Proposal and evaluation of adjusting resource amount for automatically offloaded applications

Yoji Yamato

Abstract: Heterogeneous hardware other than a small-core central processing unit (CPU) such as a graphics processing unit (GPU), field-programmable gate array (FPGA), or many-core CPU is increasingly being used. However, to use heterogeneous hardware, programmers must have sufficient technical skills to utilize OpenMP, CUDA, and OpenCL. On the basis of this, I have proposed environment-adaptive software that enables automatic conversion, configuration, and high-performance operation of once-written code, in accordance with the hardware to be placed. Although it has been considered to convert the code in accordance with the offload devices, no method has been developed to set the amount of resources appropriately after automatic conversion. In this paper, as a new element of environment-adapted software, I examine a method of optimizing the amount of resources of the CPU and offload device in order to operate the application with high-cost performance. Through the offloading of multiple existing applications, I demonstrate that the proposed method can meet the user’s request and set an appropriate amount of resources.

Subjects: Systems & Computer Engineering; Computer Engineering; Computer Science; General; Information & Communication Technology; ICT; Testing

Keywords: Environment adaptive software; GPGPU; Automatic Offloading; evolutionary computation; Resource Configuration

1. Introduction

As Moore’s Law slows down, a central processing unit’s (CPU’s) transistor density cannot be expected to double every 1.5 years. To compensate for this, more systems are using heterogeneous hardware, such as graphics processing units (GPUs), field-programmable gate arrays.
(FPGAs), and many-core CPUs. For example, Microsoft's search engine Bing uses FPGAs (Putnam et al., 2014), and Amazon Web Services (AWS) provides GPU and FPGA instances (AWS EC2 web site, 2021) using cloud technologies (e.g., (Sefraoui et al., 2012; Yamato, 2015a, 2015b, 2016; Yamato et al., 2015)). Systems with Internet of Things (IoT) devices are also increasing (e.g., (Evans & Annunziata, 2012; Hermann et al., 2016; Yamato, 2017a, 2017b; Yamato, Fukumoto, Kumazaki et al., 2017a, 2017b; Yamato et al., 2016)).

However, to properly utilize devices other than small-core CPUs in these systems, configurations and programs must be made that consider device characteristics, such as Open Multi-Processing (OpenMP; Sterling et al., 2018), Open Computing Language (OpenCL; Stone et al., 2010), and Compute Unified Device Architecture (CUDA; Sanders & Kandrot, 2011). In addition, embedded software skills are needed for IoT devices detail controls. Therefore, for most programmers, skill barriers are high.

The expectations for applications using heterogeneous hardware are becoming higher; however, the skill hurdles are currently high for using them. To surmount these hurdles, application programmers should only need to write logics to be processed, and then software should adapt to the environments with heterogeneous hardware to make it easy to use such hardware. Java (Gosling et al., 2005), which appeared in 1995, caused a paradigm shift in environment adaptation that enables software written once to run on another CPU machine. However, no consideration was given to the application performance at the porting destination.

Therefore, I previously proposed environment-adaptive software that effectively runs once-written applications by automatically executing code conversion and configurations so that GPUs, FPGAs, many-core CPUs, and so on can be appropriately used in deployment environments. For an elemental technology for environment-adaptive software, I also proposed a method for automatically offloading loop statements and function blocks of applications to GPUs or FPGAs (Yamato, 2021; Yamato et al., 2018).

This paper proposes a method to optimize the resource amount of CPU and offload device in order to operate the application with high-cost performance when offloading a normal CPU program to a device such as a GPU. I demonstrate the effectiveness of the proposed method through GPU offload of existing applications. This paper is an improvement of an international conference CANDAR short paper, which adds considerations to proposals, implementations, evaluations, and discussions.

The rest of this paper is organized as follows. In Section 2, I review technologies on the market and our previous proposals. In Section 3, I propose a method for determining the amount of resources for the CPU and offload device when it is automatically offloaded. In Section 4, I describe the implementation of the proposed method. In Section 5, I evaluate the proposed method through the offloading of existing applications to GPU. In Section 6, I describe related work, and in Section 7, I conclude the paper.

2. Existing technologies

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

CUDA is a major development environment for general-purpose GPUs (GPGPUs) (e.g., (Fung & Steve, 2004) that use GPU computational power for more than just graphics processing. To control heterogeneous hardware uniformly, the OpenCL specification and its software development kit (SDK) are widely used. CUDA and OpenCL require not only C language extension but also additional descriptions such as memory copy between GPU or FPGA devices and CPUs. 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 specified parts into device-oriented codes on the basis of directives. Open accelerators (OpenACC; Wienke et al., 2012) and OpenMP are examples of directive-based specifications, and the Portland Group Inc. (PGI) compiler (Wolfe, 2010) and gcc are examples of compilers that support these directives.

In this way, CUDA, OpenCL, OpenACC, OpenMP, and others support GPU, FPGA, or many-core CPU offload processing. Although processing on devices can be done, sufficient application performance is difficult to attain. For example, when users use an automatic parallelization technology, such as the Intel compiler (Su et al., 2002) for multi-core CPUs, possible areas of parallel processing such as “for” loop statements are extracted. However, naive parallel execution performances with devices are not high because of overheads of CPU and device memory data transfer. To achieve high application performance with devices, CUDA, OpenCL, or so on needs to be tuned by highly skilled programmers, or an appropriate offloading area needs to be searched for by using the OpenACC compiler or other technologies.

Therefore, users without skills in using GPU, FPGA, or many-core CPU will have difficulty attaining high application performance. Moreover, if users use automatic parallelization technologies to obtain high performance, much effort is needed to determine whether each loop statement is parallelized.
2.2. Previous proposals

On the basis of the above background, to adapt software to an environment, I previously proposed environment-adaptive software (Yamato, 2019), the processing flow of which is shown in Figure 1. The environment-adaptive software is achieved with an environment-adaptation function, test-case database (DB), code-pattern DB, facility-resource DB, verification environment, and production environment.

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

I will summarize this section. Because most offloading to heterogeneous devices is currently done manually, I proposed the concept of environment-adaptive software and automatic offloading to heterogeneous devices. However, after automatic conversion, the adjustment of the resource amount of the offload device has not been examined (corresponding to Step 4). Therefore, in this paper, the objective is to optimize the amount of resources of the CPU and offload device when it is automatically offloaded to a device such as a GPU.

With this paper’s contribution, this is a typical use case. When a user requests a platform operator to execute an application, the operator converts it for GPU or FPGA and sets an appropriate amount of resources according to the user budget. The operator presents the actual performance and price setting results to the user, and after the user approves, the actual service usage and usage fee billing are started. The usage fee includes the price of optimization processing such as conversion and resource amount setting.

3. Appropriate resource configuration of CPU and offload device

To embody the concept of environment-adaptive software, I have so far proposed automatic GPU and FPGA offload of program loop statements (Yamato, 2021, 2019), auto offload of program function blocks (Yamato, 2020), and automatic offload from various languages to mixed environments.

Based on these studies, Section 3.1 reviews automatic GPU offload of loop statements as an example of automatic offload and Section 3.2 studies an appropriate resource ratio between CPU and offload device. In Section 3.3, I will consider the determination of the amount of resources and its verification.
3.1. Automatic GPU offloading of loop statements

There are many cases where a program running on a normal CPU is speeded up by offloading it to a device such as a GPU or FPGA, but there are few cases where this offloading is automatically conducted. For automatic GPU offloading of loop statements, several methods have been proposed (Yamato, 2019; Yamato et al., 2018).

First, as a basic problem, the compiler can find the limitation that this loop statement cannot be processed in parallel on the GPU, but it is difficult to find out whether this loop statement is suitable for parallel processing on the GPU. Loops with a large number of loops are generally said to be more suitable, but it is difficult to predict how much performance will be achieved by offloading to the GPU without actually measuring it. Therefore, it is often the case that the instruction to offload this loop to the GPU is manually given and the performance measurement is tried. On the basis of that, Yamato et al. (2018) proposes automatically finding an appropriate loop statement that is offloaded to the GPU with a genetic algorithm (GA; Holland, 1992), which is an evolutionary computation method. From a general-purpose program for normal CPUs that does not assume GPU processing, the proposed method first checks the parallelizable loop statements. Then, for the parallelizable loop statements, it sets 1 for GPU execution and 0 for CPU execution. The value is set and geneticized, and the performance verification trial is repeated in the verification environment to search for an appropriate area. By narrowing down to parallel processing loop statements and holding and recombining parallel processing patterns that can be accelerated in the form of gene parts, patterns that can be efficiently accelerated are explored from the huge number of parallel processing patterns (see Figure 2). The paper of Yamato (2019) proposes transferring variables efficiently. Regarding the variables used in the nested loop statement, when the loop statement is offloaded to the GPU, the variables that do not have any problem even if CPU-GPU transfer is performed at the upper level are summarized at the upper level. This is because if CPU-GPU transfer is performed at the lower level of the nest, the transfer is performed at each lower loop, which is inefficient. I also propose a method to further reduce CPU-GPU transfers. Specifically, for not only nesting but also variables defined in multiple files, GPU processing and CPU processing are not nested, and for variables where CPU processing and GPU processing are separated, the proposed method specifies to transfer them in a batch.

Figure 2. Automatic GPU offload of loop statements.
Regarding GPU offload of loop statement, automatic offload is possible by optimization using an evolutionary computation method and reduction of CPU-GPU transfer.

### 3.2. Appropriate resource ratio configuration of CPU and offload device

A program for a normal CPU can be automatically offloaded to an offload device such as a GPU by the method such as in Section 3.1. In this subsection, I will consider the appropriate resource ratio between the CPU and the offload device after the program conversion that offloads the device.

Currently, multi-core CPUs and many-core CPUs can be flexibly allocated to what percentage of all cores by virtualization with virtual machines (VMs) and containers. As for GPU, virtualization similar to CPU has been done in recent years by software such as NVIDIA’s vGPU (NVIDIA vGPU software web site) and multi-instance GPU (MIG), and it is becoming possible to allocate half or a quarter of GPU board resources. For FPGAs, resource usage is often represented by the number of Look Up Table and Flip-Flop settings, and unused gates can be used for other purposes. However, in FPGA, it is difficult for another user to divide the gate and use it due to security problems, and it is assumed that the same user will use the surplus gate for another purpose. In this way, it is possible to operate using a part of all resources for CPU, GPU, and FPGA, and it is important to optimize the resources of CPU and offload the device in accordance with the application in order to improve cost performance.

Even if the application can be converted into CPU and GPU processing code by a method such as in Section 3.1 and the code itself can be converted appropriately, the cost performance is low if the amount of resources between the CPU and GPU is not properly balanced. For example, when processing a certain application, if the processing time on the CPU is 1,000 seconds and the processing time on the GPU is 1 second, even if the processing offloading to the GPU is accelerated by the GPU, the CPU will be a bottleneck as a whole. In Shirahata et al. (2010), when processing tasks with the MapReduce framework using the CPU and GPU, the overall performance is improved by allocating Map tasks so that the execution times of the CPU and GPU are the same.

In this paper, on the basis of actual processing time of the test case in the verification environment, I propose setting the resource ratio so that the processing time of the CPU and the offload device are on the same order in order to determine the resource ratio between the CPU and the offload device with reference to Shirahata et al. (2010) and to avoid the processing on any device becoming a bottleneck.

As in the method in Section 3.1, I propose a method of gradually increasing the speed on the basis of the performance measurement results in the verification environment for automatic offload. Performance varies greatly depending on the processing content such as the specifications of the hardware to be processed, the data size, and the number of loops, in addition to the code structure. Therefore, it is difficult to statically predict the performance from the code alone, and dynamic performance measurement is required. When the code is converted by the method in Section 3.1, the performance measurement result in the verification environment has already been acquired, and thus, the resource ratio is set using the result.

When measuring performance, we specify a test case and perform the measurement. For example, there is the case where the processing time of the test case in the verification environment is CPU processing: 10 seconds and GPU processing: 5 seconds. Since the resources on the CPU side are doubled and the processing time is considered to be the same, the resource ratio is 2:1. In particular, the user’s request to speed up a certain specified process offload is reflected by preparing a test case including the process and speeding up the test case by a method such as that in Section 3.1.
3.3. Appropriate resource-amount configuration of CPU and offload device

Since the resource ratio was decided in Section 3.2, then the application is deployed in the production environment. When the application is deployed to a production environment, the resource amount is determined by maintaining the resource ratio as much as possible so as to meet the cost requirement specified by the user. If the budget of a user's request is not enough to maintain the resource ratio, at least minimum units of CPU and GPU are secured, and the price of resources is provided to the user to judge whether to use them.

After allocating the program after securing resources in the production environment, automatic verification is performed to confirm that the program operates normally before the user uses it. In automatic verification, performance verification test cases and regression test cases are executed. Since the program is converted on the basis of the performance test results on the verification environment GPU, we find that the expected performance is obtained even on the production environment GPU. In the performance verification test case, the assumed test case specified by the user is performed using an automatic test execution tool such as Jenkins, and the processing time, throughput, and so on are measured. In the regression test case, information on software is acquired such as the middleware and operating system (OS) installed in the system, and the corresponding regression test is executed using Jenkins and so on. Examination for performing these automatic verifications with a small number of test case preparations has been studied in (Yamato, 2015c) and is conducted in this study.

In the test case, to check that the calculation result is not invalid even when it is offloaded, the difference between the calculation results with the case where it is not offloaded is also checked. The PGI compiler has the PCAST function with pgi_compare and acc_compare APIs to check the difference in calculation results when the GPU is and is not used. Note that the calculation results may not completely match even if parallel processing is correctly offloaded, such as the rounding error is different between the GPU and CPU. Therefore, the differences are checked on the basis of the IEEE 754 specifications or so on, and results are presented to the user judge whether to accept differences or not.

As a result of the automatic verification, information on the processing time, throughput, calculation result difference, and regression test execution result are presented to the user. The secured resources (number of VMs, specifications, etc.) and their costs are also presented to the user, and the user judges whether to start operation by referring to the information.

4. Implementation

4.1. Tools to use

In this section, I explain the implementation to evaluate the proposed method’s effectiveness. To evaluate the method, I use C/C++ language applications and appropriate resource-amount configuration with GPU offloading. Two boards of NVIDIA Tesla T4 are used. This implementation only configures GPU resources, but FPGA or many-core CPU resources can be configured with the same major flow.

The major flow is to find a suitable offload pattern through iterative tests, calculate an appropriate resource ratio from the execution time of the determined pattern, and determine the amount of resources allocated according to the user budget. This is common to both GPU and FPGA. However, in detail, there are differences. The GPU uses GA in the tests, but the FPGA conducts the iterative tests after narrowing down the loop sentences that are candidates for offloading due to arithmetic intensity and so on. This is because FPGAs take more than 6 hours to compile for a single measurement. To allocate the resources to be used, GPUs divide the resources
by the function of NVIDIA vGPU (NVIDIA vGPU software web site). There is no such virtualization mechanism in FPGAs. The circuit is configured unused resources.

To control the GPU, CUDA Toolkit 10.1 and PGI compiler 19.10, which is an OpenACC compiler for C/C++/Fortran language, are used. The PGI compiler generates byte code for GPU by specifying loop statements such as for statements with OpenACC directives #pragma acc kernels and #pragma acc parallel loop and enables GPU offloading and execution.

GPU virtualization is performed using the virtual Compute Server of NVIDIA vGPU GPU resources such as device memory are flexibly divided and allocated.

For parsing C/C++ language, I use the LLVM/Clang 6.0 parsing library (libClang’s Python binding; Clang web site, 2021). LibClang Python Binding has some bugs for the operator extraction function of expressions, which I use, and the old version has workaround of it. Therefore, I use version 6.0.

To acquire processing time of GPU and CPU, I use nvprof (NVIDIA profiler) of CUDA. Our tool calculates GPU and CPU processing time from the time stamp value. The GPU calculation time is a period during execution of the kernel function, and the CPU-GPU transfer time is a period while data are being transferred. Their sum is the GPU processing time. The CPU processing time is the value obtained by subtracting the GPU processing time from the total processing.

I implemented the proposed method with Perl 5 and Python 3.9. Perl controls GA processing, and Python controls other processing such as parsing.

4.2. Implementation behavior

The operation outline of the implementation is explained here. The implementation analyzes the code by using parsing libraries when there is a request to offload the applications. Next, loop statement offload trials are performed, and the highest performance pattern is set as the final solution through performance measurement in a verification environment. The implementation calculates the appropriate resource ratio between CPU and GPU on the basis of the performance measurement results for the final solution pattern. It determines the amount of resources from the appropriate resource ratio and the cost conditions requested by the user.

4.2.1. Loop statement offload verifications

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

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

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

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

The implementation compiles the code in which the instructions are inserted. It deploys the
compiled executable files and measures the performances. In the performance measurement,
along with the processing time, the implementation checks whether the calculation result is valid
or not. For example, the PCAST function of the PGI compiler can check the difference in calculation
results. If the difference is large and not allowable, the implementation sets the processing time to
a huge value.

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

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

4.2.2. Resource ratio and resource-amount configuration
To optimize the resource ratio, the performance measurement results when determining the final
solution of the offload pattern are used. The implementation determines the resource ratio so that
the processing times of the CPU and GPU are on the same order from the processing time results of
the test case. For example, if the processing time of the test case is CPU processing: 10 seconds
and GPU processing: 5 seconds, the resources on the CPU side are doubled and the processing time
is considered to be about the same, so the resource ratio is 2:1. Since the core number of VMs is an
integer, the resource ratio is rounded to an integer ratio when calculating from the processing
time. Also, if the processing times of the CPU and GPU are extremely different, making the
processing times the same will lead to deterioration of cost performance, so even if the processing
times are extremely different, the ratio should be up to 4:1. 4:1 requires 5 VMs in typical. Here, it is
referred to one of the public clouds that has the average number of about 5 contracted VMs of
each general user.

Once the resource ratio is determined, the next step is to determine the amount of resources
when deploying the application in the production environment. The implementation determines
the core number of VMs by maintaining the resource ratio as much as possible so as to satisfy the
cost request specified by the user at the time of the offload request when determining the
resource amount. Specifically, it selects the maximum resources while keeping the ratio within
the cost range. For example, when CPU VM is 20 USD/month and GPU is 100 USD/month, a resource
ratio is 2:1 is appropriate, and if the user has a cost request of 150 USD or less per month, the
implementation selects 2 unit sizes of CPU VM and 1 unit size of GPU. If the resource ratio cannot
be kept within the cost range, selections are started with minimum CPU unit and minimum GPU
unit and the implementation determines the resource amount so that it is as close to the
appropriate resource ratio as possible. For example, if the budget is less than 120 USD/month,
the resource ratio cannot be kept, but minimum one unit is kept for the CPU and minimum one unit
for the GPU. Once the amount of resources is determined, the implementation uses the vGPU
virtual Compute Server function to divide the GPU resources and then deploys the application to
the divided resources.
5. Evaluation
Since the automatic offload of the loop statement to the GPU is evaluated by Yamato (2019), in this paper, it is possible to offload by optimizing the resources in accordance with the user’s cost request, not the effect of GPU offload.

5.1. Evaluation method

5.1.1. Evaluated applications
The evaluated applications are the Fourier transform, matrix calculation, and fluid analysis, which are expected to be used by many users.

The Fast Fourier Transform (FFT) is used in various situations of monitoring in IoT such as analysis of vibration frequency. NAS.FT (NAS.FT web site) is one of the open-source applications for FFT processing. It calculates the 2,048 × 2,048 size of the built-in sample test. 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 the result to reduce network costs.

Matrix calculation is used in various situations of machine learning analysis. Polybench 3 mm (Polybench 3mm web site) performs three 4,000 × 4,000 matrix product calculations as a sample of matrix calculation. With the spread of IoT and artificial intelligence (AI), matrix calculation is often used in not only the cloud but also various locations such as devices, so there is a great need for automatic acceleration including existing applications.

The Himeno benchmark (Himeno benchmark web site, 2021) 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 the Himeno benchmark to verify whether it can be accelerated with our automatic offloading and resource configuration method. Data size for comparison was LARGE (512 × 256 × 256 grid calculation).

5.1.2. Experimental conditions
After inputting the code of the three target applications and the user cost request into the implemented resource-amount optimization function, the implementation attempts offloading of the loop statement to GPU to determine the offload pattern, determines the resource ratio on the basis of the performance results, determines the amount of resources that meet the cost requirements, and divides and allocates GPU resources. I show that the performance is improved and cost requirements are satisfied compared to the case when all processing is performed by a normal CPU without offloading.

The experimental conditions are as follows.

Offload applications and loop statements number: Fourier transform NAS.FT has 65 loops, matrix multiplications 3 mm has 10 loops, and the fluid analysis Himeno benchmark has 13 loops.

Number of individuals M: No more than the gene length. 20 for NAS.FT, 10 for 3 mm, and 12 for the Himeno benchmark.

Number of generations T: No more than the gene length. 20 for NAS.FT, 10 for 3 mm and 12 for the Himeno benchmark.

Goodness of fit: (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 search range narrowing due to the
Selection algorithm: Roulette selection and Elite selection. Elite selection means that one gene with a 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

Cost: One standard size CPU VM is 20 USD/month, and one standard size GPU board is 100 USD/month. When resource sizes are 1/2, 1/4, or so on, costs also become 1/2, 1/4, and so on.

User cost requirement: 150 USD/month, 70 USD/month

5.1.3. Experimental environment

Regarding GPU, I use two boards of NVIDIA Tesla T4 (CUDA core: 2560, Memory: GDDR6 16GB) and NVIDIA Quadro P6000 (CUDA core: 1792, Memory: GDDR5 8GB). I use CUDA Toolkit 10.1 and PGI compiler 19.10 for GPU control. NVIDIA vGPU virtual Compute Server virtualizes GPU resources. Using vGPU, Tesla T4 resources are divided, and resources of 1 board are divided into 1, 2, and 4 parts. Kernel-based VM (KVM) of RHEL7.9 is used for CPU virtualization. The resource of the VM that becomes 1 standard size is 2 cores and 16GB RAM. Half size (1 core), standard size (2 cores), and double size (4 cores) can be selected. For example, when setting the CPU and GPU resources with standard sizes at a time, our implementation virtualizes CPU and GPU resources and links the 2-core CPU and Tesla T4 1 board. Minimum unit sizes are 1-core CPU and 1/4 GPU board. Figure 3 shows the experimental environment and specifications. Here, the application code used by the user is specified from the client notebook PC, tuned using the bare metal verification machine, and then deployed to the virtual running environment for the actual use.
5.2. Results

For an application that is expected to be used by many users in IoT or so on, I found that the amount of resources is appropriated by automatic GPU offload of Fourier transform, matrix calculation, and fluid analysis.

Figure 4 shows an example of speeding up the Fourier transform using the verification machine #2. Regarding NAS.FT, offloading the loop statement was tried, and the performance change at that time is shown. Figure 4 graphs the maximum performance of each generation and the number of GA generations, and the performance is shown in comparison with the case of processing with only the CPU. Figure 4 shows that the 10th generation achieved about 5 times the performance. In addition, the same gene pattern with high fitness often occurs in GA, and the offload extraction process can be completed within 6 hours. The final solution of GA was 5 times the performance, and the processing time ratio of the test case was CPU:GPU = 2:1. Therefore, the appropriate resource ratio is 

\[ \text{CPU:GPU} = 2:1 \]

Figure 5 shows the performance improvement after adjusting the amount of resources. The appropriate resource ratio and the set resource amount when the user cost requirements are high and low for three applications using the GPU verification machine #1 with vGPU are also shown. The performance improvement is of the degree compared with the processing time in one standard size CPU VM. Cost performance is the value obtained by dividing the degree of performance improvement by the cost. For comparison, the cost performance value for standard size CPU and GPU is set to 1.

Regarding NAS.FT, the processing times with one standard size are CPU 5.50 sec and GPU 2.36 sec, so the appropriate resource ratio from the processing time ratio is 

\[ \text{CPU:GPU} = 2:1 \]

In the case of high-cost requirements, 2 units of CPU and 1 unit of GPU are automatically selected,
and in the case of low-cost requirements, 1 unit of CPU and 0.5 units of GPU are automatically selected and the degrees of performance improvement are 4.6 and 3.1 times, respectively. The cost performance values are 1.2 and 1.7 times, respectively. Regarding 3 mm, the processing times with one standard size are CPU 3.14 sec and GPU 0.09 sec, so the appropriate resource ratio from the processing time ratio is CPU:GPU = 4:1 with the upper ratio condition. In the case of high-cost requirements, 2 units of CPU and 0.5 units of GPU are automatically selected, and in the case of low-cost requirements, 1 unit of CPU and 0.25 unit of GPU are automatically selected and the degrees of performance improvement are 6.5 and 5.6 times, respectively. The cost performance values are 1.2 and 2.0 times, respectively. Regarding the Himeno benchmark, the processing times with one standard size are CPU 0.47 sec and GPU 1.81 sec, so the appropriate resource ratio from the processing time ratio is CPU:GPU = 1:4. In the case of high-cost requirements, 0.5 units of CPU and 1 unit of GPU are automatically selected, and in the case of low-cost requirements, 0.5 units of CPU and 0.5 units of GPU are automatically selected and the degrees of performance improvement are 20 and 16 times, respectively. The cost performance values are 1.2 and 1.9 times, respectively.

Depending on the application, the performance and the degree of improvement vary, but automatic acceleration with GPU offload can be achieved by configuring an appropriate amount of resources within 6 hours.

As described above, in the proposed method, a certain degree of improvement can be obtained by setting the resource amount so as to satisfy the price condition even if the appropriate resource ratio cannot be perfectly satisfied.

5.3. Discussion
The offload of the loop statement to the GPU in the previous research is a method of measuring the performance of multiple offload patterns in a verification environment and making it a high-speed pattern. For example, even Darknet, which is a large-scale application with more than 100 for statements, automatically offloads to the GPU and has been tripled in speed. However, since not all users can fully use the GPU and CPU, the amount of resources needed to be optimized so that the cost performance would be high among the virtualized resources. The proposed method performs automatic GPU offload and high performance can be achieved within the cost range by optimizing the amount of resources to achieve high performance after satisfying the cost requirements (1.2–2 times cost performances).

Regarding the offloading effect with hardware price, the GPU board price is about 2,000 USD. Therefore, a server with a GPU board price costs about two times as much as that for only CPU. In
data center systems such as cloud systems, the initial hardware, development, and verification costs are about 1/3 of the total cost; electricity power and operation/maintenance costs are more than 1/3; and costs of other expenses, such as service orders, are less than 1/3. Therefore, I think automatically improving application performance by more than three times will have a sufficiently positive effect even though the hardware price is about two times higher.

Regarding the time until the service starts to be used, the convergence time is based on GA, and it took about 6 hours in this study, the same as in the previous paper. Performance measurement from compilation takes about 3 minutes, and a solution takes time to search for depending on the number of individuals and generations, but compilation and measurement of the same gene pattern are omitted, so the search is completed in 6 hours. After the offload pattern is confirmed, it takes almost no time to calculate the appropriate resource ratio and configure the appropriate resource amount. When providing the service, we will provide a trial use free for the first day, try speeding up in the verification environment during that time, and from the second day, we will be able to provide the production service using GPU as well. Therefore, taking 1 day to start providing this offloading service is acceptable.

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

Regarding the optimization of the amount of resources, the implementation handled only the GPU, but it is possible to calculate the resource ratio so that one of the devices does not become a bottleneck by the same method for many-core CPUs and FPGAs. Currently, in the cloud, instances of not only virtualized CPUs but also GPUs and FPGAs are provided. Therefore, future needs will increase for the proposed technology that automatically offloads devices and sets the amount of resources with high-cost performance in accordance with the cost demands of users.

6. Related work

Wuhib et al. studied resource management and effective allocation (Wuhib et al., 2012) 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 resources. The paper of Kaleem et al. (2014) is about task scheduling when the CPU and GPU are integrated chips on the same die, and we can refer to how many tasks are assigned to the CPU and GPU.

Regarding the optimal use of resources existing on the network, there is research to optimize the inserted position of virtual resources for the servers on the network (Wang et al., 2018). In this study, the optimal placement of virtual resources is determined by consideration of communication traffic. The main target of this study is facility design, which is designed by looking at the amount of traffic increase. Specifically, the cost for each application is not taken into consideration. The purpose of this paper is to set resources appropriately for each user in accordance with the user request.

Some studies focused on offloading to GPUs (Bertolli et al., 2015; Chen et al., 2012; Lee et al., 2009). Chen et al. (Chen et al., 2012) used metaprogramming and just-in-time (JIT) compilation for GPU offloading of C++ expression templates. Bertolli et al. (Bertolli et al., 2015) and Lee et al. (Lee et al., 2009) investigated offloading to GPUs using OpenMP. There have been few studies on
automatically converting existing code into a GPU without manually inserting new directives or a new development model, which I target.

The method of (Bobby & Petke, 2018; Tomatsu et al., 2010) searches for GPU offloading areas and also uses GA to search automatically. In (Tanaka et al., 2011), to achieve GPGPU from Fortran code, offloading areas were searched. However, these targets are specific applications for which many GPU-based methods have been researched to accelerate, and a huge number of tunings are needed such as calculations for 200 generations. My proposed technology, which is also used in this study, is intended to be able to start using a general-purpose application for a CPU at a certain time when accelerating it with a GPU.

For this verification, I used a PGI compiler that interprets OpenACC in the C/C++ language. Python and Java are also commonly used languages in open-source software (OSS). In Python, there is a library called Cupy (Cupy web site, 2021) that converts Numpy Interface (Numpy web site) to CUDA and makes it executable in PyCUDA. Automatic offload via Cupy is possible by converting for loop statements to Numpy IF. From Java 8, parallel processing can be specified by lambda expression. IBM provides a just-in-time (JIT) compiler that offloads processing with lambda expressions to a GPU (Ishizaki, 2016). Automatic offloading is made possible by selecting an appropriate loop statement from the Java loop statement using evolutionary computation.

Generally, CUDA and OpenCL control intra-node parallel processing, and the message passing interface (MPI) controls inter-node or multi-node parallel processing. However, MPI also requires high technical skills in parallel processing. Thus, MPI concealment technology has been developed that virtualizes devices of outer nodes as local devices and enables such devices to be controlled by only OpenCL (Shitara et al., 2011). When we select multi-nodes for offloading destinations, we plan to use this MPI concealment technology.

SYCL is a single-source programming model for heterogeneous hardware (Keryell et al., 2015). In OpenCL, the host and device code are written separately, but in SYCL, they can be written in a single source. DPC++ is a SYCL compiler of Intel. Both OpenCL and SYCL require a new program to use heterogeneous hardware. SYCL targets a single code to run on multiple devices such as GPUs and FPGAs, but the single code is created by the programmer.

The work of Sommer et al. (2017) proposed a technology that can interpret OpenMP code and execute FPGA offloading. For FPGA offloading, instructions are needed to be manually added such as which parts to parallelize using OpenMP. Automatic conversion from C language of this paper is a different approach.

There are many methods to speed up processing by offloading to GPU, FPGA, and many-core CPU, but efforts are needed such as adding instructions of OpenMP manually that specify parts to parallelize and offload. These methods work to automatically offload existing code. In addition, there are only studies of conversions for offloading to GPU, FPGA, and many-core CPU, and there is no study to improve cost performance by optimizing the amount of resources of offload devices such as GPU as targeted in this paper.

7. Conclusion
For a new element of my environment-adaptive software, I proposed a method to optimize the amount of resources of the offload device to improve cost performance when automatically offloading to a device such as a GPU. Environment-adaptive software adapts applications to the environments to use heterogeneous hardware such as GPUs and FPGAs appropriately.
The proposed method works after the program is automatically converted so that it can be processed by a device such as a GPU. On the basis of the processing time of the test case in the verification environment, the proposed method determines the resource ratio with CPU and offload device so that the each processing time is on the same order to optimize the resource ratio between the CPU and the offload device and to avoid the processing on any device becoming a bottleneck. The proposed method is then deployed to the production environment on the basis of its resource ratio, but keeps the appropriate resource ratio as much as possible to determine the amount of resources so as to meet the cost requirements specified by the user. After provisioning the amount of resources and deploying the program, to determine whether the proposed method performs as expected before the user uses it. To confirm operation validity, automatic verification is performed, and the results of the performance, calculation validity, and normal operation with the resource cost are provided to the user, who then judges whether to actually use the resources or not.

I demonstrated the effectiveness of the proposed method by automatically configuring resource amounts for existing applications that were automatically offloaded to GPUs. In the future, I will consider optimizing the placement location of offloading devices, such as whether to place them in the cloud or on the edge. Furthermore, I will study reconfiguration of resource amounts, locations, and software parts of offloading during the operation on the basis of changes in user request trends.

Funding
The author received no direct funding for this research.

Author details
Yoji Yamato
E-mail: yoji.yamato.wo@hco.ntt.co.jp
1 Network Service Systems Laboratories, NTT Corporation, 3-9-11 Midori-cho, Tokyo 180-8585, Japan.

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

Citation information
Cite this article as: Proposal and evaluation of adjusting resource amount for automatically offloaded applications, Yoji Yamato, Cogent Engineering (2022), 9: 2085647.

References
AWS EC2 web site 2021, https://aws.amazon.com/ec2/instance-types/
Bertoli, 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,” ACM Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM’15), November, 2015.
Bobby, R. B., & Petke, J. (2018). Towards automatic generation and insertion of openACC directives. RN, 18 (4), 04 http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/RN_18_04.pdf
Chen, J., Joo, B., Watson, W., III, & Edwards, R., “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.
Clang web site. (2021). http://llvm.org/
Cupy web site 2021, https://www.preferred.jp/en/projects/cupy/
Evans, P. C., & Annunziata, M., “Industrial internet: Pushing the boundaries of minds and Machines,” Technical report of General Electric (GE) (General Electric), November, 2012.
Fung, J., & Steve, M., “Computer vision signal processing on graphics processing units,” 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 5, pp.93–96, 2004.
Gosling, J., Joy, B., & Steele, G. (2005). The java language specification, third edition. Addison-Wesley.
Hermann, M., Pentek, T., & Otto, B. (2016). Design principles for industry 4.0 scenarios. 49th Hawaii international conference on system sciences (HICSS) 3928–3937.
Himeno benchmark web site 2021, https://jl.sikken.jp/en/supercom/documents/himenobmt/Holland, J. H. (1992). Genetic algorithms. Scientific American, 267(1), 66–73. https://doi.org/10.1038/scientificamerican0792-66
Ishizaki, K., “Transparent GPU exploitation for Java,” The fourth international symposium on computing and networking (CANDAR 2016), 2016.
Kaleem, R., Barik, R., Shepisman, T., Hu, C., Lewis, B. T., & Pingsoli, K., “Adaptive heterogeneous scheduling for integrated GPUs,” 2014 IEEE 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp.151–162, August, 2014.
Keryell, R., Reyes, R., & Howes, L., “Khronos SYCL for OpenCL: A tutorial,” Proceedings of the 3rd International Workshop on OpenCL, 2015.
Lee, S., Min, S. J., & Eigenmann, R., “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.
NASA web site, https://www.nasa.gov/publications/nbp.html
Numpy web site, https://numpy.org/
NVIDIA vGPU software web site, https://docs.nvidia.com/grid/index.html
Polybench 3mm web site, https://web.cse.ohio-state.edu/pouchet.2/software/polybench/
Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantineides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hromat, A., Kim, J.-Y., Lanka, S., Laurus, J., Peterson, E., Pope, S., ... Burger, D., “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.

Sanders, J., & Kandrot, E. (2011). CUDA by example: An introduction to general-purpose GPU programming. Addison-Wesley.

Sefraoui, O., Aissaoui, M., & Euludj, M. (2012). OpenStack: Toward an open-source solution for cloud computing. International Journal of Computer Applications, 55(3), 38-42. https://doi.org/10.5120/ i1738-2991

Shirahata, K., Sato, H., & Matsuoka, S., “Hybrid map task scheduling for GPU-based heterogeneous clusters,” IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp.733–740, December, 2010.

Shitara, A., Nakahama, T., Yamada, M., Kamata, T., Nishikawa, Y., Yoshimi, M., & Amano, H., “Vegeto: 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.

Sommer, L., Korinth, J., & Koch, A., “OpenMP device offloading to FPGA accelerators,” 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017), pp.201–205, July 2017.

Sterling, T., Anderson, M., & Broadowicz, M. (2018). High performance computing: Modern systems and practices. Morgan Kaufmann.

Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering, 12 (3), 66–73. https://doi.org/10.1109/MCSE.2010.69

Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., & Petersen, P., “Compiler support of the work queuing execution model for intel SMP architectures,” In Fourth European Workshop on OpenMP, September 2002.

Tanaka, Y., Yoshimi, M., & Miki, M. (2011). Evaluation of optimization method for fortran codes with GPU automatic parallelization compiler. IPSJ SIG Technical Report, 2011(9), 1–6.

Tomatsu, T., Hirotsu, T., Yoshimi, M., & Miki, M., “gpOT: Intelligent compiler for GPGPU using combinatorial optimization techniques,” The 7th Joint Symposium between Doshisha University and Chonnam National University, August, 2000.

Wang, C. C., Lin, Y. D., Wu, J. J., Lin, P. C., & Hwang, R. H. (2018). Toward optimal resource allocation of virtualized NVM datacenters for hierarchical datacenters. IEEE Transactions on Network and Service Management, 15(4), 1532–1544. https://doi.org/10.1109/TNSM.2018.2862422

Wieken, S., Springer, P., Terboven, C., & an Mey, D. (2012). OpenACC-first experiences with real-world applications. Euro-Par 2012 Parallel Processing, 859–870 https://faculty.kfupm.edu.sa/coe/mayej/pscoes501/Project-references/OpenACC/OpenACC%20First%20Experiences.pdf

Wolfe, M., “Implementing the PGI accelerator model,” ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43–50, March. 2010.

Wuhih, F., Stadler, R., & Lindgren, H., “Dynamic resource allocation with management objectives - implementation for an OpenStack cloud,” In Proceedings of Network and service management, 2012 8th international conference and 2012 workshop on systems virtualization management, pp.309–315, October. 2012.

Yamato, Y., Nishizawa, Y., Nagao, S., & Sato, K. (2015, September). Fast and reliable restoration method of virtual resources on openstack. IEEE Transactions on Cloud Computing 6(2), 572–583. https://doi.org/10.1109/TCC.2015.2481392x

Yamato, Y. (2015a, October). Automatic system test technology of virtual machine software patch on IaaS cloud. IEEE Transactions on Electrical and Electronic Engineering, 10(51), 165–167. https://doi.org/10.1002/tee.22179

Yamato, Y., “Use case study of HHD-SSD hybrid storage, distributed storage and HDD storage on openstack,” 19th International Database Engineering & Applications Symposium (IDEAS15), pp.228–229, July 2015a.

Yamato, Y. (2015c, February). Automatic verification technology of software patches for user virtual environments on IaaS cloud. Journal of Cloud Computing, 4(1). Springer. https://doi.org/10.1186/s13677-015-0028-6

Yamato, Y., Fukumoto, Y., & Kumazaki, H., “Analyzing machine noise for real time maintenance,” 2016 8th International Conference on Graphic and Image Processing (ICGIP 2016), October. 2016.

Yamato, Y., “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.

Yamato, Y., “Proposal of vital data analysis platform using wearable sensor,” 5th IIAE International Conference on Industrial Application Engineering 2017 (ICIAE2017), pp.138–143, March. 2017a.

Yamato, Y., Fukumoto, Y., & Kumazaki, H. (2017a, June). Proposal of shoplifting prevention service using image analysis and ERP check. IEEE Transactions on Electrical and Electronic Engineering, 12(51), 141–145. https://doi.org/10.1002/tee.22427

Yamato, Y., “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 2017b.

Yamato, Y., Fukumoto, Y., & Kumazaki, H., “Security camera movie and ERP data matching system to prevent theft,” IEEE Consumer Communications and Networking Conference (CCCN 2017), pp.1021–1022, January 2017.

Yamato, Y., Demizu, T., & Noguchi, H. (2018, September). Automatic GPU flooding technology for open IoT environment. IEEE Internet of Things Journal, 5(5), 3978–3990. https://doi.org/10.1109/JIOT.2018.2872545

Yamato, Y. (2019). Study of parallel processing area extraction and data transfer number reduction for automatic GPU offloading of IoT applications. Journal of Intelligent Information Systems, Springer 54(3), 567–584. https://doi.org/10.1007/s10844-019-00575-8

Yamato, Y., “Proposal of automatic offloading for function blocks of applications,” The 8th IIAE International Conference on Industrial Application Engineering 2020 (ICIAE 2020), pp.4–11, March, 2020.

Yamato, Y. (2021, April). Automatic offloading method of loop statements of software to FPGA. International Journal of Parallel, Emergent and Distributed Systems, Taylor and Francis, 36(5), 482–494. https://doi.org/10.1080/17445576.2021.1918020
