Title	Automatic Offloading Method of Loop Statements of Software to FPGA
Author(s)	Yoji Yamato
Citation	Journal title (Repository name etc.), Volume, Issue, Pages (Article number) etc. ・ジャーナル名（刊行物・サイト名）・巻号・ページ（その他論文番号等）： International Journal of Parallel, Emergent and Distributed Systems ・DOI (URL) https://doi.org/10.1080/17445760.2021.1916020 Publication Date: yyyy/mm/dd ・出版日：2021 年 4 月 19 日 Publisher ・出版者：Taylor and Francis
Declaration	This preprint is the Author Accepted Manuscript(AAM) of the above. ・本プレプリントは、上記論文の (受理済)著者最終稿(AAM) である。 All necessary permissions from the publisher have ・ジャーナル（出版者）から必要な許諾を been obtained / not been obtained [✓ 得ている] 得ていない
Notes	"This is an Accepted Manuscript of an article published by Taylor & Francis in International Journal of Parallel, Emergent and Distributed Systems, on 19th Apr., 2021, available online: https://doi.org/10.1080/17445760.2021.1916020."

ORIGINAL PAPER

Automatic Offloading Method of Loop Statements of Software to FPGA

Yoji Yamato[a]

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

ARTICLE HISTORY
Compiled January 24, 2021

ABSTRACT
With Moore's law coming to an end, using hardware other than central processing units (CPUs), such as more energy efficient field-programmable gate arrays (FPGAs), has recently been increasing. However, when using heterogeneous hardware other than CPUs, barriers of technical skills, such as in using Open Computing Language (OpenCL), are high. Therefore, I previously proposed environment-adaptive software that enables automatic conversion, configuration, and high-performance operation of once-written code according to the hardware to be placed. Offloading of some applications to graphics processing units (GPUs) was automated previously. In this paper, I proposed an automatic offloading method of appropriate target loop statements of applications as the first step in offloading to FPGAs. I evaluated the effectiveness of the proposed method by applied it to multiple applications.

KEYWORDS
Environment Adaptive Software; FPGA; Automatic Offloading; Performance

1. Introduction

Moore's Law is ending; thus, the semiconductor density of a central processing unit (CPU) cannot be expected to double in 1.5 years. Applications with heterogeneous hardware such as field programmable gate arrays (FPGAs) or graphics processing units (GPUs) are increasing. In particular, FPGAs have better power efficiency than CPUs. For example, Microsoft's search engine Bing tries to use FPGAs [1] to improve search efficiency and reduce data-center energy consumption. Amazon Web Services (AWS) provides FPGA and GPU instances using cloud technologies (e.g., [2]-[5]).

To obtain high offloading performance by effectively using heterogeneous hardware, programmers need to program and configure considering hardware specifications and have expert skills such as in using Open Computing Language (OpenCL) [6] and Compute Unified Device Architecture (CUDA) [7]. These barriers are high for many programmers.

Along with the progress in Internet of Things (IoT) technology ([8]-[15]), network-connected devices are increasing rapidly. Connected devices have reached several tens of billions and will reach trillions in 2030. There are various application areas of IoT

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

and many IoT applications are developed using service coordination technologies (e.g., [16][17]).

In IoT applications, knowledge of embedded software are sometimes required for detailed control of IoT devices. In many applications, one IoT gateway (GW) of small computers, such as Raspberry Pi, controls many IoT devices, but design studies for the deployed environments are needed because the resources of small computers are limited.

In summary, systems with heterogeneous hardware, such as FPGAs, GPUs and many IoT devices, are expected to increase; however, barriers are high to use such hardware effectively. To remove these barriers and use heterogeneous hardware easily and effectively, a platform is needed on which developers only write logics to be processed so that software can adapt to the deployed environments with heterogeneous hardware by automatic conversion and configuration.

Java, which appeared in 1995, caused a paradigm shift in environment adaptation that allows codes written once to run on other vendors' CPU machines. However, there is no guarantee of application performance at the porting destination. Therefore, I previously proposed environment-adaptive software, which executes once-written applications with high performance by automatic code conversion and configurations so that FPGAs, GPUs, IoT devices or others hardware can be effectively used on deployed environments. As elementary technology of environment-adaptive software, I also achieved automatic GPU offloading of some applications software [18][19]. In this paper, I propose a method for automatic offloading of appropriate loop statements of application software as the first step in offloading to FPGAs, which are energy efficient compared to CPUs. I implemented the proposed method and evaluated its effectiveness in offloading to FPGAs using several applications.

The rest of this paper is organized as follows. In Section 2, I review current heterogeneous hardware technologies. In Section 3, I present my proposed automatic FPGA offloading method for loop statements. In Section 4, I explain the implementation to verify the proposed method's effectiveness. In Section 5, I discuss the evaluation of the proposed method and discuss the results. I describe related work in Section 6 and summarize the paper in Section 7.

2. Current technologies

A typical example of environment-adaptive software is Java. Using a virtual execution environment, i.e., Java virtual machine, once-written Java code can be run on CPU machines of even different OSes or vendors without additional compiling (Write Once, Run Anywhere). However, it has not been considered whether high performance of application can be achieved at the porting destination, and developers' workload, such as debugging and performance tuning, at the porting destination is huge (Write Once, Debug Everywhere). If applications use heterogeneous hardware, such as FPGAs, this workload increases even more.

To control heterogeneous hardware uniformly, OpenCL specifications are used and its SDK has become widespread. For general purpose GPUs that use GPU parallel computation power not only for graphics processing (e.g., [20]) but also for other purposes, CUDA is a major environment. OpenCL and CUDA need not only C language extension grammars but also additional hardware-oriented descriptions such as memory copy between FPGA devices and CPUs. Because of these difficulties, there are few OpenCL or CUDA programmers.

Comparing to OpenCL, for easy use of heterogeneous hardware, there are technologies that specify parallel processing areas by specified directives, and compilers convert these codes into device-oriented codes based on specified directive meanings. Open Accelerators (OpenACC) is an example of a directive-based specification [21], and the Portland Group (PGI) compiler is an example of a directive-based compiler [22]. For example, users specify OpenACC directive "#pragma acc kernels" on Fortran/C/C++ codes to process them in parallel, and the PGI compiler checks the parallel processing possibility and outputs and deploys binary files to execute on CPUs and GPUs. For Java, IBM JDK supports offloading to GPUs based on a Java lambda expression [23].

In this way, OpenCL, CUDA, OpenACC and other technologies enable offloading to FPGAs or GPUs. Although offloading to FPGAs or GPUs can be done, high offloading performance of application is difficult to achieve. For example, there are automatic parallelization technologies, such as the Intel compiler [24], for many-core CPUs. They are used to extract possible areas of parallel processing such as for and while loop statements. However, naive parallel processing performance with FPGAs or GPUs is not high because of overheads of CPU and FPGA/GPU device memory data transfer. To achieve high offloading performance of application with FPGAs/GPUs, highly skilled programmers need to tune using OpenCL/CUDA or appropriate offloading areas need to be searched for.

Thus, it is difficult for programmers without OpenCL or CUDA skills to obtain high offloading performance of application by using FPGAs or GPUs. Moreover, if programmers use automatic parallelization technologies to achieve high offloading performance of application, it takes a long time of try and error for each loop statement is improved or not by parallelization.

3. Proposal of automatic offloading method of loop statements to FPGA

3.1. Flow of environment-adaptive software

To achieve software adaptation to environments, I previously proposed environment-adaptive software, the flow of which is shown in Figure 1 (the steps are listed below). This environment-adaptive software is executed with an environment-adaptation function, which is the main function, verification environment, production environment, test case database (DB), code pattern DB, facility resource DB.

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

In Steps 1-7, the processing flow for environment adaptation is carried out with code conversion, resource-amount adjustment, placement-location adjustment, and reconfiguration during operation. Here, operation verification is conducted using cloud automatic verification technologies such as [25][26]. However, it is also possible to extract only some of the steps of environment adaptation. For example, if we only want to convert code for an FPGA, we only need to conduct Steps 1-3.

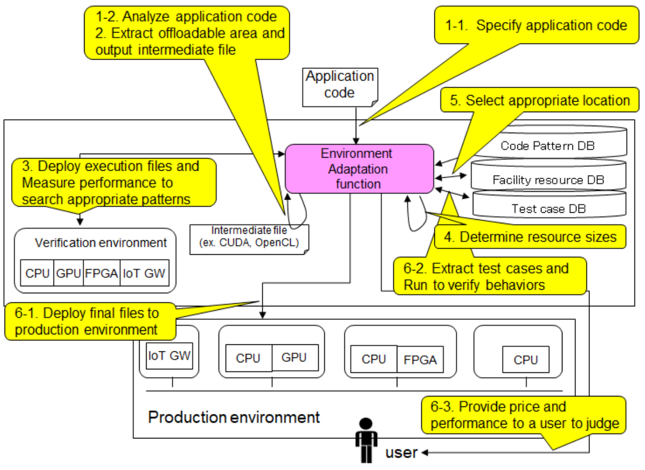

1-2. Analyze application code
2. Extract offloadable area and output intermediate file
1-1. Specify application code
Application code
5. Select appropriate location
Environment Adaptation function
Code Pattern DB
Facility resource DB
Test case DB
3. Deploy execution files and Measure performance to search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

Figure 1. Flow of environment-adaptive software

3.2. Considerations for automatic offloading to FPGA

For code analysis (Step 1), parse and analysis of application code is executed using a parsing tool such as Clang. Code analysis is difficult to generalize because it is necessary to consider devices to be offloaded. However, in this step, it is necessary to understand the structure of the source code such as loop statements, reference relations with the variables, and function blocks of specified processing or calling a specified function library such as fast Fourier transform (FFT) processing. It is difficult for a machine to detect function blocks automatically; thus, I use similarity-detection tools such as Deckard to judge code similarity. Clang is a tool for C/C++; thus, it is necessary to choose a tool suited to the language to be parsed.

In Steps 2-3, it is necessary to consider extraction and search according to the offload destination, such as an FPGA, GPU, or IoT GW; thus, it is assumed that processing functions are plugged in for each offloading destination. It is generally difficult to automatically detect the configuration that will enable maximum offloading performance at one time; thus, we repeatedly try the offload patterns in the verification environment several times to detect an appropriate offload pattern by using an evolutionary computation method. I achieved automatic offloading to GPUs for some applications in a previous study [18]. Therefore, this paper is focused on offloading of application software to FPGAs.

Applications that users want to offload are various. However, typical processing that requires a large amount of computation time has many loops such as image analysis for movie processing and machine-learning processing for analyzing sensor data. Therefore, the first target to offloading to FPGAs is also loop statements.

However, high offloading performance requires pipeline and/or parallel processing of appropriate areas. In particular, because of memory data transfer between a CPU and FPGA, offloading performance sometimes does not improve unless the data size or number of loops is sufficiently large. Because the clock speed of an FPGA is slower than that of a CPU, non-suitable computation for FPGA does not improve offloading performance.

Moreover, combinations of individual loop statements that can be accelerated by pipeline processing sometimes does not enable maximum offloading performance configurations depending on memory process status and timing of data transfers. For example, when #1, #3, and #5 loop statements can be accelerated by pipeline processing compared with with CPU processing in 5 loop statements, a three-pipeline combination of #1, #3, and #5 is not always the fastest configuration.

For offloading to GPUs, OpenACC only specifies the #pragma acc kernels directive so that specified loop statements can be executed on a GPU or CUDA can describe more detail control. To control an FPGA, OpenCL can be used to describe detailed control similar to using CUDA, and high level synthesis (HLS) tools can specify more abstract control similar to using OpenACC. The following ten description steps are needed for OpenCL. (1. Prepare devices. 2. Prepare kernels. 3. Allocate devices memory. 4. Transfer data from hosts to devices. 5. Configure variables of kernel functions. 6. Execute kernel functions. 7. Transfer data from devices to hosts. 8. Release devices memory. 9. Release kernels. 10. Release other objects such as devices.) Each vendor HLS has a different specification, but it can control an FPGA more abstractly. For example, Xilinx Vivado HLS specifies FPGA processing by #pragma HLS PIPELINE, #pragma HLS UNROLL, and so on, similar to using OpenACC. Intel FPGA SDK for OpenCL recognizes not only the OpenCL standard but also other extensions such as #pragma directives.

To extract appropriate offloading areas from general CPU programs automatically, I previously checked all loop statements to determine whether they can be processed then executed performance verification repeatedly in the verification environment using the genetic algorithm for processable loop statements to search for the appropriate offload pattern [18]. However, code compiling to an FPGA usually takes several hours, and performance measurements of many patterns, such as those in my previous study [18], are difficult. Therefore, it is assumed that the number of performance measurements will be reduced after narrowing down the patterns for offloading performance measurements with actual FPGAs.

3.3. Proposed automatic offloading method

Based on above considerations, I propose an offloading method of loop statements to FPGA.

The method first parses source codes to be offloaded. It then grasps the loop statements and variable information according to the language of the source codes.

Next, a process to narrow down candidates is performed for whether or not to try offloading loop statements to FPGAs. Arithmetic intensity can be one indicator of whether a loop statement has an offloading effect. Arithmetic intensity increases when the number of calculations are large and decreases when the number of data accesses is large. Processing with high arithmetic intensity is heavy for the processor and takes time. Therefore, an arithmetic intensity analysis tool analyzes the arithmetic intensity of a loop statement and narrows down the high-intensity loop statements as offloading candidates. However, arithmetic intensity ignores loop numbers; thus, we may also check loop numbers by using profiling tools.

Even if there is a high arithmetic intensity and a large number of loops in a loop statement, it is problem that a large amount of FPGA resources are consumed. Therefore, we move on to estimating the amount of resources when offloading loop statements to FPGAs. When compiling to an FPGA, a program is converted from a high-level language, such as OpenCL, to a hardware-level language, such as HDL, and actual wiring processing is carried out based on the hardware-level language. At this time, wiring processing takes much time, but it takes only a minute to extract the HDL as the intermediate state. Since resources, such as flip flop and look up table, used in FPGAs can be estimated at the HDL level, the amount of resources used can be determined in a short time even if compiling is not completed. Since the target loop statement is converted into a high-level language and the resource amount is calculated from OpenCL, the arithmetic intensity, loop number, and resource amount when the loop is offloaded are determined. With this method, the loop statements with high resource efficiency are further narrowed down as offload candidates. High resource efficiency means that arithmetic intensity/resource amount or arithmetic intensity*loop number/resource amount is high.

Two processes are required to convert a loop statement into a high-level language. One is to divide a CPU processing program into a kernel (FPGA) program and host (CPU) program based on the syntax of the high-level language. The other is to include techniques for accelerating loop statements. There are techniques for accelerating loop statements using FPGAs such as local memory cache, stream processing, multiple instantiation, loop statement expansion, integration of nested loop statements, and memory interleaving. Depending on the loop statement, these may not have an absolute effect but are often used for accelerating loop statements.

Next, because some loop statements with high resource efficiency are selected, several offloading patterns that measure the offloading performance using these loop statements are generated. There are several types for accelerating loop statements in FPGAs, one type is accelerating them by concentrating the amount of FPGA resources to one loop statement, and the other is accelerating them by distributing FPGA resources to multiple loop statements. The proposed method generates patterns for the selected loop statements, compiles them to run on an FPGA, and measures the offloading performance. The method then generates combination patterns for individual loop statements that can be accelerated, and measures the offloading performance in the same manner.

Finally, a high-speed pattern is selected as the solution among the multiple patterns whose performance has been measured in the verification environment.

Therefore, the method focuses on loop statements with high arithmetic intensity, loop number, and resource efficiency, creates offloading patterns, and searches for patterns at high speed through actual measurements in the verification environment (Figure 2).

4. Implementation

In this section, I explain the implementation of the proposed method. To confirm the method's effectiveness, I used C/C++ language applications for offloading applications and Intel PAC with an Intel Arria10 GX FPGA for FPGA. I also conducted a compiling on DELL EMC PowerEdge R740 with Intel Xeon Bronze 3104 /1.70 GHz CPU and 32GB RDIMM DDR4-2666 *2 RAM.

To control the FPGA, I used Intel Acceleration Stack Version 1.2 with Intel FPGA SDK for OpenCL 17.1.1 and Quartus Prime Version 17.1.1. Intel FPGA SDK for OpenCL is a HLS tool that compiles #pragma directives in addition to the standard OpenCL. It compiles OpenCL code that describes the kernel program processed by the FPGA and the host program processed by the CPU, outputs information such as the amount of resources, and performs FPGA wiring to operate the code using the FPGA. Even a small program of about 100 lines takes about 3 hours to be able to

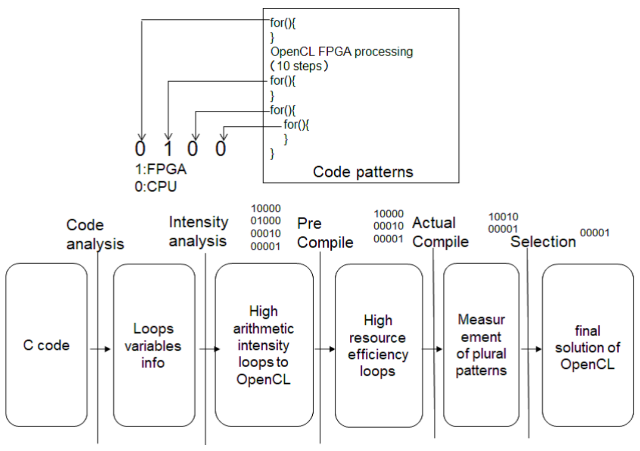


Figure 2. Proposed automatic offloading method of loop statements to FPGA

run on an actual FPGA, but an error occurs early when the amount of resources is exceeded of resource size of Arria10 FX FPGA. With OpenCL code that cannot be processed by an FPGA, an error is output after several hours.

I now explain the outline of the implementation in which I used Python 2.7.

When a C/C++ application is specified, the implementation analyzes C/C++ code, detects "for" loop statements, and detects the program structure such as variable data used in the "for" statements. For parsing, the python program uses parsing libraries of LLVM/Clang 6.0 libClang python binding.

Next, the implementation executes the arithmetic intensity analysis tool to determine the possibility of the offloading effect of each loop statement to the FPGA and obtain an index of arithmetic intensity. Only the number of top A loop statements with the highest arithmetic intensity are targeted. ROSE framework 0.9 is used for arithmetic intensity analysis. Although this framework has many functions, it can be used for arithmetic intensity analysis only. This implementation also executes the profiling tool to obtain the repeat number of loops of a loop statement. Gcov is used for profiling in this implementation; Gprof can be also used. The implementation narrows down the number of target loop statements with high arithmetic intensity and high loop numbers.

The implementation then generates offloading OpenCL code for candidate loop statements to be offloaded to the FPGA. The OpenCL code is divided into the loop statement as an FPGA kernel and the rest as a CPU host program. When the FPGA kernel code is generated, the loop sentence is expanded by a certain number B as an acceleration technique. The loop-statement expansion process increases the amount of resources, but is effective for accelerating some loop statements. The number of expansions is limited to a fixed B, and the amount of resources does not become enormous.

The implementation then pre-compiles the A OpenCL codes using Intel FPGA SDK for OpenCL and calculates the amount of resources, such as flip flop and look up table, to be used. The amount of resources used is displayed as a percentage of the total resource amount. The resource efficiency of each loop statement is calculated from the arithmetic intensity and resource amount. For example, a loop statement with an arithmetic intensity of 10 and resource amount of 0.5 has a resource efficiency of 10/0.5=20. A loop statement with a arithmetic intensity of 3 and a resource amount of 0.3 has a resource efficiency of 3/0.3=10; the former is high. In loop statements, the implementation selects C OpenCL codes with high resource efficiency.

Next, the implementation generates patterns to be measured using selected C loop statements as candidates. For example, if #1, #3, and #5 loops are highly resource efficient, the implementation generates and compiles OpenCL patterns with #1 offloaded, #3 offloaded, and #5 offloaded. In the first measurement, the implementation generates patterns within D and conducts performance measurements on a server with the FPGA in the verification environment. For offloading performance measurement, the sample processing specified by the application to be accelerated is conducted. For example, in the case of an application of Fourier transform, the offloading performance is measured using transform processing with sample data as a benchmark. If #1 and #3 offloading can be accelerated, the implementation generates a pattern with both #1 and #3 offloaded in the second measurement. Note that when generating a combination of individual loops, the amount of resources is also increased, so if the combination pattern does not fit within the upper limit of resource size, it is not generated.

The implementation finally selects the maximum offloading performance pattern from several measured patterns.

5. Evaluation

5.1. Evaluation method

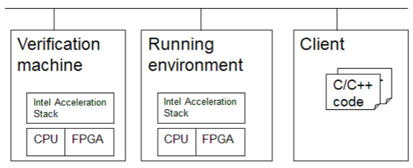
5.1.1. Evaluated applications

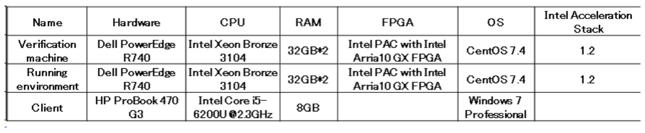
I evaluated three applications, signal processing of a time-domain finite-impulse response filter, image processing of magnetic resonance imaging Q (MRI-Q), and matrix calculation of symmetric matrix-multiply (Symm).

The time-domain finite-impulse response filter performs processing in a finite time on the output when an impulse function is input to a system. There are various implementations of this filter. I used [27] C code and sample tests with it for offloading performance measurement. When considering applications that transfer signal data from IoT devices over the network, to reduce network costs, it is assumed that signal processing such as filters are conducted on IoT devices sides. Thus, I think the proposed method has a wide range of applications regarding signal processing.

MRI-Q [28] computes a matrix Q, representing the scanner configuration for calibration, used in 3D MRI reconstruction algorithms in non-Cartesian space. In an IoT environment, image processing is often necessary for automatic monitoring from camera videos, and performance enhancements are requested in many cases. During application performance measurement, MRI-Q executes 3D MRI image processing to measure processing time using 64*64*64 size sample data.

Symm [29] is a benchmark of Polybench for symmetrix matrix multiply calculation. Matrix multiply is frequently used for manual GPU or FPGA acceleration using CUDA or OpenCL; thus, I used it to confirm whether matrix calculation can be accelerated with the proposed method. During application performance measurement, Symm executes matrix multiply calculation to measure processing time using 1,024*1,024 matrix size sample data.





Name	Hardware	CPU	RAM	FPGA	OS	Intel Acceleration Stack
Verification machine	Dell PowerEdge R740	Intel Xeon Bronze 3104	32GB*2	Intel PAC with Intel Arria10 GX FPGA	CentOS 7.4	1.2
Running environment	Dell PowerEdge R740	Intel Xeon Bronze 3104	32GB*2	Intel PAC with Intel Arria10 GX FPGA	CentOS 7.4	1.2
Client	HP ProBook 470 G3	Intel Core i5-6200U @2.3GHz	8GB		Windows 7 Professional	

Figure 3. Experimental environment

5.1.2. Experimental conditions

For automatic offloading to an FPGA, I did not conduct many performance measurements, as in a previous study on automatic offloading to a GPU [19]. In the verification of the proposed method, the offloading performance measurement results of the sample tests with multiple offload patterns in the verification environment were recorded together with intermediate information such as arithmetic intensity, resource efficiency, and HDL-related information during compilation. Through offloading performance measurement, the highest performance pattern was the search solution. The performance of the solution was compared by to performance of all CPU processing.

The experimental conditions were as follows.

Offloading target number (number of loop statements): 6 for the time-domain finite-impulse response filter, 16 for MRI-Q and 9 for Symm.

Narrowing down using arithmetic intensity: Narrow down to the top five loop statements of arithmetic intensity.

Number of loop-statement expansions: 1. Expansion processing and multiple instantiations may accelerate offloading performance but the amount of FPGA resources are increased. Thus, in this verification, I confirmed the effectiveness of offloading to an FPGA with OpenCL without expansions.

Narrowing down using resource efficiency: Narrow down to the top three loop statements in resource efficiency analysis. The implementation selects the top three loop statements with high arithmetic intensity/resource amount in this verification.

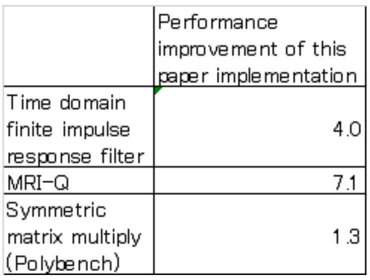
Number of measured offload patterns: 4. In the first measurement, the top three loop statement offload patterns were measured. Then, the second measurement was measured with the combination pattern of two loop statement offloads that were high performance at the first measurement.

5.1.3. Experimental environment

I used physical machines with Intel PAC with an Intel Arria10 GX FPGA for offloading verification and Intel Acceleration Stack Version 1.2 for FPGA control. Figure 3 shows the experimental environment and specifications. A client note PC specifies C/C++ application codes, codes are tuned by trial and error on a verification machine, and final codes are deployed in a running environment for users.

5.2. Performance results

I confirmed automatic acceleration of several applications to FPGAs.



	Performance improvement of this paper implementation
Time domain finite impulse response filter	4.0
MRI-Q	7.1
Symmetric matrix multiply (Polybench)	1.3

Figure 4. Application performance improvement with proposed method

Figure 4 shows the measurement results of how much application performance improved with the proposed automatic offloading method. Figure 4 shows how many times the performance of the final solution was higher than that of all CPU processing. The proposed method improved application performance by 4.0 times for the time domain finite impulse response filter. For MRI-Q, the proposed method improved application performance by 7.1 times, and for Symm, it improved application performance by 1.3 times. It took about half a day to automatically verify four offload patterns because it takes about 3 hours to compile one offload pattern.

5.3. Discussion

My previous study [19] confirmed that application performance improved 3-5 times for large applications, such as Darknet, which has more than 100 loop statements and NAS.FT. In this study, the proposed method improved offloading performance by 4 times for signal processing and 7 times for image processing.

I now discuss cost effectiveness. FPGA boards, such as Intel Arria, cost about 3,000 USD. Therefore, FPGAs costs about twice as much as hardware with CPUs. In data centers, hardware, development, and verification costs of a system, such as a cloud system, is generally about 1/3 the total cost; electricity and operation/maintenance costs are more than 1/3, and other expenses, such as service orders, is the final 1/3. In the AWS case, a FPGA instance with one FPGA costs about 1188 USD/month, which is the same as hosting a general dedicated server. Therefore, I think improving offloading performance with several times in loop statements of applications that take much time will have a sufficiently positive cost effect even though the hardware price doubled.

I will now discuss time to start production services. Unlike offloading to GPUs, FPGAs takes a long time to compile, so the solution search time varies greatly depending on the number of performance measurements. From this evaluation, by setting the number of performance measurements to about 4, a certain amount of acceleration can be achieved with half a day of verification. When service providers provide production IoT or cloud services, we provide the first day for free and try to accelerate application performance in the verification environment during the first day. From the second day, we provide production service using FPGAs. Therefore, I think that one-day automatic offloading is acceptable.

Next, I compare manual and automatic FPGA offloading. When manually accelerating application performance with FPGAs, programmers analyze and design which loop statement is processed on FPGAs, use OpenCL or HDL, and verify offloading performance by compiling on an FPGA. However, since there is no fixed way of accelerating application performance using FPGAs by applying a fixed method, trial and error is required currently. When my co-worker of NTT laboratories manually accelerates the finite impulse response, it takes about 20 days because of several trials and errors. Therefore, one or half day automation is effective with regard to programmer workload.

To search for an offloading pattern within a shorter time, we can process each pattern-performance measurement on multiple machines in parallel. In addition, parameter tuning, such as threshold of arithmetic intensity, is also needed.

Even though the programmer workload is low, there are some applications in which automatic improvement in offloading performance with the proposed method may not be sufficient compared to manual improvement. There are two directions for further improving offloading performance.

First, we apply several techniques of improving offloading performance when creating OpenCL descriptions. When offloading a loop statement using OpenCL, there are common techniques for accelerating the local memory cache, stream processing, multiple instantiation, loop-statement expansion, integration of nested loop statements, memory interleaving, and so on. For example, multiple instances may be accelerated depending on the amount of resources, and it is possible to set an appropriate resource size in consideration of the amount of available resources.

Second, we offload in units of larger function blocks rather than in each loop statement. For FPGA acceleration, it is often the case that an algorithm for a CPU is changed to an algorithm suitable for hardware processing. For example, in matrix multiplication, there is an example in which data A are read in the row direction, data B are read in the column direction, and the local memory with limited capacity is used efficiently. In addition to loop-statement offloading, I am also studying offloading to the existing implemented FPGA IP core in units of large function blocks such as matrix multiplication and FFT calculation.

6. Related work

Wuhib et al. studied resource management and effective allocation [30] on the OpenStack [31] cloud. The proposed method is effective for network-wide resource management and allocation including the cloud, but it focuses on appropriate offloading on heterogeneous hardware servers. I previously proposed a method of selecting appropriate servers from heterogeneous hardware servers [32] based on cloud servers performances, but the proposed method for this paper can offload logic parts of applications automatically and is novel for this reason.

There have been many studies on FPGA offloading [33][34][35][36]. Liu et al. [33] proposed a method of offloading nested loops to an FPGA and found that nested loops can be offloaded with an additional 20 minutes of manual work. Alias et al. [34] proposed a method with which an HLS tool configures an FPGA by specifying C language code, loop tiling, and so on when using Altera HLS C2H. The method proposed by Sommer et al. [35] can be used to interpret OpenMP code and perform FPGA offloading. Putnum et al. [36] used a CPU-FPGA hybrid machine to accelerate a program with a slightly modified standard C language. These methods require manually adding instructions such as which parts to parallelize using OpenMP or other specifications. There have been few studies on automatically offloading existing codes to FPGAs.

To control FPGAs, development tools of OpenCL for FPGAs are provided by Altera and Xilinx. Automatically offloading not only loop statements but also various application logics is difficult for machines because it is similar to recognizing code meaning. On the other hand, FPGAs have been accelerated on the basis of programmers' expertise, such as FFT acceleration. Therefore, to use existing technique, there is a method [37] with which providers prepare well-known offloading patterns in a DB. When applications have well-known processing such as FFT, the processing is changed to call an FPGA by extracting OpenCL of offloading patterns from the DB.

Generally, CUDA and OpenCL control intra-node parallel processing, and Message Passing Interface (MPI) controls inter-node or multi-node parallel processing. However, MPI also requires high technical skills in parallel processing. Thus, MPI concealment technology has been developed that virtualizes devices of outer nodes as local devices and enables devices of outer nodes to be controlled by only OpenCL [38]. When we select multi-nodes for offloading destinations, we plan to use this MPI concealment technology.

Even if an extraction of an offloading area is appropriate, offloading performance of application may not be high when the resource balance of a CPU and devices is not appropriate. For example, a CPU takes 100 seconds and FPGA 1 second when one task is processed, so a CPU slows processing. Shirahata et al. [39] attempted to improve total offloading performance by distributing Map tasks with the same execution times of a CPU and GPU in MapReduce processing. Referring to that study, I will study how to deploy functions on appropriate locations and resource amounts to avoid bottlenecks of any hardware.

There have been many studies on using FPGAs to achieve high offloading performance for many high arithmetic intensity applications such as fluid calculation and game algorithm processing. Automatic offloading is common for automatic parallelization compilers like the Intel compiler for multicore CPUs. However, there have not been any studies that involved automatic offloading performance evaluations with several candidate offloading areas and offloading general large applications to FPGAs.

7. Conclusion

I previously proposed environment-adaptive software that adapts applications to the deployed environments to effectively use heterogeneous hardware such as FPGAs and GPUs. As an elementary technology of this, I proposed and evaluated an automatic FPGA offloading method for loop statements of software codes. It is often said that FPGAs are more energy efficient than CPUs.

The proposed method is the same as the offloading method to GPUs [19] until loop statement detection by analyzing the source code. However, it takes a long time to compile to an actual FPGA. To cope with long time compile, the loop statements of offloading candidates are narrowed down before the actual measurement trials are conducted. To detect loop statements, a loop statement having a high arithmetic intensity is extracted using an arithmetic intensity analysis tool. Then, pre-compile is carried out to conduct offloading to an FPGA such as expansion processing for loop statements with high arithmetic intensity. This finds a loop statement with high resource efficiency and high arithmetic intensity. For the narrowed-down loop statements, our method generates OpenCL codes that offload each loop statement or a combination of those loop statements, compiles them on the FPGA, measures the offloading performance of application, and selects the highest-performance OpenCL code as the solution.

I evaluated the proposed method to automatically offload loops statements of several applications to an FPGA. In the future, I will evaluate the energy efficiency of the proposed method and study not only loop-statement but also function-block offloading such as FFT functions.

Disclosure statement

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

References

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

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

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

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

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

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

[7] J. Sanders, E. Kandrot, "CUDA by example : an introduction to general-purpose GPU programming," Addison-Wesley, ISBN-0131387685, 2011

[8] M. Hermann, T. Pentek and B. Otto, "Design Principles for Industrie 4.0 Scenarios," Working Draft, Rechnische Universitat Dortmund. 2015, http://www.snom.mb.tu-dortmund.de/cms/de/forschung/Arbeitsberichte/Design-Principles-for-Industrie-4_0-Scenarios.pdf

[9] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Predictive Maintenance Platform with Sound Stream Analysis in Edges," Journal of Information Processing, Vol.25, pp.317-320, Apr. 2017.

[10] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Proposal of Shoplifting Prevention Service Using Image Analysis and ERP Check," IEEJ Transactions on Electrical and Electronic Engineering, Vol.12, Issue.S1, pp.141-145, June 2017.

[11] Y. Yamato, "Proposal of Vital Data Analysis Platform using Wearable Sensor," 5th IIAE International Conference on Industrial Application Engineering 2017 (ICIAE2017), pp.138-143, Mar. 2017.

[12] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Security Camera Movie and ERP Data Matching System to Prevent Theft," IEEE Consumer Communications and Networking Conference (CCNC 2017), pp.1021-1022, Jan. 2017.

[13] Y. Yamato, "Experiments of posture estimation on vehicles using wearable acceleration sensors," The 3rd IEEE International Conference on Big Data Security on Cloud (BigDataSecurity 2017), pp.14-17, May 2017.

[14] Y. Yamato, Y. Fukumoto and H. Kumazaki, "Analyzing Machine Noise for Real Time Maintenance," 2016 8th International Conference on Graphic and Image Processing (ICGIP 2016), Oct. 2016.

[15] Y. Yamato, H. Kumazaki and Y. Fukumoto, "Proposal of Lambda Architecture Adoption for Real Time Predictive Maintenance," 2016 Fourth International Symposium on Computing and Networking (CANDAR 2016), pp.713-715, Nov. 2016.

[16] Y. Yokohata, Y. Yamato, M. Takemoto and H. Sunaga, "Service Composition Architecture for Programmability and Flexibility in Ubiquitous Communication Networks," IEEE International Symposium on Applications and the Internet Workshops (SAINTW'06), pp.142-145, Jan. 2006.

[17] Y. Yamato, "Ubiquitous Service Composition Technology for Ubiquitous Network Environments," IPSJ Journal, Vol.48, No.2, pp.562-577, Feb. 2007.

[18] Y. Yamato, T. Demizu, H. Noguchi and M. Kataoka, "Automatic GPU Offloading Technology for Open IoT Environment," IEEE Internet of Things Journal, DOI: 10.1109/JIOT.2018.2872545, Sep. 2018.

[19] Y. Yamato, "Study of parallel processing area extraction and data transfer number reduction for automatic GPU offloading of IoT applications," Journal of Intelligent Information Systems, Springer, DOI:10.1007/s10844-019-00575-8, 2019.

[20] J. Fung and M. Steve, "Computer vision signal processing on graphics processing units," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 5, pp.93-96, 2004.

[21] S. Wienke, P. Springer, C. Terboven and D. an Mey, "OpenACC-first experiences with real-world applications," Euro-Par 2012 Parallel Processing, pp.859-870, 2012.

[22] M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.

[23] K. Ishizaki, "Transparent GPU exploitation for Java," The Fourth International Symposium on Computing and Networking (CANDAR 2016), Nov. 2016.

[24] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah and P. Petersen, "Compiler support of the workqueuing execution model for Intel SMP architectures," In Fourth European Workshop on OpenMP, Sep. 2002.

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

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

[27] Time domain finite impulse response filter web site, http://www.omgwiki.org/hpec/files/hpec-challenge/tdfir.html

[28] MRI-Q website, http://impact.crhc.illinois.edu/parboil/parboil.aspx

[29] Polybench symm website, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

[30] F. Wuhib, R. Stadler, and H. Lindgren, "Dynamic resource allocation with management objectives - Implementation for an OpenStack cloud," In Proceedings of Network and service management, 2012 8th international conference and 2012 workshop on systems virtualiztion management, pp.309-315, Oct. 2012.

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

[32] Y. Yamato, "Server Selection, Configuration and Reconfiguration Technology for IaaS Cloud with Multiple Server Types," Journal of Network and Systems Management, Springer, DOI: 10.1007/s10922-017-9418-z, Aug. 2017.

[33] Cheng Liu, Ho-Cheung Ng and Hayden Kwok-Hay So, "Automatic nested loop acceleration on fpgas using soft CGRA overlay," Second International Workshop on FPGAs for

Software Programmers (FSP 2015), 2015.

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

[35] L. Sommer, J. Korinth and A. Koch, "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.

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

[37] Y. Yamato, "Optimum Application Deployment Technology for Heterogeneous IaaS Cloud," Journal of Information Processing, Vol.25, No.1, pp.56-58, Jan. 2017.

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

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