]

tive effect of increasing the data transfer amount between CPU and GPU. It is also said that the greater the number of loop iterations, the greater the effect of GPU offload improvement. The evaluated applications do not explicitly specify the number of loop iterations, but as the size increases, the number of loop iterations increases by multiplication accordingly. Followings are examples of Java results.

For Himeno benchmark, the data sizes are SSMALL 32*32*64, SMALL 64*64*128, MIDDLE 128*128*256 and LARGE 256*256*512. On the other hand, CPU execution time, JCUDA execution time, and performance improvement ratio are as follows. SSMALL 0.43 sec, 0.017 sec, 26 times, SMALL 3.8 sec, 0.030 sec, 130 times, MIDDLE 35 sec, 0.13 sec, 270 times, LARGE 290 sec, 0.90 sec, 320 times. For 3mm, the data sizes are MINI 32*32, SMALL 128*128, STANDARD 1024*1024, LARGE 2000*2000, EXTRALARGE 4000*4000. On the other hand, CPU execution time, JCUDA execution time, and performance improvement ratio are as follows. MINI 0.0065 sec, 0.00086 sec, 7.6 times, SMALL 0.024 sec, 0.0021 sec, 11 times, STANDARD 6.9 sec, 0.018 sec, 380 times, LARGE 130 sec, 0.070 sec, 1900 times, EXTRALARGE 1700 sec, 0.43 sec, 3900 times.

Table 1 shows the degree of performance improvement of applications with all three languages using verification machine #1. Regarding the absolute value of the processing time, the performances of applications are different with three languages. Therefore, Table 1 shows the execution time of only CPU processing, execution time of offloading and degree of improvement ratio compared with when processing is performed only by the CPU in each language. In case of Python, performance on the only CPU is the result of calculation on the CPU using the Numpy library, and performance of offloading is the result of calculation on the GPU via PyCUDA using Cupy library. This is because using Numpy library is major way for calculation in Python case, so Numpy performance is set as 1. The degree of improvement for the Fluid calculation of Himeno benchmark is 4.8 times for C language, 21 times for Python, and 320 times for Java. The degree of improvement for matrix multiplication of 3mm is 440 times for C language, 11 times for Python, and 3,900 times for Java. Searching with a genetic algorithm takes within 4 hours for all applications. The degree of improvement varies depending on the language, but any language can achieve automatic acceleration with GPU offload.

	C language			Python			Java		
	execuion time of only CPU	execuion time of offloading	improve ratio	execuion time of only CPU	execuion time of offloading	improve ratio	execuion time of only CPU	execuion time of offloading	improve ratio
Himeno benchmark	4.7 sec	0.97 sec	4.8	150 sec	7.2 sec	21	290 sec	0.90 sec	320
Polybench 3mm	86 sec	0.19 sec	440	1.5 sec	0.14 sec	11	1700 sec	0.43 sec	3900

Table 1. Performance comparison of applications with three languages

5.3. Discussion

In our previous study on C language loop statement offloading to GPUs, we used a method to gradually search for high performance patterns automatically. For example, even large applications with more than 100 loop statements, such as Darknet, are automatically offloaded to GPUs and treble in performance. However, since various applications are not limited to C language, heterogeneous devices need to be able to be used for general purposes even in languages with a large programmer population such as Python and Java. The proposed method automatically offloads to the GPU so that the migration source language can be either C language, Python, or Java and can achieve more than four times the performance.

Regarding the offloading effect with costs, GPU boards cost about 500-4,000 USD. Therefore, a server with a GPU board costs about two times as much as that for only CPU. In data center systems such as a cloud systems, hardware, development, and verification costs are about a 1/3 of the total cost; electricity and operation/maintenance costs are more than 1/3; and other expenses, such as service orders, are less than 1/3. Therefore, we think automatically improving application performance by more than four times will have a sufficiently positive cost effect even though the hardware cost is about two times higher.

Regarding the time to start production services, in this verification example, verification took about several hours for GA search. When we provide production services, we provide the first day for free and try to speed up using the verification environment during the first day, and from the second day, we provide the production service with many-core CPUs, GPUs, and FPGAs. Therefore, we think one-day verification is acceptable. If the user wants to use a service immediately, on the basis of the verification order of the proposal, the user may use the service after completing the verification with only the function block offload or with many-core CPU and GPU.

Regarding the time to start deployed services, the convergence time is based on GA, and it took about six hours this time. Performance measurement from compilation takes about three minutes, and it takes time to search for a solution depending on the number of individuals and generations, but compilation and measurement of the same gene pattern are omitted, so it is completed in six hours. When providing the service, providers will provide a trial use for the first day, will verify speeding up in the verification environment during the first day, and will be able to provide the production service using GPU as well from the second day. Therefore, taking one day to start the actual service is acceptable.

To offload function blocks, it is expected to pre-register the well-known libraries commonly used by many applications in the DB by providers. In addition to general-purpose processing such as FFT and matrix multiplication, functions are assumed to be registered in specific fields such as machine learning and signal processing. We have already confirmed that code that is copied and slightly modified can be found by similarity-detection tools using the abstract syntax tree technique. 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 support vector machine (SVM) and deep learning in the future.

Regarding loop statement offload to GPUs, to search for the offload part in a shorter time, the performance can be measured in parallel for multiple verification machines. In addition, although the crossover rate Pc is set to a high value of 0.9 and a wide range is searched to find a certain performance solution early, parameter tuning is also conceivable. For higher speeds, the difficulty of automation increases, but it is conceivable to appropriately perform memory processing such as proper use of multiple memories, coreless access, and branch suppression in Warps using CUDA.

In terms of languages, C language, Python, and Java are the three most common, with about half the market share. This time, we validated the proposed method on three programming languages, but script language and intermediate languages can be automatically offloaded by the same common method. Therefore, when supporting different languages, the common parts such as gene processing are retained, and we have to implement the individual function processing corresponding to each language.

6. Related work

The method of [35] searches for GPU offloading areas and also uses GA to search automatically. However, its target is specific applications for which many GPU-based methods have been researched to accelerate like the Himeno benchmark of fluid calculation, and a huge number of tunings is needed such as calculations for 200 generations. Our previously proposed technology, which is also used this study, aims to be able to start using a general-purpose application for a CPU in a certain time when accelerating it with a GPU.

Some studies focused on offloading to GPUs [36][37][38]. Chen et al. [36] used metaprogramming and just-in-time (JIT) compilation for GPU offloading of C++ expression templates. Bertolli et al. [37] and Lee et al. [38] investigated offloading to GPUs using OpenMP. There have been few studies on automatically converting existing code into a GPU without manually inserting new directives or a new development model, which we target.

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 in 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 [39]. When we select multi-nodes for offloading destinations, 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 between 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 processing. Shirahata et al. [40] attempted to improve total application performance by distributing Map tasks with the same execution times of CPUs and GPUs in MapReduce processing. The study by Kaleem et al. [41] is also related to task scheduling when a CPU and GPU are integrated chips of the same die. Referring to their papers, we study how to deploy functions on appropriate locations and resource amounts to avoid bottlenecks of CPUs or GPUs.

Regarding FPGA offloading, Liu et al. [42] proposed a technology that offloads

nested loops to FPGAs. The nested loops can be offloaded with an additional 20 minutes of manual work. Alias et al. [43] proposed a technology in which an HLS configures an FPGA by specifying C language code, loop tiling, and so on using Altera HLS C2H. Putnum et al. [44] used a CPU-FPGA hybrid machine to speed up a program with a slightly modified standard C language. There have been many studies on FPGA offloading, but instructions needed to be manually added such as which parts to parallelize using OpenMP or other specifications.

Many methods have been proposed to improve performances by offloading to GPU, FPGA, and many-core CPU, but most approaches involve manually adding an instruction such as which part to parallelize like OpenMP directives. Few methods automatically offload existing codes. In addition, most methods consider only one type of device as an offloading destination, and few methods offload to mixed environments of GPU, FPGA, and many-core CPU, which is the subject of this paper.

There are many works to speed up by offloading to GPU, FPGA, many-core CPU, but efforts are needed such as adding instructions of OpenMP manually that specify parts to parallelize and offload. There are few works to automatically offload existing code. In addition, there are many studies on offloading C language applications. However, there is no study on common methods and language-dependent processing in various migration source languages such as C language, Python, and Java, which is the subject of this paper.

7. Conclusion

In this paper, we proposed a method that can be automatically offloaded regardless of whether the migration source language is C language, Python, or Java. This method is an element of environment-adaptive software for automatically adapting software in accordance with the deployment destination environment and appropriately using a graphics processing unit (GPU), a field-programmable gate array (FPGA), and a many-core central processing unit (CPU) to operate applications with high performance.

First, we set the GPU as the migration destination and examined a common method for automatically offloading to the GPU regardless of the migration source language. For loop statements, the loop statement offload pattern suitable for offloading to the GPU is automatically searched for by an evolutionary computation method, and transfer specifications are specified to reduce unnecessary CPU-GPU transfers in accordance with each offload pattern. For function blocks, the function blocks that can be offloaded to the GPU are found by name matching of library calls and similarity detection of function blocks, and the speed is increased by replacing them with the already implemented GPU libraries calls. Next, when processing the common method for each of C language, Python, and Java, we consider language-dependent / independent processing, and separate loop, variable, functional block management and gene processing as language-independent common processing parts. In addition, we also implemented language-dependent processing using tools and compilers on the market.

It was demonstrated that GPU automatic offload by the proposed method can be performed to speed up existing plural applications even in various languages such as C language, Python, and Java, and the effectiveness of the method was shown. There are not many applications that can be used as same as for three languages. Therefore, we would like to consider the comparison of three languages in more applications as a future task. 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

The author declares no conflicts of interest associated with this manuscript.

Notes on contributors

Yoji Yamato received a B.S. and M.S. in physics, and a Ph.D. in general systems studies from the University of Tokyo in 2000, 2002, and 2009. He joined NTT in 2002, where he has been conducting developmental research on a cloud computing platform, an IoT platform and a technology of environment adaptive software. Currently, he is a distinguished researcher of NTT Network Service Systems Laboratories. Dr. Yamato is a senior member of IEEE and IEICE, and a member of IPSJ.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," Proceedings of the 41th Annual International Symposium on Computer Architecture (ISCA'14), pp.13-24, June 2014.

[2] O. Sefraoui, M. Aissaoui and M. Eleuldj, "OpenStack: toward an open-source solution for cloud computing," International Journal of Computer Applications, Vol.55, No.3, 2012.

[3] Y. Yamato, "Automatic verification technology of software patches for user virtual environments on IaaS cloud," Journal of Cloud Computing, Springer, 2015, 4:4, DOI: 10.1186/s13677-015-0028-6, Feb. 2015.

[4] Y. Yamato, "Automatic system test technology of virtual machine software patch on IaaS cloud," IEEJ Transactions on Electrical and Electronic Engineering, Vol.10, Issue.S1, pp.165-167, Oct. 2015.

[5] Y. Yamato, "Proposal of Optimum Application Deployment Technology for Heterogeneous IaaS Cloud," 2016 6th International Workshop on Computer Science and Engineering (WCSE 2016), pp.34-37, June 2016.

[6] Y. Yamato, "Use case study of HDD-SSD hybrid storage, distributed storage and HDD storage on OpenStack," 19th International Database Engineering & Applications Symposium (IDEAS15), pp.228-229, July 2015.

[7] Y. Yamato, Y. Nishizawa, S. Nagao and K. Sato, "Fast and Reliable Restoration Method of Virtual Resources on OpenStack," IEEE Transactions on Cloud Computing, DOI: 10.1109/TCC.2015.2481392, Sep. 2015.

[8] T. Sterling, M. Anderson and M. Brodowicz, "High performance computing : modern systems and practices," Cambridge, MA : Morgan Kaufmann, ISBN 9780124202153, 2018.

[9] J. E. Stone, D. Gohara and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," Computing in science & engineering, Vol.12, No.3, pp.66-73, 2010.

[10] J. Sanders and E. Kandrot, "CUDA by example : an introduction to general-purpose GPU programming," Addison-Wesley, 2011.

[11] M. Hermann, T. Pentek and B. Otto, "Design Principles for Industrie 4.0 Scenarios," Rechnische Universitat Dortmund. 2015.

[12] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Proposal of Shoplifting Prevention Service Using Image Analysis and ERP Check," IEEJ Transactions on Electrical and Electronic Engineering, Vol.12, Issue.S1, pp.141-145, June 2017.
[13] Y. Yamato, "Proposal of Vital Data Analysis Platform using Wearable Sensor," 5th IIAE International Conference on Industrial Application Engineering 2017 (ICIAE2017), pp.138-143, Mar. 2017.
[14] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Security Camera Movie and ERP Data Matching System to Prevent Theft," IEEE Consumer Communications and Networking Conference (CCNC 2017), pp.1021-1022, Jan. 2017.
[15] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Analyzing Machine Noise for Real Time Maintenance," 2016 8th International Conference on Graphic and Image Processing (ICGIP 2016), Oct. 2016.
[16] Y. Yamato, "Experiments of posture estimation on vehicles using wearable acceleration sensors," The 3rd IEEE International Conference on Big Data Security on Cloud (BigDataSecurity 2017), pp.14-17, May 2017.
[17] Y. Yamato, T. Demizu, H. Noguchi and M. Kataoka, "Automatic GPU Offloading Technology for Open IoT Environment," IEEE Internet of Things Journal, DOI: 10.1109/JIOT.2018.2872545, Sep. 2018.
[18] Y. Yamato, "Study of parallel processing area extraction and data transfer number reduction for automatic GPU offloading of IoT applications," Journal of Intelligent Information Systems, Springer, DOI:10.1007/s10844-019-00575-8, 2019.
[19] Y. Yamato, "Automatic Offloading Method of Loop Statements of Software to FPGA," International Journal of Parallel, Emergent and Distributed Systems, Taylor and Francis, DOI: 10.1080/17445760.2021.1916020, Apr. 2021.
[20] J. Fung and M. Steve, "Computer vision signal processing on graphics processing units," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 5, pp.93-96, 2004.
[21] S. Wienke, P. Springer, C. Terboven and D. an Mey, "OpenACC-first experiences with real-world applications," Euro-Par 2012 Parallel Processing, pp.859-870, 2012.
[22] M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.
[23] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah and P. Petersen, "Compiler support of the workqueuing execution model for Intel SMP architectures," In Fourth European Workshop on OpenMP, Sep. 2002.
[24] Y. Yamato, "Proposal of Automatic Offloading for Function Blocks of Applications," The 8th IIAE International Conference on Industrial Application Engineering 2020 (ICIAE 2020), pp.4-11, Mar. 2020.
[25] TIOBE web site, https://www.tiobe.com/tiobe-index/
[26] Deckard web site, http://github.com/skyhover/Deckard
[27] CloneDigger web site, http://clonedigger.sourceforge.net/
[28] Numpy web site, https://numpy.org/
[29] Cupy web site, https://cupy.dev/
[30] PyCUDA web site, https://developer.nvidia.com/pycuda
[31] JCuda web site, http://www.jcuda.org/
[32] NAS.FT website, https://www.nas.nasa.gov/publications/npb.html
[33] Himeno benchmark web site, http://accc.riken.jp/en/supercom/
[34] Polybench 3mm web site, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/
[35] Y. Tomatsu, T. Hiroyasu, M. Yoshimi and M. Miki, "gPot: Intelligent Compiler for GPGPU using Combinatorial Optimization Techniques," The 7th Joint Symposium between Doshisha University and Chonnam National University, Aug. 2010.
[36] J. Chen, B. Joo, W. Watson III and R. Edwards, "Automatic offloading C++ expression templates to CUDA enabled GPUs," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp.2359-2368, May 2012.
[37] C. Bertolli, S. F. Antao, G. T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z.

Sura, H. Sung, G. Rokos, D. Appelhans and K. O'Brien, "Integrating GPU support for OpenMP offloading directives into Clang," ACM Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM'15), Nov. 2015.

[38] S. Lee, S.J. Min and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'09), 2009.

[39] A. Shitara, T. Nakahama, M. Yamada, T. Kamata, Y. Nishikawa, M. Yoshimi and H. Amano, "Vegeta: An implementation and evaluation of development-support middleware on multiple opencl platform," IEEE Second International Conference on Networking and Computing (ICNC 2011), pp.141-147, 2011.

[40] K. Shirahata, H. Sato and S. Matsuoka, "Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters,"IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp.733-740, Dec. 2010.

[41] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs.," 2014 IEEE 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp.151-162, Aug. 2014.

[42] Cheng Liu, Ho-Cheung Ng and Hayden Kwok-Hay So, "Automatic nested loop acceleration on fpgas using soft CGRA overlay," Second International Workshop on FPGAs for Software Programmers (FSP 2015), 2015.

[43] C. Alias, A. Darte and A. Plesco, "Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA," 2013 Design, Automation and Test in Europe (DATE), pp.575-580, Mar. 2013.

[44] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan and S. Eggers, "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures," IEEE 2008 International Conference on Field Programmable Logic and Applications, pp.173-178, Sep. 2008.