Journal of Network and Systems Management (2024) 32:11
https://doi.org/10.1007/s10922-023-09789-2

Proposal and Evaluation of GPU Offloading Parts Reconfiguration During Applications Operations for Environment Adaptation

Yoji Yamato[1]

Received: 1 June 2023 / Revised: 18 October 2023 / Accepted: 30 October 2023 /
Published online: 28 November 2023

Abstract
In recent years, not only CPUs with few cores but also heterogeneous hardware such as GPUs, FPGAs, and multi-core CPUs are increasingly used in many applications. However, to fully utilize these, users need to have technical knowledge that covers hardware such as CUDA. To overcome this high technical barrier, we have proposed environment-adaptive software that enables high-performance operation by automatically converting application code written for normal CPUs by engineers in accordance with the deployed environment and by setting appropriate amounts of resources. So far, we have also verified the elemental technologies that automatically offload to GPU and FPGA before the start of operation. Until now, we only considered conversions and settings before the start of operation. In this paper, we verify that the logic is reconfigured in accordance with the usage characteristics during operation. Especially for GPU logic, there is no example of reconfiguration during operation, so the proposed method can be expected to have a great impact on clouds or similar businesses. We propose a GPU reconfiguration method during operation and find that the application running on the GPU is reconfigured to other offload loops or other offload applications in accordance with the current usage trends. Through a reconfiguration experiment, performance improvement and break time are measured, and the effectiveness of the method is demonstrated.

Keywords Environment adaptive software · Automatic offloading · GPGPU · Reconfiguration during applications operations · Cost performance

✉ Yoji Yamato
yoji.yamato@ntt.com

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

1 Introduction

Moore's law, which states that the degree of semiconductor integration of central processing units (CPUs) will double every 1.5 years, is expected to finally slow down. Under such circumstances, not only CPUs but also devices such as FPGAs (Field Programmable Gate Arrays) and GPUs (Graphics Processing Units) are being increasingly used. For example, Microsoft is making efforts to improve the search efficiency of Bing using FPGA [1], and Amazon provides FPGA and GPU as instances [2] using cloud technologies (for example, [3, 4]). In addition, as heterogeneous devices, the use of small devices such as IoT (Internet of Things) devices (for example, [6–12]) is also increasing.

However, to properly utilize devices other than CPUs with a small number of cores in the system, settings and program coding need to be made while considering device characteristics. For example, programmers require knowledge of OpenMP (Open Multi-Processing) [13], OpenCL (Open Computing Language) [14], and CUDA (Compute Unified Device Architecture) [15]. Therefore, for most programmers, the skill barrier is high.

The number of systems that utilize devices such as GPUs, FPGAs, and multi-core CPUs other than few core CPUs is expected to increase in the future, but there are high technical barriers to making the best use of them. Therefore, to lower these barriers and make it possible to fully use devices other than few core CPUs, the software in which the programmer describes the processing logic is adapted to the environment (FPGA, GPU, multi-core CPU, or so on) of the deployment destination. Therefore, a platform is needed that can be adaptively converted, configured, and operated in accordance with the environment.

Java [16] appeared in 1995, making it possible to operate the code once written on a device equipped with a CPU of another vendor and causing a paradigm shift regarding environmental adaptation at the software development site. However, the performance at the migration destination was not always appropriate. Therefore, we proposed environment-adaptive software so that the code once written can be used with the GPU, FPGA, multi-core CPU, or so on in the environment of the deployment destination. The environment-adaptive software automatically performs conversion, resource setting, and so on to operate the application with high performance. At the same time, we also proposed and evaluated methods of automatically offloading loop statements and function blocks of C language program code to GPU and FPGA as elements of environment-adaptive software [17, 19–23].

However, our verification has so far only been to perform adaptation processing such as conversion before the start of operation, and we have not considered re-adapting in accordance with the actual usage characteristics in the production environment during operation. We will consider an example of image processing. Before the start of operation, the logic was built so that GPU processing would be performed on the premise that there would be a lot of classification processing. However, when analyzing the actual number of requests three months after the start of operation, the object detection process may be longer. In this case, it is better to change the logic that performs GPU processing.

This paper focuses on the reconfiguration of GPU logic, while reconfiguring the applications in accordance with usage characteristics after the start of operation. There is no example of reconfiguring GPU logic in accordance with usage characteristics during operation using GPU for application acceleration in the production cloud (e.g., AWS GPU instance). GPU reconfiguration will have a big impact, we think. First, the existing application is automatically offloaded to the GPU, and the operation is started. We propose a reconfiguration method that analyzes the usage characteristics, suggests to the user that the GPU logic be reconfigured to another offload, and changes it with a shorter break time. The effectiveness of the proposed method is evaluated by the degree of performance improvement and the break time through GPU reconfiguration during operation. This paper is an improved version of a paper at the international conference ICIET and adds details to the proposal, implementation, evaluation, and discussion.

The structure of this paper is as follows. Section 2 outlines the existing technology and explains our previously proposed environment-adaptive software. In Sect. 3, we propose a specific method for reconfiguring GPU logic in accordance with the usage characteristics during operation. Section 4 explains the implementation and implementation points of the proposed method. In Sect. 5, we evaluate the proposed method through GPU offload of existing applications and GPU reconfiguration during operation. We compare our study with related studies in Sect. 6 and summarize it in Sect. 7.

2 Existing Technologies

2.1 Market Technologies

Java is one example of environment-adaptive software. Java uses Java Virtual Machine, which is a virtual execution environment, to run Java code written once on CPU machines of different vendors and different OSes without recompiling (Write Once, Run Anywhere). However, it was not known how good performance would be at the migration destination, and there was a big problem in the operation of debugging and tuning related to performance at the migration destination (Write Once, Debug Everywhere).

CUDA is widespread as an environment for performing GPGPU (General Purpose GPU) (for example, [24]) that uses the parallel computing power of GPU for non-image processing. CUDA is NVIDIA's environment for GPGPU, but OpenCL has come out as a specification to handle heterogeneous devices such as FPGA, multi-core CPU, and GPU in the same way, and its development environment [25] has also come out. CUDA and OpenCL are forms that extend the C language, but the difficulty of the program is high. (For example, it is difficult to explicitly describe the copy and release of memory data between the kernel such as FPGA and the host of the CPU.)

To use a heterogeneous device more easily than CUDA or OpenCL, there are technologies that specify the place to perform parallel processing etc. on the basis of the directive (instruction line), and the compilers output executable files on the

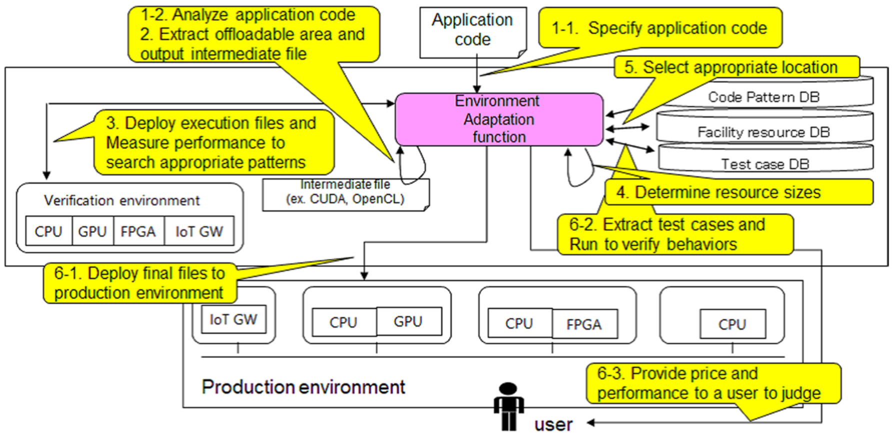

1-2. Analyze application code
2. Extract offloadable area and output intermediate file
Application code
1-1. Specify application code
5. Select appropriate location
Code Pattern DB
Environment Adaptation function
3. Deploy execution files and Measure performance to search appropriate patterns
Facility resource DB
Test case DB
Intermediate file (ex. CUDA, OpenCL)
4. Determine resource sizes
Verification environment
CPU
GPU
FPGA
IoT GW
6-2. Extract test cases and Run to verify behaviors
6-1. Deploy final files to production environment
IoT GW
CPU
GPU
CPU
FPGA
CPU
Production environment
6-3. Provide price and performance to a user to judge
user

Fig. 1 Flow of environment-adaptive software

GPUs, multi-core CPUs, or so on in accordance with the directive. Directive specification examples are OpenACC [26] and OpenMP, and compilers of them are PGI (Portland Group, Inc.) compiler [27] and gcc.

By using technologies such as CUDA, OpenCL, OpenACC, and OpenMP, it is possible to offload to FPGA, GPU, and multi-core CPU. However, even if the device processing itself can be performed, there is a problem in speeding up. For example, there is an Intel compiler [28] that has an automatic parallelization function for multi-core CPUs. At the time of automatic parallelization, these parallelize by extracting the part that can be processed in parallel in the loop statement in the code. However, there are many cases where the performance is not good even if the loop statements that can be parallelized are simply parallelized due to the effect of memory data processing. When speeding up with FPGA or GPU, OpenCL or CUDA engineers repeat tuning, or use PGI compiler to search for an appropriate parallel processing area in a trial-and-error fashion.

For this reason, programmers with poor technical skills have difficulty accelerating applications by utilizing FPGAs, GPUs, and multi-core CPUs, and even when using automatic parallelization technology, a parallel processing area can be searched for in a trial-and-error fashion, but this needs time and effort. To automate trial-and-error searches for parallel processing areas, we propose GPU automation using evolutionary computation.

2.2 Flow of Environment Adaptation

We propose the processing flow in Fig. 1 to enable software to adapt to the environment. The environment-adaptive software operates by coordinating the functions of the verification environment, production environment, test case DB (database), code pattern DB, and facility resource DB, centering on the environment adaptation function.

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: Verify execution-file placement and operation
Step 7: Reconfigure in-operation

Here, Steps 1-6 convert code, set the resource amount, set the placement location, and verify operation, which are actions required before the start of operation. In contrast, Step 7 analyzes the actual usage data after the start of operation and reconfigures applications when it is better to change. The contents to be reconfigured are the same as before the start of operation, such as code conversion, resource-amount setting, and placement-location setting.

This paragraph summarizes tasks. Manual coding is the mainstream method to accelerate applications with heterogeneous devices. We have proposed environment-adaptive software and GPU and FPGA automatic offload methods, but these are only verified before the start of operation, and reconfiguration after the start of operation is not considered. Therefore, this paper focuses on the reconfiguration during operation, especially the reconfiguration of the GPU offload logic according to the actual usage. There are many contents to be reconfigured, such as GPU offload logic, resource amount, and placement location, but there is no example of reconfiguring GPU logic in accordance with usage data in a production cloud such as Amazon Web Services. Therefore, GPU reconfiguration is targeted because its difficulty is high and its effect will be large.

3 GPU Reconfiguration During Operation

We have proposed automatic GPU [23], FPGA offload of loop statements, automatic adjustment of resource amount, automatic arrangement, and so on to embody the concept of environment-adaptive software. On the basis of these elemental technology studies, 3.1 reviews the automatic GPU offload of the loop statement performed before the start of operation, and 3.2 studies the basic policy for reconfiguration to an appropriate logic in accordance with the usage characteristics during operation. In 3.3, we propose a specific method of GPU reconfiguration during operation.

3.1 Review of Automatic GPU Offload Before Operation Start

We review the automatic GPU offload method verified in our previous papers.

There are two main features for automation. [17] proposed extracting a loop statement suitable for GPU by using the genetic algorithm (GA) [29], and [23] proposed transferring the variables used in the nested loop statement on the upper loop as much as possible to suppress GPU-CPU data transfer.

There are various applications that we want to offload, but in applications with a large amount of calculation such as image analysis for video processing and machine learning for analyzing sensor data, iterative processing by loop statements takes a long time. It is said that 99% of time is consumed in loop statements processing. Therefore, the target is to speed up by automatically offloading the loop statement to the GPU.

As a basic task, the compiler can find the limitation that this loop statement cannot be processed in parallel on the GPU, but it is difficult to determine whether this loop statement is suitable for parallel processing on the GPU. A loop with a high amount of calculation such as a large number of loops is generally said to be more suitable, but it is very difficult to predict that the performance will be improved by offloading to GPU without actually measuring performances. Therefore, the usual solution was to manually instruct a certain loop to be offloaded to the GPU and take measurements in trial-and-error fashion to improve performances.

On the basis of that, [17] proposes that GA automatically finds an appropriate loop statement to be offloaded to the GPU. From a general-purpose program that is not supposed to be parallelized, the compiler function first checks the parallelizable loop statement. Next, for the parallelizable loop statement group, a value of 1 is set for GPU execution and 0 is set for CPU execution, and performance verification trials are repeated in the verification environment to search for an appropriate offload part. Processing specifications such as calculations on the GPU are specified by the OpenACC directives, and the execution is performed by the PGI compiler, which is an OpenACC compiler. Patterns that can be efficiently accelerated from the enormous amount of parallel processing patterns are searched for by holding and recombining better parallel processing patterns in the form of genes after focusing on parallel loop statements (Fig. 2).

The method in [23] considers the transfer of variables used in the nested loop statement to reduce the transfer between CPU and GPU. When offloading a loop statement to the GPU, if CPU-GPU transfer is performed at the lower level of the nest, transfer is performed at each lower loop, but this is not efficient. Therefore, for variables that do not have any problem even if CPU-GPU transfer is performed at the upper level, transferring them collectively at the upper level is proposed. Loops that take a long time to process and are large in number are often nested, so this is an idea that is effective in speeding up by reducing the number of transfers.

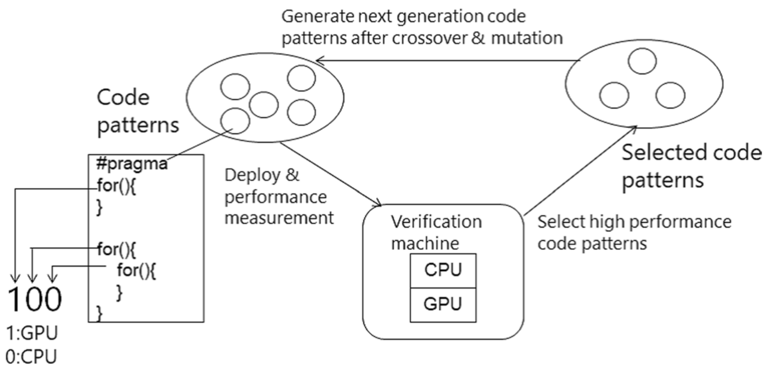

Generate next generation code patterns after crossover & mutation
Code patterns
#pragma
for(){
}
for(){
for(){
}
}
100
1:GPU
0:CPU
Deploy & performance measurement
Verification machine
CPU
GPU
Selected code patterns
Select high performance code patterns

Fig. 2 Automatic GPU offload method for loop statements

On the basis of these two ideas, we have found that the automatic speedup is several times higher even in medium sized applications with more than 100 loop statements. More practical improvements also reported in [23] include further reducing CPU-GPU transfer and expanding loop type that can specify GPU processing.

In the case of GPU, combination patterns of loop statements are performed by GA, and measurements on a scale of 1000 times are performed to search for the optimum pattern. In the case of FPGA, it takes more than 6 h to compile once, so in loop statement offload, we focus on loop statements with high arithmetic intensity, number of loops, and high resource efficiency; create offload patterns; and measure them to search for high-speed patterns.

3.2 Basic Policy for GPU Reconfiguration During Operation

By using the method in 3.1, the loop statements suitable for the GPU can be automatically offloaded to the GPU in the application specified by the user. After offloading to the production environment used by the user, the actual performance and price in the production environment are determined, and the user starts using the application. However, the performance optimization test case (item for performance measurement when comparing performance with multiple offload patterns) used for offload in 3.1 uses the assumed usage data specified by the user because the operation is not started. Therefore, the data may possibly be significantly different from the data actually used during operation.

Therefore, in 3.2, we study whether the GPU logic should be reconfigured with less influence on the user if the usage pattern after the start of operation is different from the initial assumption and the performance is improved by offloading another logic to the GPU. Reconfiguration may change to different loop statement offload in the same application, or it may change to a different application offload.

GPU offload logic reconfiguration requires changing the executed OpenACC, but there are multiple ways to do this. First, there is a method of stopping current OpenACC and starting new OpenACC on the running machine. OpenACC is stopped and started in a short time, and the break time is about several seconds. Next, there is a method of newly building the machine itself that executes new OpenACC, then switching the routing to the new machine when all the current OpenACC processing is completed, and then stopping the current running machine. Routing switching is short, and there is almost no break time. Depending on the degree of user influence, the GPU logic reconfiguration method may be selected, but in either case, there will be a slight break time, and changes to another logic will require an operation confirmation test. Therefore, we think reconfiguration should not be done frequently. We will set restrictions such as proposing only when the effect is above the threshold of improvement.

The reconfiguration process begins with an analysis of request trends over a long-term period, such as one month. Request trends are analyzed to see if there is a higher processing load than the currently offloaded application. Next, for applications with high processing load, GPU offload optimization trials are performed in the verification environment using data that is actually used for the production environment instead of the initial assumed usage data. Then, it is judged whether the new offload pattern found by the verification has a sufficiently higher improvement effect than the current offload pattern, depending on if it is above or below the threshold. If the effect exceeds the threshold, reconfiguration is proposed to the user. After the user approves, the production environment is reconfigured, but the user influence is suppressed as much as possible.

3.3 Method Proposal of GPU Reconfiguration During Operation

On the basis of the basic policy in 3.2, 3.3 proposes a specific reconfiguration method. The reconfiguration method consists of 6 steps, and each step is explained in detail. Step 1 is particularly complicated, so a supplementary explanation will be given at the end.

1. Analyze the production request data history for a certain period (long term), identify multiple applications with high processing time load, and acquire representative data when using those applications.

1-1. Calculate the actual processing time and the total number of uses from each application usage history for a certain period.

However, for application that is GPU offloaded, the processing time if not offloaded is calculated. From the test history of the assumed usage data before the operation, the improvement coefficient is calculated as (actual processing time when only CPU processing is performed)/(actual processing time when GPU is used). Next, the sum of the values obtained by multiplying the actual processing time by the improvement coefficient is used as the total processing time for comparison.

1-2. Compare the total actual processing time for all applications.

1-3. Sort by the total actual processing time, and identify multiple applications with high processing time load.

1-4. Acquire request data for a certain period (short term) of high load applications, arrange the data size for each fixed size, and create a frequency distribution.

1-5. Select one of the actual request data corresponding to the Mode class of the data size frequency distribution as the representative data.

2. Offload patterns are extracted through verification environment measurement to speed up test cases of production representative data in multiple high load applications.

2-1. In high load applications, count the number of for statements by parsing, and use the GPU compiler function to find and remove for statements that cannot be processed by the GPU.

2-2. Create a certain number of gene patterns with the number of remaining for statements as the gene length, 1 means GPU computation, 0 means CPU execution, and corresponding OpenACC directives are added. Directives are both GPU computations such as #pragma acc kernels and data processing such as #pragma acc data copy.

2-3. Compile the OpenACC file corresponding to the gene pattern in the verification environment machine, measure the performance in the test case of the production representative data, and set the higher goodness of fit for faster patterns.

2-4. Depending on the goodness of fit, GA processing such as crossover is performed, next-generation gene patterns are created, GA processing is repeated for a certain number of generations, and the final highest performance pattern is determined as a solution.

3. The processing time with production representative data of the current offload pattern and the extracted new offload patterns is measured, and the performance improvement effect is obtained on the basis of the frequency of production use.

3-1. Calculated with current offload pattern (reduction of actual processing time in verification environment)*(frequency of use in production environment).

3-2. Calculated with new offload patterns (reduction of actual processing time in verification environment)*(frequency of use in production environment).

4. The reconfiguration proposal is judged on the basis of whether the performance improvement effect of the new offload pattern is higher than that of the current offload pattern.

4-1. Calculate a high load application (3-2)/(3-1), check if it is above the threshold, then propose a reconfiguration if it is above, and do nothing if it is below.

5. Propose GPU reconfiguration to the contracted user and obtain an OK/not OK response.

6. Start new OpenACC in a production environment and reconfigure. There are two methods when performing reconfiguration.

6-1. Stop the old OpenACC and start the new OpenACC.

6-1′. A new machine that processes the new OpenACC is built, and the routing is changed so that the new processing is sent to the new machine.

In Step 1, the high load applications are selected, but for the application that is GPU offloaded, the case where it is not offloaded is calculated by multiplying the improvement coefficient, and the application is compared with other applications for CPU processing. Also, when selecting representative data, the Mode class of the data size is used because the average data size may differ significantly from the actual data.

4 Implementation

4.1 Tools to Use

The implementation for evaluating the effectiveness of the proposed method will be explained. To evaluate the effectiveness of GPU reconfiguration, the target applications are C/C++ language applications, and the GPU is NVIDIA GeForce RTX 2080 Ti. The machine to compile is Iiyama LEVEL-F039-LCRT2W-XYVI (CPU: AMD Ryzen Threadripper 2990WX, RAM: 64 GB).

GPU control uses PGI compiler 19.10 and CUDA toolkit 10.1. The PGI compiler is a compiler for C/C++/Fortran that interprets OpenACC. By specifying loop statements such as for statements with OpenACC directives #pragma acc kernels, #pragma acc parallel loop or so on, binary codes for GPU are generated and executed to enable GPU offload. In addition, directives such as #pragma acc data copyin/copyout/copy and #pragma acc data present can be used to explicitly instruct data transfer or state that transfer is not required.

For parsing the C/C++ language, LLVM/Clang 6.0 parsing library (libClang's Python binding) [30] is used.

With GPU offload, for statements that cannot be processed by GPU are found by the function of the PGI compiler and removed from GA. Here, loops with a small number of loops (for example, 1,000 or less) may be excluded because the effect is likely to be small, and the gene length may be shortened. The profiler gcov can be used to analyze the number of loops.

Implementation is done in Perl 5 and Python 3, and the following processing is performed. Perl focuses on processing of offload pattern performance measurement, and Python focuses on other processing such as parsing.

4.2 Each Step Implementation Point

Before the start of operation, the GPU is offloaded with the same operation as the previous implementation tool [23]. Regarding the newly added reconfiguration during operation, as the methods described in 3.3, six steps of reconfiguration are implemented to process in the following order: 1) analyze load, 2) extract new offload pattern, 3) calculate degree of performance improvement, 4) judge reconfiguration proposal, 5) approve reconfiguration, and 6) reconfigure production.

A supplementary explanation will be given regarding the points for implementation.

First, in the load analysis of Step 1, the request data for a certain period (long term) is analyzed to determine the applications with high load, the actual request data for the fixed period (short term) of those applications are selected as the representative data, and these are used when extracting the new offload patterns. Here, the number of high load applications and the fixed period can be set arbitrarily by the business operator. However, a long span of one month or more is assumed for a long term and a short span such as 12 h is assumed for a short term. In the request data analysis, the actual processing time and the number of times the application is used are totaled, and it is acquired by the Linux Time command. Since the Time command shows the actual processing time of the application in the log, the values can be calculated from the number of logs and the total time.

Next, in the representative data selection in Step 1, the actual request data for a certain period (short term) of the high load applications are arranged for each fixed size to create a frequency distribution, and the class number is determined by the Sturges formula. In the Sturges formula, it is appropriate to set the number of classes to $1+\log_2 n$ when the application is used n times. After determining the number of classes and selecting the Mode class, one representative data needs to be selected from the Mode class. When selecting representative data, the implementation selects the data whose data size is closest to the center of the Mode class as the representative data.

In Step 2, by using the selected production representative data, GPU offload is performed for the applications with the highest load by the same processing as before the start of operation. It is different from before the start of operation in that the test case used for performance measurement uses production representative data instead of assumed usage data.

In Step 3, the improvement effect needs to be seen when the production environment is reconfigured to the new offload pattern. The new offload pattern search is done on the verification environment server, but the improvement of application performance is measured using the representative data of production use, and the degree of total improvement is calculated by using the frequency of production use. With this calculation, we will compare the effect when the production environment is reconfigured.

In the proposal judgement in Step 4, if the improvement is below the threshold value, no proposal is made. Frequent proposals are inconvenient for the user, so by setting the threshold to a value sufficiently larger than 1 time, it is possible to suppress the frequent occurrence of proposals and leave a case of truly effective reconfiguration. The implementation can set the threshold variably, but this time 2.0 is set.

At the time of reconfiguration approval in Step 5, information on the price after reconfiguration and the improvement effect in the verification environment is given, and the reconfiguration is proposed to the contract user. From this, the contract user can judge whether it is better to reconfigure.

At the time of reconfiguration in Step 6, the GPU processing machine stops old OpenACC, starts new OpenACC, and then reconfigures GPU offload logic. Since OpenACC needs to be stopped and started, a very short break time can occur. If we want to minimize the break time, we can build a new machine that processes new OpenACC and switch the routing to the new machine for new requests. GPU resources will be used twice, but this usage is temporary before and after switching.

5 Evaluation

The automatic offloading of the loop statement to the GPU itself has been demonstrated with [23]. In this paper, we evaluate the performance improvement effect and break time of reconfiguring GPU logic to offloading new loop statements or new applications to the usage characteristics.

5.1 Evaluation Condition

5.1.1 Evaluated Applications

The evaluated applications are mainly neural networks, Fourier transforms, and fluid calculations that are expected to be used by many users on GPUs.

There are various types of neural networks and many libraries for GPUs, but as an example, we use Darknet [31], which is written in C language. Darknet is a neural network framework that can perform various processing such as classification, detection, and nightmare, but this time, it speeds up object detection and image modification, which are the basic processing. In the offload verification before operation, detection processing (image detection) is used in the sample application installed in Darknet.

The Fourier transform process is used in various situations of monitoring in IoT such as analysis of vibration frequency. NAS.FT [32] is one of the open source applications for FFT (Fast Fourier Transform) processing. When considering an application that transfers data from a device to a network in IoT, the device is expected to perform primary analysis such as FFT processing and send it to reduce network costs. For offload verification before operation, the sample test case attached to NAS.FT is used. The basic data of Class A is that the grid size is 256*256*128 and the number of iterations is 6.

Himeno Benchmark [33] is a benchmark software used to measure the performance of incompressible fluid analysis and solves the Poisson equation solution by the Jacobi iterative method. It is frequently used for manual speedup on GPU and is used this time to demonstrate that speedup can be done automatically. The basic data used for offload is LARGE (512*256*256 grid size).

NAS.BT [34] for block diagonal solver computation and MRI-Q [35] for 3D MRI image processing are run on the same machine and receive execution requests.

5.1.2 Evaluation Method

Assuming two users, user A specifies Darknet detection and user B specifies NAS.FT for automatic offloading before operation. Each application is offloaded to each production environment, and the remaining four applications are executed on the CPU only. Different request loads are applied to the production environment for each user for a certain period of time, results are analyzed, reconfiguration to a new offload pattern that is more effective in improving performance is proposed, and the offload pattern is reconfigured after user approval.

The following conditions are set during GPU offloading.

Number of loop statements: Darknet 171, NAS.FT 81, Himeno 13, NAS.BT 120, MRI-Q 16

Number of individuals M: Less than the number of loop statements (Darknet 30, NAS.FT 30, Himeno 10, NAS.BT 30, MRI-Q 10)

Number of generations T: less than the number of loop statements (Darknet 20, NAS.FT 20, Himeno 10, NAS.BT 20, MRI-Q 10)

Goodness of fit: (processing time)$^{-1/2}$. The shorter the processing time, the higher the goodness of fit. The (-1/2) power prevents the goodness of fit of a particular individual with a short processing time from becoming too high. This prevents the search area from becoming too narrow.

Selection: Roulette selection. To keep (not crossed or mutated) best goodness of fit gene in a generation, elite selection to next generation is also applied.

Crossover rate Pc: 0.9

Mutation rate Pm: 0.05

The operational conditions for GPU reconfiguration are as follows.

Request load:

A: Darknet detection 16 req/h, Darknet nightmare 20 req/h, NAS.FT 4 req/h, Himeno 3 req/h, NAS.BT 2 req/h, MRI-Q 1 req/h. Darknet detection and nightmare (image modification) processing uses 3 equipped sample images with 111, 170, and 374 KB. These data are requested in a ratio of 3:5:2. Data of NAS.FT, Himeno, NAS.BT and MRI-Q are the same as the sample data.

B: Darknet detection 4 req/h, Darknet nightmare 3 req/h, NAS.FT 20 req/h, Himeno 30 req/h, NAS.BT 2 req/h, MRI-Q 1 req/h. In NAS.FT, Class W, A, and B sizes of sample data are requested in a ratio of 3:5:2. In Himeno, M, L, and XL sizes of sample data are requested in a ratio of 2:5:3. The data of Darknet detection and nightmare, NAS.BT and MRI-Q are the same as the sample data.

Long term during load analysis: 3 h

Short term during representative data selection: 1 h

Number of high load applications: 2

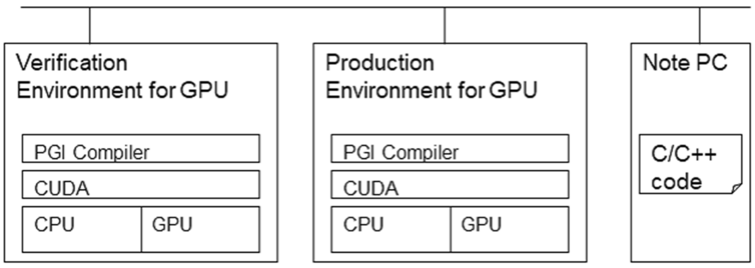
Performance improvement effect threshold: 2.0

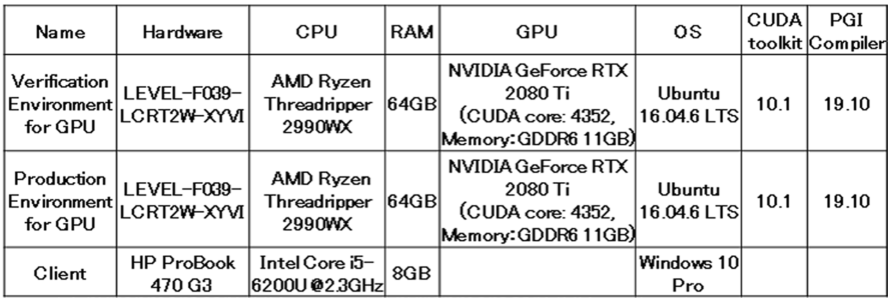
During the reconfiguration, the performance improvement effect and the processing time of each step are obtained.

5.1.3 Evaluation Environment

NVIDIA GeForce RTX 2080 Ti is used as the evaluation GPU. The machine equipped with GeForce RTX 2080 Ti is Iiyama LEVEL-F039-LCRT2W-XYVI. GPU control uses PGI compiler 19.10 and CUDA toolkit 10.1. By adding the #pragma directives to the C language program in accordance with the OpenACC syntax, GPU offload processing is performed, and reconstruction to another OpenACC program is also processed by the PGI compiler.

Figure 3 shows the evaluation environment and specifications. Here, the notebook PC specifies the application code to be offloaded, extracts the offload pattern


Verification Environment for GPU
PGI Compiler
CUDA
CPU
GPU
Production Environment for GPU
PGI Compiler
CUDA
CPU
GPU
Note PC
C/C++ code



Name	Hardware	CPU	RAM	GPU	OS	CUDA toolkit	PGI Compiler
Verification Environment for GPU	LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX	64GB	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory:GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	19.10
Production Environment for GPU	LEVEL-F039-LCRT2W-XYVI	AMD Ryzen Threadripper 2990WX	64GB	NVIDIA GeForce RTX 2080 Ti (CUDA core: 4352, Memory:GDDR6 11GB)	Ubuntu 16.04.6 LTS	10.1	19.10
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 10 Pro		

Fig. 3 Performance measurement environment

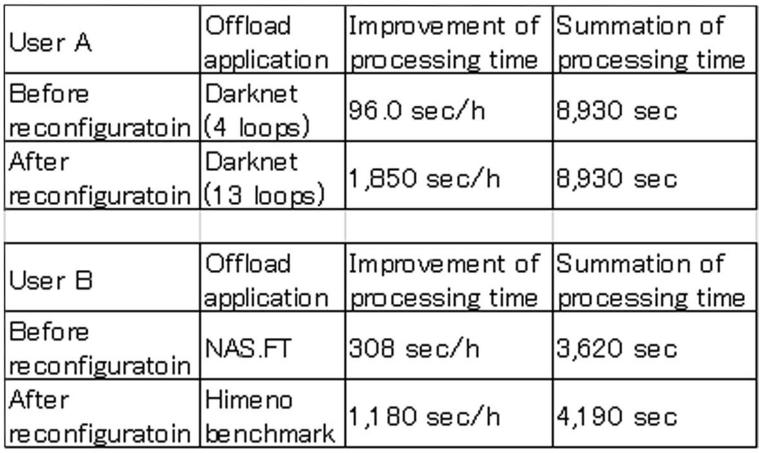
through performance measurement in the verification environment, and then deploys it to the production environment. Requests are made to the production environment applications from the notebook PC on a regular basis. The production environment request history is analyzed, a new offload pattern is extracted using the verification environment, and after user approval, the production environment is reconfigured into the new offload pattern.

5.2 Results

Figure 4 shows the degree of improvement in processing time before and after reconfiguration of users A and B, and the total processing time (corrected for improvement coefficient) for a certain period related to it.

Before the reconfiguration for user A, 4 for statements of Darknet are offloaded, the degree of improvement in the assumed data before operation is 2.92, and the loads are 16 req/h for detection processing and 20 req/h for nightmare processing after the start of operation. The total corrected processing time is 8,930 s, which can be calculated by the total actual processing time of the request*2.92. As a result of the calculation, Darknet and NAS.FT are two high load applications. Among them, after the start of operation, nightmare processing of Darknet was used many times in the production environment, thus new offload patterns for nightmare processing using representative data are searched, a new pattern is found that offloads 13 for statements, and the total processing time that multiplies the number of production uses is reduced. The degree of improvement is 96.0 s/h for Darknet before reconfiguration and 1,850 s/h for Darknet after reconfiguration.

Before the reconfiguration for user B, NAS.FT was offloaded, and the degree of improvement in the assumed data was 2.54. The total corrected processing time is 3,620 s, which can be calculated by the total actual processing time of the request*2.54. As a result of the calculation, Himeno and NAS.FT are two high



User A	Offload application	Improvement of processing time	Summation of processing time
Before reconfiguratoin	Darknet (4 loops)	96.0 sec/h	8,930 sec
After reconfiguratoin	Darknet (13 loops)	1,850 sec/h	8,930 sec

User B	Offload application	Improvement of processing time	Summation of processing time
Before reconfiguratoin	NAS.FT	308 sec/h	3,620 sec
After reconfiguratoin	Himeno benchmark	1,180 sec/h	4,190 sec

Fig. 4 Performance improvement through proposed reconfiguration

load applications. By searching for offload patterns using representative data after the start of operation and multiplying the number of production uses, the degree of improvement in processing time reduction is 308 s/h in NAS.FT before reconfiguration and 1,180 s/h in Himeno after reconfiguration.

From Fig. 4, the improvement threshold value 2.0 is checked, and in the case of user A, a reconfiguration with offload loop statement change of Darknet is proposed, and in the case of user B, a reconfiguration with offload application change from NAS.FT to Himeno is proposed.

The request analysis is small because only several hours of data is analyzed this time, but it will take longer in proportion to the size. This time, it takes about only 1 s for request analysis and representative data selection, less than a day for improvement effect calculation, and almost no time for new OpenACC start. For offload trials before the operation and new offload pattern trials during operation, it takes about 7 h to measure the performance of GAs of several tens of generations, so it takes less than a day if there are three high load applications. However, most of the processing such as analysis, including the trial of the new offload pattern, is performed in the background during the operation in the production environment, so there is no user influence. Also, regarding the conduction of production environment reconfiguration, the application may have a break time, but it takes less than 50 milliseconds even if the current OpenACC is stopped and a new OpenACC is started, and almost no effect was found. If no break time is required, it is possible to prepare a new OpenACC on the new machine and switch the routing.

It was found that the GPU processing application in operation was reconfigured to a different loop statement offload or to a different application offload in accordance with the usage characteristics of each user. It was shown that the degree of performance improvement increased above the threshold through the reconfiguration and the break time was sufficiently short.

5.3 Discussion

In the GPU offload in the previous paper, the performance of many offload patterns is measured in the verification environment using GA, and the pattern is automatically converted to a high-speed pattern. The automatic speedup has been increased by three times in Darknet, five times in NAS.FT, and five times in Himeno Benchmark. Since the GPU requires a program in accordance with the hardware characteristics, manual design is the mainstream, and automatic offload can be said to be an advance. This time, by proposing a method of reconfiguring to more appropriate logic in accordance with the usage characteristics during operation as well as before operation, we can improve the efficiency of resource utilization in GPUs where the amount of resources is limited.

The cost is considered. GPU boards such as NVIDIA Tesla T4 board cost about 2,000 USD each. Therefore, looking only at the hardware price, the price of a machine equipped with a GPU is usually twice that of a machine with only a CPU. However, in general, the cost of a data center is less than 1/3 of the cost of system development such as hardware and cloud, the operation cost of electricity cost and maintenance/operation system is more than 1/3, and the service order and other costs are about 1/3. Therefore, even if the hardware price itself doubles, we think that it is cost-effective to use this technology that improves the performance of loop statement processing, which used to take time when using CPU processing, by more than three times and reconfigures it into appropriate logic.

The time required for reconfiguration is considered. The analysis required for reconfiguration takes almost no time, but it takes a long time to search for a new offload pattern depending on the GA. This is because GA performs a large number of performance measurements for search, and the current situation is that it takes about 7 h*number of applications, thus it depends on the number of applications with high load. However, this measurement time is the time required in the verification environment and is irrelevant to users in the production environment, so the effect can be said to be small. The only thing that affects the user is the break time when reconfiguring OpenACC in the production environment, but since it is about less than 50 milliseconds, the effect is still small. In summary, the verification time of less than one day and the break time of less than 50 milliseconds are considered to be acceptable considering that the cost effectiveness is improved by the reconfiguration.

The GPU offload pattern search time can be shortened by preparing multiple GPU machines in the verification environment and performing measurements in parallel. It is also effective to reduce the number of measurements by narrowing down the number of loop statements to be recombined by the loop amount. In addition, GPU offload acceleration can be further accelerated by changing the logic to match the hardware, including the algorithm of the entire function, rather than simply offloading the loop statement. For example, the NAS.FT loop statement offload improves performance by 5 times, but when the entire Fourier transform function is offloaded to a CUDA library called cuFFT that implements logic suitable for GPU, the performance is improved by 700 times. We will consider using such a function block offload together with a loop statement offload.

This time, among the reconfigurations during operation, we specifically targeted the reconfiguration of GPU logic. Before the start of operation, the environment-adaptive software has verified the automatic conversion to multi-core CPU, GPU, FPGA, the appropriate amount of processing resources [21], and the appropriate placement location [22]. Even in the reconfiguration during operation, the amount of resources other than GPU logic can be reconfigured. We will also consider these various types of reconfigurations and aim to improve cost performance by reconfiguring during operation as a whole. They are reported in other papers such as [21, 22].

6 Related Work

The papers of [36, 37] describe techniques for automatically searching for areas that are offloaded to the GPU. The methods of [36, 37] are similar to our previous method in that they optimize the offload part by GA. However, [36] is intended for applications that are often accelerated by GPU and is evaluated with 200 generations and requires repeated execution for a long time. Our previously proposed technique, which is also used this time, aims to be able to start using a general-purpose application for CPU in a short time when accelerating it with a GPU, which is different from these other techniques.

Techniques for offload to GPU include [38–40]. [38] mentions metaprogramming and JIT (Just-In-Time) compilation usage for GPU offloading of C++ expression template, and [39, 40] work on offloading to GPU using OpenMP. There are few methods that automatically convert existing code for GPU as we aim to, without manually inserting new directives or new development models.

In this verification, we used a PGI compiler that interprets OpenACC in C/C++ language. Java is a language also often used in OSS (Open Source Software). In Java, parallel processing description in lambda format is possible from Java 8. IBM provides a JIT compiler that offloads parallel processing descriptions in lambda format to the GPU [41]. In Java, the same automation is possible by using this and performing in GA whether or not the loop is processed in GPU.

Even if the offload area can be extracted accurately, performance will not be adequate if the CPU and the amount of offload destination resources are not properly balanced. In MapReduce [42], the overall performance is improved by allocating map tasks so that the execution times of CPU and GPU are the same. When optimizing the function layout, we attempt to avoid a layout where the processing on any device becomes a bottleneck by referring to [42] and so on.

Many cases of speeding up by using GPGPU such as game processing have been reported. Regarding offload automation, there are some methods for automatic parallelization functions such as Intel compiler [28] for multi-core CPUs. In GPU offload, the processing of CPU and GPU is difficult to allocate because the memory is separate. [43] describes task scheduling when the CPU and GPU are integrated chips on the same die. We can refer to how many tasks are assigned to the CPU and GPU.

As mentioned above, there are many methods to speed up by offloading to GPU and FPGA, but the mainstream method is to manually add an instruction to parallelize whichever part with OpenMP and offload accordingly. There are few methods to automatically offload existing code. In addition, the methods only convert applications to offload to the GPU before the start of operation, and basically the speedup is done only once. There is no method to improve cost performance by feeding back the user's usage characteristics and reconfiguring the GPU offload logic of the application during operation to an appropriate logic suitable to current usage as targeted in this paper.

7 Conclusion

We proposed environment-adaptive software for automatically adapting software to the deployment destination environment and appropriately using GPU, FPGA, and so on to operate applications with high performance. In this paper, as the elemental technology, we proposed a GPU reconfiguration method that reconfigures the appropriate GPU logic during operation in accordance with the usage characteristics after the application operation starts.

Before starting operation, one application loop statement is automatically offloaded to the GPU. In the proposed method, the applications with the high CPU processing time load are acquired from the actual request data at regular intervals, and the corresponding representative test cases are also acquired. Next, the new offload patterns that speed up representative test cases are extracted through trial measurements in the verification environment. This is almost the same as offload before the start of operation. Next, the processing time of the current offload pattern and the new offload patterns are measured, and the processing time improvement is calculated on the basis of the frequency of production use. Here, if the new offload pattern has an effect greater than the threshold of the current pattern, the implementation proposes reconfiguration to the user. After obtaining the user's consent, the implementation stops OpenACC in the production environment and reconfigures GPU logic by starting the new OpenACC. The experiment showed that the GPU logic was reconfigured to offload other loop statements or other applications by reconfiguring the application that was automatically offloaded to the GPU during operation in accordance with the usage characteristics. The reduction in processing time was improved by reconfiguration, and at the same time, reconfiguration was performed with almost no break time, and the effectiveness of the method was demonstrated.

In the future, we will further expand the scope of reconfiguration during operation, reconfigure not only GPU logic but also multi-core CPU logic, and reconfigure settings other than offload logic such as processing resource amount and allocation. We will also attempt to improve performance by reconfiguration during operation and evaluate cost performance improvement.

Author Contributions The author of this manuscript is only YY. YY proposed the method, implemented the simulation program, conducted the simulation, discussed the results and so on. YY wrote all of contents.

Funding The author declares no funding with this manuscript.

Data Availability Not Applicable.

Declarations

Conflict of interest The author declares no competing interest with this manuscript.

Ethical Approval and Consent to Participate Not applicable.

Consent for Publication Not applicable. Only Yoji Yamato did all of this research and Yoji Yamato approved the publication of this manuscript.

References

1. Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D.: A reconfigurable fabric for accelerating large-scale datacenter services. In: Proceedings of the 41th Annual International Symposium on Computer Architecture (ISCA'14), pp.13–24 (2014)
2. AWS EC2 web site, https://aws.amazon.com/ec2/instance-types/
3. Sefraoui, O., Aissaoui, M., Eleuldj, M.: OpenStack: Toward an open-source solution for cloud computing. Int. J. Comput. Appl. **55**(3), 38–42 (2012)
4. Yamato, Y., Nishizawa, Y., Nagao, S.: Fast restoration method of virtual resources on OpenStack. In: IEEE Consumer Communications and Networking Conference (CCNC2015), pp. 607–608 (2015)
5. Hermann, M., Pentek, T., Otto, B.: Design Principles for Industrie 4.0 Scenarios. Rechnische Universitat Dortmund (2015)
6. Yamato, Y., Fukumoto, Y., Kumazaki, H.: Proposal of real time predictive maintenance platform with 3D printer for business vehicles. Int. J. Inf. Electron. Eng. **6**(5), 289–293 (2016)
7. Yamato, Y.: Experiments of Posture Estimation on Vehicles Using Wearable Acceleration Sensors. In: The 3rd IEEE International Conference on Big Data Security on Cloud (BigDataSecurity 2017), pp. 14–17 (2017)
8. Yamato, Y., Takemoto, M., Shimamoto, N.: Method of service template generation on a service coordination framework. In: 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004) (2004)
9. Noguchi, H., Demizu, T., Hoshikawa, N., Kataoka, M., Yamato, Y.: Autonomous Device Identification Architecture for Internet of Things. In: 2018 IEEE 4th World Forum on Internet of Things (WF-IoT 2018), pp. 407–411 (2018)
10. Noguchi, H., Kataoka, M., Yamato, Y.: Device identification based on communication analysis for the internet of things. IEEE Access (2019). https://doi.org/10.1109/ACCESS.2019.2910848
11. Noguchi, H., Demizu, T., Kataoka, M., Yamato, Y.: Distributed search architecture for object tracking in the internet of things. IEEE Access (2018). https://doi.org/10.1109/ACCESS.2018.2875734
12. Evans, P. C., Annunziata, M.: Industrial Internet: Pushing the Boundaries of Minds and Machines. In: Technical Report of General Electric (GE) (2012)
13. Sterling, T., Anderson, M., Brodowicz, M.: High performance computing: modern systems and practices. Morgan Kaufmann, Cambridge, MA, ISBN 9780124202153 (2018)
14. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
15. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, New York (2011)
16. Gosling, J., Joy, B., Steele, G.: The Java Language Specification, 3rd Edn. Addison-Wesley. ISBN 0-321-24678-0 (2005)
17. Yamato, Y., Demizu, T., Noguchi, H., Kataoka, M.: Automatic GPU offloading technology for open IoT environment. IEEE Internet Things J. (2018). https://doi.org/10.1109/JIOT.2018.2872545
18. Yamato, Y.: Study and evaluation of automatic gpu offloading method from various language applications. Int. J. Parallel Emergent Distrib. Syst. (2021). https://doi.org/10.1080/17445760.2021.1971666
19. Yamato, Y.: Improvement proposal of automatic GPU offloading technology. In: The 8th International Conference on Information and Education Technology (ICIET 2020), pp. 242–246 (2020)
20. Yamato, Y.: Proposal of automatic offloading for function blocks of applications. In: The 8th IIAE International Conference on Industrial Application Engineering 2020 (ICIAE 2020), pp. 4–11 (2020)

21. Yamato, Y.: Proposal and evaluation of adjusting resource amount for automatically offloaded applications. In: Cogent Engineering. Taylor and Francis, New York (Vol. 9, Issue 1). https://doi.org/10.1080/23311916.2022.2085467 (2022)
22. Yamato, Y.: Study and evaluation of optimum location deployment for environment adaptive applications. Int. J. Parallel Emergent Distrib. Syst. (2022). https://doi.org/10.1080/17445760.2022.2088749
23. Yamato, Y.: Study and evaluation of improved automatic GPU offloading method. Int. J. Parallel Emerg. Distrib. Syst. (2021). https://doi.org/10.1080/17445760.2021.1941010
24. Fung, J., Steve, M.: Computer vision signal processing on graphics processing units. In: 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 5, pp. 93–96 (2004)
25. Xilinx SDK web site, https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/lyx1504034296578.html
26. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC-first experiences with real-world applications. In: Euro-Par 2012 Parallel Processing, pp. 859–870 (2012)
27. Wolfe, M.: Implementing the PGI accelerator model. In: ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 43–50 (2010)
28. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for Intel SMP architectures. In: Fourth European Workshop on OpenMP (2002)
29. Holland, J.H.: Genetic algorithms. Scientific American **267**(1), 66–73 (1992)
30. Clang website, http://llvm.org/
31. Darknet website, https://pjreddie.com/darknet/
32. NAS.FT website, https://www.nas.nasa.gov/publications/npb.html
33. Himeno benchmark web site, http://accc.riken.jp/en/supercom/
34. NAS.BT website, https://www.nas.nasa.gov/publications/npb.html
35. MRI-Q website, http://impact.crhc.illinois.edu/parboil/
36. Tomatsu, Y., Hiroyasu, T., Yoshimi, M., Miki, M.: gPot: Intelligent compiler for GPGPU using combinatorial optimization techniques. In: The 7th Joint Symposium between Doshisha University and Chonnam National University (2010)
37. Bruce, Bobby R., Petke, J.: Towards automatic generation and insertion of OpenACC directives. RN, 18.04: 04 (2018)
38. Chen, J., Joo, B., Watson III, W., Edwards, R.: Automatic offloading C++ expression templates to CUDA enabled GPUs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 2359–2368 (2012)
39. Bertolli, C., Antao, S. F., Bercea, G. T., Jacob, A. C., Eichenberger, A. E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D., O'Brien, K.: Integrating GPU support for OpenMP offloading directives into Clang. In: ACM Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM'15) (2015)
40. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09) (2009)
41. Ishizaki, K.: Transparent GPU exploitation for Java. In: The Fourth International Symposium on Computing and Networking (CANDAR 2016) (2016)
42. Shirahata, K., Sato, H., Matsuoka, S.: Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters. In: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 733–740 (2010)
43. Kaleem, R., Barik, R., Shpeisman, T., Hu, C., Lewis, B. T., Pingali, K.: Adaptive heterogeneous scheduling for integrated GPUs. In: 2014 IEEE 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 151–162 (2014)

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. He is a senior member of IEEE and IEICE, and a member of IPSJ.