Title	Study and Evaluation of Improved Automatic GPU Offloading Method
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/10.1080/17445760.2021.1941010 Publication Date: yyyy/mm/dd ・出版日：2021 年 6 月 16 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 16th Jun., 2021, available online: https://doi.org/10.1080/17445760.2021.1941010."

ORIGINAL PAPER

Study and Evaluation of Improved Automatic GPU Offloading Method

Yoji Yamato[a]

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

ARTICLE HISTORY
Compiled June 5, 2021

ABSTRACT
With the slowing down of Moore's law, the use of hardware other than CPUs, such as graphics processing units (GPUs) or field-Programmable gate arrays (FPGAs), is increasing. However, when using heterogeneous hardware other than CPUs, barriers to technical skills, such for compute unified device architecture (CUDA) and open computing language (OpenCL), are high. Therefore, I previously proposed environment adaptive software that enables automatic conversion, configuration, and high-performance operation of once written code according to the hardware to be placed. As part of environment adaptive software, I also proposed a method to offload loop statements of applications to GPUs automatically. In this paper, I improved upon this automatic GPU offloading method to expand its applicability to more applications and enhance offloading performance. I implemented the improved method to evaluate its effectiveness for multiple applications.

KEYWORDS
Environment Adaptive Software; GPGPU; Automatic Offloading; Performance; Evolutionary Computation

1. Introduction

Moore's Law will end shortly, and transistor density cannot be expected to double in 1.5 years. Based on this situation, system usages of heterogeneous hardware, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are increasing. Some providers use special servers with powerful GPUs for mining Bitcoin or special servers with FPGAs to accelerate specific signal processing of network function virtualization (NFV). Amazon Web Services (AWS) [1] provides GPUs and FPGAs using cloud technologies (e.g., [2]-[8]), and Microsoft's search engine Bing uses FPGAs [9].

However, to achieve high application performance using heterogeneous hardware for various applications, developers need to program and configure appropriately considering hardware and need to be skilled in technologies such as compute unified device architecture (CUDA) [10], open computing language (OpenCL) [11] and open multiprocessing (OpenMP) [12]. This prevents easy utilization of heterogeneous hardware.

Due to the progress in Internet of Things (IoT) technology (e.g., Industrie 4.0 and so

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

on [13]-[18]), IoT devices are increasing rapidly. There are various application fields of IoT such as manufacturing, distribution, medical, and agriculture. In IoT applications, knowledge of embedded software and assembly are required for detailed control of IoT devices.

Expectations of applications using heterogeneous hardware, such as GPUs and FPGAs, and many IoT devices are increasing; however, the hurdles are currently high for using them. To break down such a barrier, I think it is required that application programmers only need to write 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 [19] caused a paradigm shift in environment adaptation by allowing software written once to run on different CPU architectures. However, no consideration was given to application performance.

I previously proposed environment adaptive software that effectively runs once-written applications by automatically executing code conversion and configurations so that GPUs, FPGAs, and IoT devices can be effectively used on deployment environments [20]. Even if the performance is not so high compared to manual tuning of hihgly skilled engineers, automation is important, I think. As part of environment adaptive software, I also proposed a method to offload loop statements of applications to GPUs [21] and FPGAs [22] automatically. In this paper, I improved upon the GPU offloading method, which I previously proposed using evolutionary computation [20][21], to be applied to more applications with higher performance. I implemented this method and evaluated its effectiveness for several applications.

The rest of this paper is organized as follows. In Section 2, I review current heterogeneous hardware technologies. In Section 3, I present the improved GPU offloading method to resolve the problems with the original. In Section 4, I explain its implementation and discuss its evaluation and the results in Section 5. In Section 6, I mention related work and conclude the paper in Section 7.

2. Current heterogeneous hardware technologies

Java [19] is one example of environment adaptive software. By using a virtual execution environment called Java Virtual Machine, written software can be to run even on machines of different OS without compiling (Write Once, Run Anywhere). However, it was not considered whether the expected application performance could be attained at the porting destination, and much effort was needed for performance tuning and debugging at the porting destination (Write Once, Debug Everywhere). If software uses heterogeneous hardware, tuning becomes more difficult.

CUDA is a major development environment for general purpose GPUs (GPGPUs) that use GPU computational power not only for graphics processing (e.g., [23]). To control heterogeneous hardware such as GPUs, FPGAs, and many core CPUs uniformly, OpenCL specifications and its software development kit (SDK) are widely used. CUDA and OpenCL need not only a 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 on the basis of specified directives. Open accelerators (OpenACC) [24] is a directive-based specification, and the PGI compiler [25] is a 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 a Java lambda expression [26].

CUDA, OpenCL, OpenACC, and others support GPU or FPGA offload processing. Although processing on a GPU or FPGA can be done, sufficient offloading performance is difficult to attain. For example, when users use an automatic parallelization technology such as the Intel compiler [27] for a multicore CPU. The Intel compiler specifies whether to process for loop statements that can be parallelized in parallel, but if we simply specify it, it will often be slower. Naive parallel execution performance with GPUs or FPGAs is not high because of the 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 using the PGI compiler or other compilers.

Therefore, it is difficult for users without GPU or FPGA skills to attain high offloading performance. Moreover, if users use automatic parallelization technologies to attain high offloading performance, it takes a long time of trial and error to determine if each loop statement is parallelized, and there are many applications that cannot be improved.

In my previous studies, I attempted to automate the trial and error of parallel processing areas [20][21]. Loop statements suitable for GPU offloading were extracted appropriately by repeating the performance measurements in the verification environment using evolutionary computation, and as many variables in the nested loop statements as possible were transferred in the upper loop. However, when considering actual use, there were two problems, i.e., the accelerated applications were limited, and performance improvements were insufficient compared to manual acceleration using CUDA.

3. Improvement of automatic GPU offloading method for loop statements

3.1. Processing flow of environment adaptive software

To adapt software to an environment, I previously proposed environment adaptive software, as shown in Figure 1. The environment adaptive software is realized in cooperation with functions including an environment adaptation function, a test-case database (DB), code-pattern DB, facility-resource DB, verification environment, and production environment. The environment adaptation is processed with following steps.

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:

In Steps 1 to 7, the processing flow for executing code conversion, resource-amount adjustment, placement-location adjustment, and in-operation reconfiguration for environment adaptation are carried out. It is possible to skip some of the steps. For example, if we only want to convert code for a GPU, we only need to conduct Steps 1 to 3. We can use only the necessary processing of the environment adaptation function.

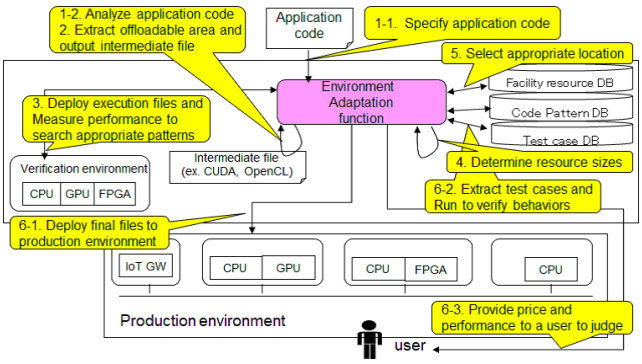


Figure 1. Processing flow of environment adaptive software

3.2. Previous automatic GPU offloading method and its problems

I now explain my previous automatic GPU offloading method. There are two main features for applications offloading. The first one is the loop statements suitable for a GPU are extracted using evolutionary computation [21] of the genetic algorithm (GA) [28], and the variables used in nested loop statements are transferred between the CPU and GPU in the outer-most loop possible [20].

The applications that users want to offload are various. However, typical applications that require a large amount of computation time, such as image analysis for video processing, machine-learning processing for analyzing sensor data, and loop processing, often take a long time. Therefore, our GPU offloading targets are also loop statements.

It is currently possible for compilers to find the restriction that avoid these loop statements to be processed in parallel on a GPU, but it is difficult to determine if these loop statements are suitable for GPU parallel processing. Loops with a high arithmetic intensity, such as a large number of repetitions, is generally suitable; however, it is difficult to predict how much offloading performance can be attained without actual performance measurements using a GPU. Because th memory differs between CPU and GPU, so the performance greatly depends on the timing of data transfer and calculation processing on GPU core. Therefore, there are many cases in which an instruction to offload this loop to a GPU is given manually, and performance measurement is executed by trial and error.

Based on this situation, I previously argued that the GA automatically finds an appropriate loop statement to be offloaded to a GPU [21] (Figure 2). First, parallelizable checks are conducted 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 set for GPU execution and 0 for CPU execution. Then, performance-verification trials are repeated in a verification environment to search for an appropriate offloading area. Since this method searches for faster offload patterns in the verification environment recursively, it cannot be slow from the original pattern in principle. Patterns that can be efficiently accelerated from the enormous amount of parallel processing patterns are searched by holding and recombining better parallel processing patterns in the form of genes after focusing on parallel loop statements.

In addition to the appropriate extraction of loop statements, I previously proposed to transfer as many variables used in the nested loop statements as possible to the upper level loop to reduce the number of transfers between CPU and GPU [20]. When

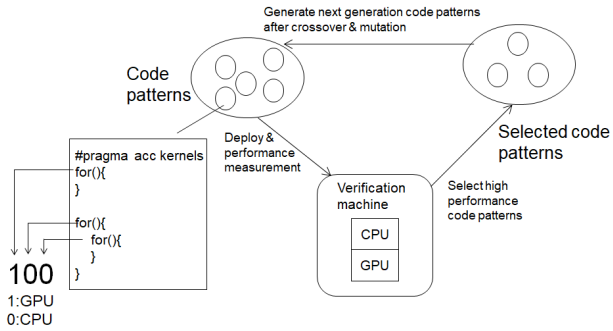


Figure 2. Image of GPU offloading area search

offloading a loop statement to a GPU with a naive method, if CPU-GPU transfer is executed at the lower level of nesting, transfer is done at each lower-level loop, which is not efficient. The cause of long processing time is often nested loops. This shows the effect on offloading performance improvement by reducing the number of CPU-GPU transfers.

Whether variables can be collectively transferred from the CPU to the GPU depends on the overall configuration including other loops, so GPU processing of each loop cannot be independent. Since the offload patterns of loop execution and variables transfer become enormous depending on the loop statements number, GA is used to find some extent offload pattern at a high speed.

Based on these two ideas, I confirmed automatic performance improvements even for medium-scale applications with more than 100 loop statements. However, there were two issues when considering practical use; 1) insufficient performance and 2) insufficient applicable applications.

In terms of performance improvement, there are many applications in which automatic performance improvement using OpenACC is not sufficient compared to manual improvement using CUDA. With CUDA-acceleration methods, CPU-GPU data transfer reduction, appropriate use of multiple memories (shared, constant, texture, local, and global), coreless access, suppression of a branch in Warp of CUDA, high occupancy by Warp simultaneous multi-threading, task parallelization by stream processing, parallelization granularity tuning suitable for the number of threads, and so on are carried out. Of course, changing to parallel processing from serial processing is a major premise. The important tuning point is the reduction of CPU-GPU transfer rather than memory allocation in a GPU due to absolute values of the transfer speed.

There are some applications that cannot be used because many errors occur at compiling before GA of my implementation tools [20]. [20] used only #pragma acc kernels that handle single and tightly nested loops were targeted, and for statements of non-tightly nested loops resulted in errors when #pragma acc kernels were added. As a result, applications with many non-tightly nested loops have not been able to start GA. In the next subsection, I examine the expansion of directives so that instructions can be given to loop statements that were excluded in previous studies due to errors of instructing GPU processing.

3.3. Reduction in number of CPU-GPU transfers and expansion of GPU processing directives

First, to reduce CPU-GPU data transfers more, we match many variable transfer times and reduce transfers that compilers automatically transfer. CPU-GPU transfer occurs when loop statement processing is offloaded to a GPU, but offloading performance can be increased by reducing the number of transfers with time matching and unnecessary transfer preventions.

To reduce the number of CPU-GPU transfers, not only the nesting loops but also the variables that summarize the transfer time to a GPU are transferred in a batch. For example, unless the result of GPU processing can be processed by the CPU and processed again by a GPU, it may be possible to define variables by the CPU used in multiple loop statements that are transferred in batch to the GPU before GPU processing begins and return to the CPU after all GPU processing is complete. Since the reference relationship of variables can be understood during code analysis, when variables defined in multiple files are used separate CPU and GPU processes and not staggered, these variables are specified batch transfer using OpenACC "data copy" and "update". In addition, variables that are transferred in a batch and do not need to be transferred at that time are clearly indicated using the directive of "data present". "Present" is a clause that a GPU already has a variable.

We then consider reducing the number of transfers that compilers may transfer automatically. For example, Figure 3 shows the case of a PGI compiler that is well-known as an OpenACC compiler. If GPU processing is simply specified in the #pragma acc kernels clause without using the OpenACC "data copy" or "present" clause, the variables in the loop are synchronized between the CPU and GPU in each loop. The method explicitly specified "data copy" to mitigate this problem of nesting loop statements [20]. However, even when "data copy" or "present" is specified with OpenACC, variables may be automatically transferred by a compiler judgement. Compilers basically process on the safe side by judging from multiple conditions. Conditions are whether a variable is global or local, where it is initialized, obtained from another function including a loop, only referenced, or updated within a loop. Depending on the compiler specifications, transfers occur even if unnecessary. Therefore, to reduce the number of transfers that degrade offloading performance but are not intended by OpenACC instructions, we create a temporary area, initialize variables in the temporary area, and transfer variables using it for CPU-GPU transfer.

Next, we consider expanding directives to increase the number of applications that can be applied. Specifically, the directive to specify GPU processing is expanded to the "parallel loop" and "parallel loop vector" directives in addition to the "kernels" directive used in previous studies. In the OpenACC standard, "kernels" are for single loops and tightly nested loops, "parallel loop" is for loops that include non-tightly nested loops, and "parallel loop vector" is for loops that cannot be parallelized but vectorized. A tightly nested loop is a simple nested loop. For example, when two loops that increment i and j are nested, the processing using i and j is carried out in the lower loop not in the upper loop. In implementations examples such as PGI compilers, the difference is that "kernels" determine the parallelism by compilers, and "parallel" determines the parallelism by programmers. Simple loops were considered in previous studies, but non-tightly nested loops and loops that cannot be parallelized but vectorized, so the areas of applicable applications were narrow.

Therefore, in this study, we used "kernels" for single and tightly nested loops, specified non-tightly nested loops by "parallel loop", and specified loops that cannot be

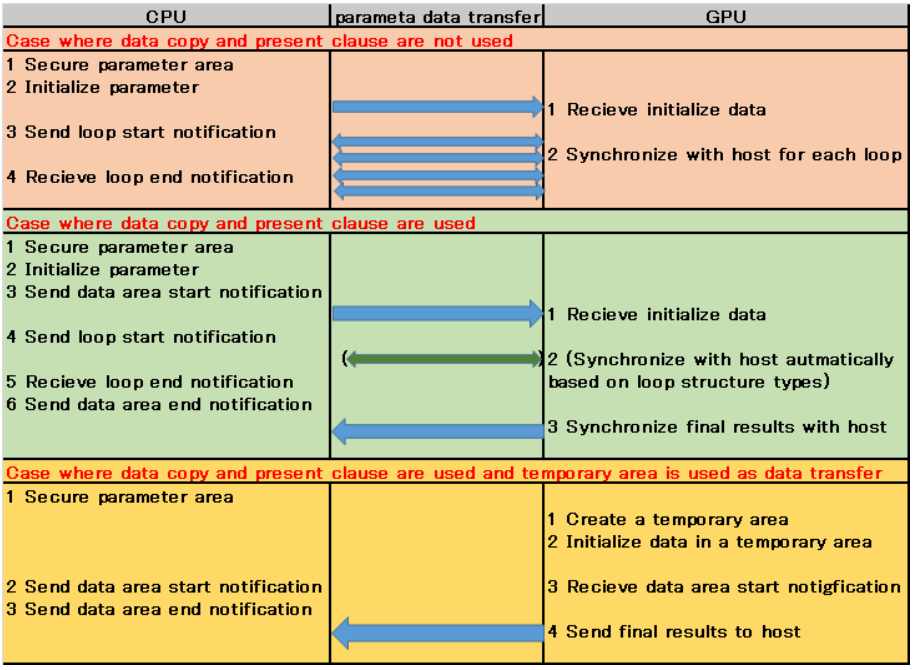


Figure 3. CPU-GPU transfer using temporary transfer area

parallelized but vectorized by "parallel loop vector". Because "parallel loop" has more degrees of freedom than "kernels", there is a possibility that calculation results differences may increase compared to the case of only "kernels" by adding also "parallel". Thus, it is assumed that the offload program is confirmed by conducting a sample test, checking the result difference with only CPU and CPU+GPU, and showing the check results to the user whether to accept or not. Since CPUs and GPUs differ in terms of hardware, there are differences in the number of significant digits, rounding errors, and so on. I think that it is necessary to check the differences in calculation results with only CPU processing even in only "kernels" cases.

From my previous research [20], this research can make more applications faster by reducing transfers and increasing the pattern of target for loop statements.

4. Implementation

The implementation of the improved method is as follows. After code analysis, the method is used to prepare gene individuals that specify GPU processing with "kernels", "parallel loop", or "parallel loop vector" for loop statements, specifies a temporary area to reduce the number of CPU-GPU transfers for each individual, and specifies batch transfer of variables in multiple files. It then compiles individuals' codes and measures application performance in a verification environment. High-performance individuals are selected, and GA processing is carried out with crossover mutation. Through GA processing, faster offloading patterns are searched by trial and error. After the final solution is determined, the method checks the calculation error when using only CPUs and using CPUs and GPUs.

The purpose of this implementation was to confirm the effectiveness of automatic

GPU offloading improvement. The application I used was a C/C++ language application, and GPU processing used PGI compiler 19.4.

The 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 for loop statements by the OpenACC directives of #pragma acc kernels, #pragma acc parallel loop, and #pragma acc parallel loop vector, and executed on the GPU. The PGI compiler can also specify whether to transfer data by specifying the #pragma acc data copyin/copyout/copy or #pragma acc data present directive.

We implemented the method using Perl 5 and Python 2.7. The Perl program controls GA processing, and the Python program parses source codes.

When a C/C++ application is specified, this implementation analyzes C/C++ code and detects loop statements and variables used in loop statements. For parsing, the Python program uses parsing libraries of LLVM/Clang 6 [29] libClang Python binding. We can also use the similarity-detection tool Deckard to understand the functions used.

We need to exclude loop statements in which GPU processing is impossible. The Pgcc of the PGI compiler can determine for each statement such as kernels that can be processed because of tightly nested loop, cannot be parallelized but vectorized or so on. Therefore, for each for statement that makes such a judgement, the implementation attempts inserting #pragma acc kernels, #pragma acc parallel loop, and #pragma acc parallel loop vector directives to determine whether errors occurred during compilation. Three directives of #pragma acc kernels, parallel loop, and parallel loop vector will result in errors, and for statements that cannot be processed by GPU will be excluded from the search target for GPU processing in GA. We also can exclude loops with few loop number by checking gcov or gprof.

If the counted GPU processable loop statements are A, A is the gene length. Gene value 1 corresponds to a GPU processing directive, and 0 corresponds to no directive. Source codes with several loop statements are mapped to genes with length A.

A specified number of individual genes is prepared as the initial setting. Initial genes are created to assign 0 and 1 randomly. The implementation adds GPU processing directives #pragma acc kernels, #pragma acc parallel loop, and #pragma acc parallel loop vector to C/C++ loop statements when corresponding gene values are 1. The reason for not adding parallel directives to tightly nested loop statements is that kernel directives perform better if they execute the same processing with that of the PGI compiler. After gene values are set, parts to be processed by the GPU in source codes are determined.

Next, data-related directives are added; "data copy", "data present", and a temporary area are specified based on the reference relation of variables in the loop statement analyzed by Clang. The following are cases of data transfers are needed. When variables set and defined on the CPU side and variables referenced on the GPU side overlap, variable transfer from the CPU to GPU is required. When variables set on the GPU side are referenced and those set or defined on the CPU side overlap, variable transfer from the GPU to CPU is required. If there is no repetition, such as processing the results processed by the GPU and processing it again by the GPU, data transfers may be possible in batch. Specifically, for variables that can be transferred in a batch before GPU processing and after GPU processing, the implementation specifies #pragma acc data with the same timing for variables defined in multiple files. Even if multiple files cannot be batched, variables that can be batched in nesting loops are batched, as discussed in my previous study [20]. Since many variables are transferred in a batch, there are variables which are already in GPU when GPU processes each loop statement. In this case, the implementation instructs that variable transfers are

not required by specifying #pragma acc data present. In addition, to prevent automatic transfer of compilers, when CPU-GPU transfer is required with #pragma acc data copy and so on, the implementation creates a temporary area on the GPU side (#pragma acc declare create), initializes the data, and synchronizes the data via this temporary area (#pragma acc update).

The C/C++ codes with the added #pragma directives are compiled by the PGI compiler on a verification machine with a GPU. The implementation deploys compiled binary files and measures application performance by using sample test tools. Along with performance measurement, calculation results differences check is also conducted using PCAST function of the PGI compiler. By specifying options of the pgi_compare or acc_compare API, various checks, such as error check according to IEEE 754 specifications, are possible.

After the performances of all individuals are measured, the implementation sets a goodness-of-fit value to each gene on the basis of each performance result. If the calculation result difference is large, the goodness of fit value may be reduced. The implementation selects individuals based on goodness-of-fit values. For selected individuals, the implementation processes evolution operation of crossover, mutation, or copy to create the next-generation of individuals.

For next-generation individuals, directive insertion, compile, performance measurement, setting a goodness-of-fit value, selection, crossover, and mutation are carried out. After completion of GA processing for the specified number of generations, the C/C++ code with directives that corresponds to the gene with the highest performance is the final solution.

A sample test was conducted for the final solution and calculation result difference is checked again using PCAST. Performance and calculation result difference are showed to the user and the user can judge whether to start or not based on performance and calculation result difference are acceptable or not.

5. Evaluation

5.1. Evaluation method

5.1.1. Evaluated applications

I evaluated five applications, NASA Advanced Supercomputing Fourier Transform (NAS.FT), discrete Fourier Transform (DFT), Himeno benchmark, Multi-resolution Adaptive Numerical Environment for Scientific Simulation (MADNESS), and Laplace 2D equation. Many users will also use fast Fourier transform (FFT) and other analyses in an IoT environment. I also evaluated these applications in my previous studies [21][20].

In an IoT environment, FT is often necessary for monitoring such as vibration or sound of vehicle or factory-machine sensors. NAS.FT [30] calculates 3D FFT. When considering an application that transfers data from a device to a cloud via a network, it is assumed that the device side executes primary analysis such as FFT processing to reduce network traffic. I used a sample test of Fourier transformation as a benchmark for tuning, which sample test is equipped in NAS.FT files. Sample test parameters of grid size is 256*256*128 and the number of iterations is 6. I also tested a DFT [31] because FT has many patterns, and it needs to be confirmed whether other patterns can also be improved.

Himeno benchmark [32] 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, I used it to confirm whether it can be accelerated with my improved method. Sample test parameters of Large data set is 512*256*256.

MADNESS is a high-level software environment for solving integral and differential equations in many dimensions based on multi-resolution analysis and separated representations [33]. I used a test tool to calculate 1000*1000 test data as a benchmark for tuning.

Laplace 2D equation [34] solves the Laplace equation for a 2-dimensional data set. I used a test tool to calculate 4096*4096 test data as a benchmark for tuning.

5.1.2. Experimental conditions

Since I evaluated the convergence of acceleration using the GA in previous studies [21][20], I do not illustrate the performance of each application for each generation of the GA in this paper. However, I measured application performance for all generation and individual patterns. The highest performance code pattern when the specified number of generations were attempted was the final solution, and this solution was evaluated by comparing it with CPU processing of all codes.

Parameters and conditions of the GA are as follows.

Gene length: Number of GPU processable loop statements. 65 for NAS.FT, 10 for DFT, 13 for Himeno benchmark, 10 for MADNESS, and 5 for Laplace 2D equation.

Number of individuals M (no more than the gene length): 30 for NAS.FT, 10 for DFT, 10 for Himeno benchmark, 10 for MADNESS, and 5 for Laplace 2D equation.

Number of generations T (no more than the gene length): 20 for NAS.FT, 10 for DFT, 10 for Himeno benchmark, 10 for MADNESS and 5 for Laplace 2D equation.

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), I prevent the narrowing of the search range due to too high of the goodness of fit of 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.

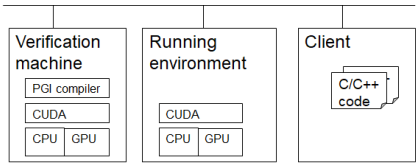
Selection algorithm: Roulette selection and Elite selection. Elite selection means that one gene with maximum goodness of fit must be reserved for the next generation without crossover or mutation.

Crossover rate Pc: 0.9

Mutation rate Pm: 0.05

5.1.3. Experiment environment

I used physical machines with NVIDIA Quadro P4000 for evaluations of the improved method. The CUDA core number of NVIDIA Quadro P4000 was 1792. I used PGI compiler community edition v19.4 and CUDA toolkit v10.1. Figure 4 shows the experimental environment and specifications. A client note PC specifies C/C++ application codes, the codes are then tuned with trial and error on a verification machine, and final codes are deployed in a running environment for users after GA tuning.



Name	Hardware	CPU	RAM	GPU	OS	CUDA toolkit	PGI Compiler
Verification Machine	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	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	
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 7 Professional		

Figure 4. Experimental environment

5.2. Performance results

All five applications improved in performance. Due to the characteristics of the GA, it does not converge at the same number and same performance every time.

Figure 5 shows an example of performance change in my previous study [20]. It shows maximum performance change of NAS.FT in each generation with GA-generation transitions (the vertical axis shows how many times faster GPU offloading was than using only a CPU). Application performance improved and GPU offloading was about 5 times faster (processing time shortened from 31.3 to 5.8 seconds). Because the same gene patterns often occur in the GA process, the search process completed within 7 hours.

Based on the previous results, Figure 6 shows the measurement results of how much application performance improved with the improved method. The figure also shows how many times the final code was compared to the CPU processing of all codes.

From Figure 5, the performance of NAS.FT only improved 5.4 fold in my previous studies [20], but improved 10.0 fold with the improved method. The performance of DFT improved from 5.1 to 23.5 fold, that of Himeno benchmark improved from 4.8 to 15.4 fold, that for MADNESS improved from 12.8 to 16.3 fold, and that of Laplace 2D equation improved from 13.6 to 15.8 fold.

Regarding the Fourier transform, in addition to the previous and this research, I compared the case of using the manually created CUDA library and the case of naive use of OpenACC. When using the Fourier Transform library cuFFT [35] which was created by NVIDIA, the improvement was 730 fold. In the case of naive use of OpenACC, in the example of simply adding #pragma acc kernels to 58 for loop statements, the improvement was 2.1 fold. When the appropriate for loop statements were selected in my method, #pragma directives were inserted to the 27 for loop statements. The use of cuFFT was faster than my method, but the CUDA library is manufactured by NVIDIA highly skilled engineers with special algorithm suitable for NVIDIA GPU. The difference of my method is the automatic speedup with no technical engineer efforts. By searching for appropriate offload patterns automatically, this research also improved more than naive use of OpenACC.

In the experiment, there is only one type of test data, but the absolute value of the performance improvement changes depending on the data type. For example, there are four types of prepared Himeno benchmark sample test data: Small (128*64*64), Medium (256*128*128), Large (512*256*256) and Extra Large (1024*512*512). The result with Large data used in the experiment had 15 fold the performance, but the Small data had a different degree of improvement, such as staying several times im-

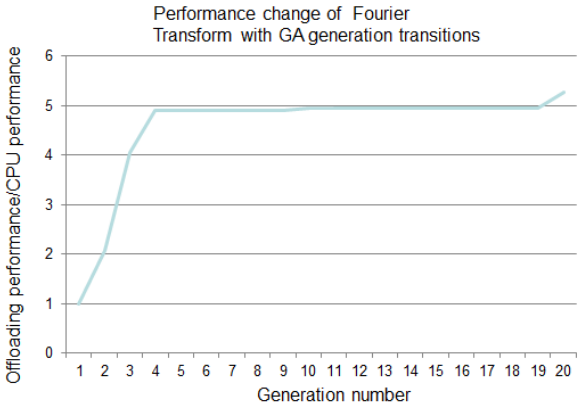


Figure 5. Reference graph: performance change of NAS.FT with GA generation [20]

	Performance improvement of previous papers [20, 21]	Performance improvement of this paper proposals
NAS.FT	5.4	10
DFT	5.1	23.5
Himeno benchmark	4.8	15.4
MADNESS	12.8	16.3
Laplace 2D	13.6	15.8

Figure 6. Comparison of performance improvement between my previous study and this study

provement. Basically, the larger the data size, the higher the improvement in GPU offloading. However, even if the absolute value of improvement differs depending on the data type, the fact that it has been improved compared to only CPU processing and that this research has been improved from the previous research remains the same.

5.3. Discussion

In my previous study [20], I confirmed a 3 to 5-fold improvement in performance of large applications such as Darknet, which has more than 100 loop statements. However, manual speedup with CUDA often resulted in a 10-fold increase in performance. By reducing as many CPU-GPU transfers as possible, the automatic offloading based on OpenACC resulted in a 10-fold improvement in performances for large applications.

In my previous study [20] regarding improvable applications, we targeted simple loops of single and tightly nested loops and carried out GPU parallelization processing using the "kernels" directives of OpenACC. However, there was a problem in that there were many applications that caused errors during GPU compilation. "Parallel" directives can be used for non-tightly nested loops, and the "parallel loop vector" directives can be used for codes that cannot be parallelized but can be vectorized. Thus, many applications can be applied to attempt to improve application performance.

Next, we discuss cost performance. GPU boards, such as NVIDIA Quadro, cost about 2,000 USD. Therefore, hardware with a GPU costs about twice as much as

hardware with only a CPU. In data centers, hardware, development, and verification costs of a system, such as a cloud system, are generally about 1/3 the total cost; electricity and operation/maintenance cost is more than 1/3 and other expenses, such as service orders, are roughly the other 1/3. In AWS, a GPU instance with one GPU costs about 650 USD/month, which is the same as hosting a general dedicated server. Therefore, I think improving performance 10 fold in various type loops of applications that take much time will have a sufficiently positive cost effect even though the hardware cost is doubled.

We now discuss the time to start production services. Convergence time of a GA took several hours in this study as well as in my previous studies. The performance measurement from compilation of one trial is about 3 minutes, and it takes time to find a solution according to the number of individuals and generations. However, the measurement of the same gene patterns as before is omitted, so it takes only several hours in many cases. When we provide production services, we will provide the first day for free and try to improve application performance in the verification environment during the first day, and from the second day we will provide the production service using a GPU. Therefore, the convergence time is considered acceptable.

To search for the optimum offloading pattern within a shorter time, we can process each individual performance measurement on multiple machines in parallel.

In these evaluation experiments, I set a high Pc to search for a wide area and searched for a certain level of solutions relatively fast but parameters may be tuned more.

The target type of loop statements was also expanded from only "kernels" but also "parallel loop" and "parallel loop vector"; therefore, it was necessary to confirm what type of application can be automatically accelerated within existing applications.

Of course, there are applications that cannot be accelerated using the GPU. NAS.BT [30] which calculates Block Tri-diagonal solver has a lot of CPU-GPU data transfer, so we could not speed it up. Since there was no data transfer with manycore CPUs, the speed of NAS.BT could be increased to more than 5 times by the similar GA method using AMD Ryzen 32 core CPU. Therefore, I also study selecting an appropriate migration destination when there are multiple migration destination candidates such as GPU, FPGA, manycore CPU, but multiple migration destinations are out of scope in this paper.

6. Related work

Wuhib et al. studied resource management and effective allocation [36] on the OpenStack cloud. My method is a network wide resource management and effective allocation method including the cloud, but it focuses on appropriate offloading on heterogeneous hardware servers. I previously proposed methods [**?**] for selecting appropriate servers from heterogeneous hardware servers, but the improved method in this paper can offload appropriate loop statements of applications automatically, which is novel.

Some studies focused on offloading to GPUs [37][38][39]. Chen et al. [37] used metaprogramming and just in time (JIT) compilation for GPU offloading of C++ expression templates, Bertolli et al. [38] and Lee et al. [39] investigated offloading to GPUs using OpenMP. There have been few studies on automatically converting existing code to a GPU without manually inserting new directives or a new development model which the author targets.

In [40][41], GPU offloading areas were searched, and the GA was also used to search

for them automatically in [41]. However, its target was specific applications that have many papers to accelerate by GPU, and a huge number of tunings is needed such as calculations for 20 individuals and 200 generations. My method aims to complete the GA process in a short time because we want to start production services quickly for general CPU applications by focusing on parallelizable loop statements for GA calculations.

I evaluated my improved method by using an OpenACC PGI compiler for C/C++ applications. In addition to the C/C++ language, Java is often used for open source software (OSS) applications. From Java 8, parallel processing can be specified by a lambda expression. IBM provides a JIT compiler that offloads processing with lambda expressions to a GPU [26]. In the case of Java, we can extract an appropriate offloading area by using this JIT compiler and the GA that checks whether each loop statement requires parallel processing with a lambda expression.

Generally, CUDA and OpenCL control intra-node parallel processing, and message passage interface (MPI) controls inter-node or multi-node parallel processing. However, MPI also requires high technical skills for parallel processing. Thus, MPI concealment technology has been developed that virtualizes devices of outer nodes as local devices and enables devices of outer nodes to be controlled by only OpenCL [42]. When we select multiple nodes for offloading destinations, I 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. [43] 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. [44] is also related to task scheduling when a CPU and GPU are integrated chips of the same die. Referring to their papers, I study how to deploy functions on appropriate locations and resource amounts to avoid bottlenecks of CPUs or GPUs.

There have been many reports on using GPGPUs to achieve high offloading performance for high-arithmetic-intensity applications. Automatic offloading to automatic parallelization compilers, such as the Intel compiler for multicore CPUs, is common. However, there has been no study that combines automatic tuning of an appropriate offloading area by evolutional calculation and a reduction in the number of data transfers between CPUs and GPUs. My method can offload large applications to GPUs.

7. Conclusion

I improved upon my previous automatic GPU offloading method for loop statements of applications, which is an elemental component of environment adaptive software. Environment adaptive software adapts applications to environments to effectively use hardware such as GPUs.

My previous automatic GPU offloading method extracts appropriate loop statements to be offloaded to a GPU by using an evolutionary calculation method and transfers as many variables in the nested loop statements as possible in the upper loop. To increase offloading performance and expand the number of applications, I improved upon this method. To reduce CPU-GPU transfers for higher offloading performance, the range of batch CPU-GPU transfers is extended to multiple file variables. Also, variables that are already in the GPU after batch transfer are instructed that

transfer is not necessary using OpenACC #pragma acc data present. Variables are also transferred via a temporary area. Furthermore, GPU processing of a single loop and tightly nested loop is instructed using OpenACC #pragma acc kernels. For loops that can be GPU processed even in non-tightly nested loops, I use OpenACC #pragma acc parallel. This expands the applicable loop statements.

I evaluated the improved method for several applications and showed that they perform 10 times better. In the future, I will evaluate it for more applications and study not only loop statements but also function-block offloading such as Fourier Transform block.

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] AWS EC2 website, https://aws.amazon.com/ec2/

[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, 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.

[4] 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.

[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, "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.

[7] Y. Yamato, "Cloud Storage Application Area of HDD-SSD Hybrid Storage, Distributed Storage and HDD Storage," IEEJ Transactions on Electrical and Electronic Engineering, Vol.11, pp.674-675, 2016.

[8] 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, 2015.

[9] 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.
[10] J. Sanders, E. Kandrot, "CUDA by example : an introduction to general-purpose GPU programming," Addison-Wesley, ISBN-0131387685, 2011.
[11] 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.
[12] T. Sterling, M. Anderson and M. Brodowicz, "High performance computing : modern systems and practices," Cambridge, MA : Morgan Kaufmann, ISBN 9780124202153, 2018.
[13] M. Hermann, T. Pentek and B. Otto, "Design Principles for Industrie 4.0 Scenarios," Rechnische Universitat Dortmund. 2015.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] J. Gosling, B. Joy and G. Steele, "The Java language specification, third edition," Addison-Wesley, 2005. ISBN 0-321-24678-0.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.
[26] K. Ishizaki, "Transparent GPU exploitation for Java," The Fourth International Symposium on Computing and Networking (CANDAR 2016), Nov. 2016.
[27] 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.
[28] J. H. Holland, "Genetic algorithms," Scientific american, Vol.267, No.1, pp.66-73, 1992.
[29] Clang website, http://llvm.org/
[30] NAS website, https://www.nas.nasa.gov/publications/npb.html
[31] DFT website, http://programming.blogo.jp/c/fourier_transform

[32] Himeno benchmark web site, http://accc.riken.jp/en/supercom/
[33] MADNESS website, https://github.com/m-a-d-n-e-s-s/madness
[34] Laplace equation website, https://github.com/parallel-forall/cudacasts/tree/master/ep3-first-openacc-program
[35] cuFFT web site, https://docs.nvidia.com/cuda/cufft/index.html
[36] F. Wuhib, R. Stadler, and H. Lindgren, "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 virtualiztion management, pp.309-315, Oct. 2012.
[37] 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.
[38] 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.
[39] 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.
[40] Y. Tanaka, M. Yoshimi. M. Miki and T. Hiroyasu, "Evaluation of Optimization Method for Fortran Codes with GPU Automatic Parallelization Compiler," IPSJ SIG Technical Report, 2011(9), pp.1-6, 2011.
[41] 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.
[42] 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.
[43] 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.
[44] 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.