Noname manuscript No.
(will be inserted by the editor)

Study and evaluation of many-core CPU offloading for computing processes in environment adaptive software

Yoji Yamato

Received: date / Accepted: date

Abstract In recent years, the number of applications using diverse hardware such as FPGAs and GPUs has been increasing. However, utilizing these for high performance requires knowledge of hardware characteristics, which is quite difficult. Against this background, we propose environment-adaptive software that automatically converts and configures software code written by ordinary programmers for small-core CPUs according to the deployment environment, resulting in high-performance processing. This paper focuses on automatic offloading to many-core CPUs according to the computation type of processing, such as block matrix calculations and trivially parallel processing. We semantically analyze the existing application to be offloaded using an abstract syntax tree with pattern matching to determine whether the computation type has a replaceable OpenMP. If OpenMP is found, we replace it with that OpenMP and confirm the performance improvement. We confirm the effectiveness of the proposed method for automatic offloading by measuring processing time using an AMD Ryzen Threadripper 3995WX 64-core CPU.

Keywords Environment Adaptive Software · Automatic Offloading · Many-Core CPU · Calculation Type · Pattern Matching

1 Introduction

In recent years, it has been said that Moore's Law, which predicted the increasing integration density of CPUs (Central Processing Units), is no longer applicable. As a result, applications are now using a variety of hardware, including not only the commonly used few-core CPUs, but also many-core CPUs

Yoji Yamato
NTT Network Service Systems Labs., NTT, Inc., 3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585 Japan
Tel.: +81-422-59-5744
Fax: +81-422-60-7420
E-mail: yoji.yamato@ntt.com

with 30 or more cores, Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), quantum computers, and IoT devices. Amazon offers not only VMs (Virtual Machines) with few cores, but also VMs with multi-core CPUs, FPGAs, and GPUs on the cloud (e.g., [1]). [3] Microsoft has described the use of FPGAs for search and also offers a variety of VMs on the cloud. Furthermore, an increasing number of IoT services are being provided on the cloud using IoT PFs and connected IoT devices (e.g., [4]).

However, to utilize diverse hardware at high performance, programs that take advantage of the hardware's characteristics are required. This often requires the use of programming languages such as OpenMP (Open Multi-Processing) [5] for many-core CPUs, OpenCL (Open Computing Language) [6] for FPGAs, and CUDA (Compute Unified Device Architecture) [7] for GPUs. This makes it challenging for many programmers with expertise in scripting languages like Python and PHP.

GPUs are currently used in most AI, and their power consumption is a major issue. Power characteristics vary significantly depending on specific products and workloads. In general, both GPUs and many-core CPUs tend to have higher power consumption than few-core CPUs, and some examples are similar levels. Therefore, if a many-core CPU is faster than a GPU for certain type computations, offloading to a many-core CPU can be meaningful in terms of power savings. However, high-performance processing on many-core CPUs requires skills such as OpenMP. Therefore, a platform is needed that automatically converts and configures programs written in the same way as for a standard few-core CPU to suit the operating environment (many-core CPU, FPGA, GPU, etc.) to reduce the complexity and enable high-performance use of diverse hardware.

To address this issue, we have proposed the concept of adaptive software, which automatically converts existing programs for FPGAs or GPUs to enable high-performance use in the operating environment, thereby accelerating application processing. Furthermore, as a core technology for adaptive software, we have proposed and evaluated methods for automatically offloading loop statements in existing programs to many-core CPUs, FPGAs, GPUs, etc. [8]-[21].

However, to date, we have mainly examined automatic offloading of loop statements to many-core CPUs. While this method can achieve some speed improvement, it does not achieve the same speed as manually creating OpenMP algorithms tailored to the computation type.

This paper focuses on offloading to many-core CPUs according to computation type, such as block matrix calculations and trivially parallel processing. The existing program to be offloaded is semantically analyzed using an abstract syntax tree with pattern matching to determine whether a suitable OpenMP alternative exists for the computation type. If no suitable OpenMP alternative is available, a conventional method for accelerating loop statements is attempted. If a suitable OpenMP alternative is found, the accelerated OpenMP alternative is substituted. The proposed method's automatic

offloading capability is confirmed by measuring processing time on an AMD Ryzen Threadripper 3995WX [22] 64-core many-core CPU.

This paper is structured as follows: Section 2 provides an overview of existing technologies and describes the author's previously proposed environment-adaptive software. Section 3 outlines previously proposed methods, such as many-core CPU offloading of loop statements, and proposes an automatic offloading method to many-core CPUs using pattern matching based on the computation type. Section 4 describes the implementation tools and an overview of the implementation. Section 5 demonstrates the proposed method through many-core CPU offloading of two applications. Section 6 compares it with related research. Section 7 concludes.

2 Exisiting technology

2.1 Commercial technology

OpenCL is a standard specification for handling a wide variety of hardware, including many-core CPUs, FPGAs, and GPUs, and various companies offer OpenCL interpretive execution tools. OpenCL is a software specification that extends the C language, and its implementation hurdles are high (explicitly describing the release, movement, and duplication of memory data between the device kernel and the host).

Unlike OpenCL, some other specifications use directives to easily use a wide variety of hardware, specifying the parts of the program that perform specific processing and creating binary files according to the directives. For example, OpenMP is a specification for computing using multiple cores, such as many-core CPUs. OpenMP is a standard API for developing shared-memory, multithreaded parallel application software in a parallel computing environment. By adding the #pragma omp directive to your code, the processing corresponding to that directive is performed via an OpenMP interpretive execution tool. Examples of OpenMP interpretive execution tools include gcc[23]. OpenACC[24] is a specification that uses the #pragma directive to direct GPU processing, and the PGI compiler[25] (currently nvc) is an OpenACC interpreter and execution tool.

Processing using many-core CPUs is now possible using OpenCL, OpenMP, and other technologies. However, even if processing is possible, there are still barriers to achieving high speed. For example, the Intel compiler[26] is available to utilize many-core CPUs. This parallelizes the parallelizable portions of loop statements such as for statements. However, if data is not used efficiently, parallel processing of loop statements will not improve speed. Furthermore, many-core CPUs often use pipeline processing to take advantage of hardware characteristics to achieve efficient processing. OpenCL and OpenMP experts tune the algorithms and repeatedly run the tools to create appropriate OpenCL and OpenMP. Aiming to automate loop offloading, the author proposes a method to convert loop offloading patterns into OpenMP, and then

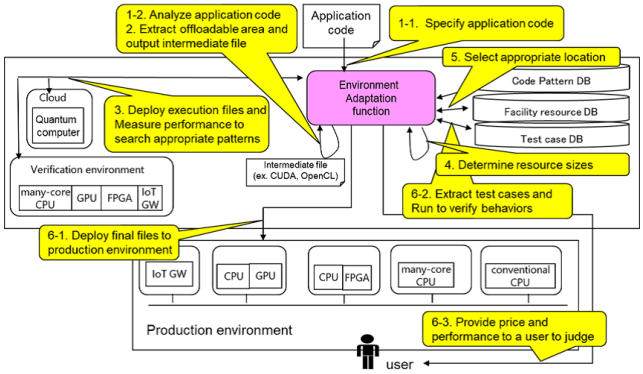


Fig. 1 Processing of environment-adaptive software

use a genetic algorithm to determine whether or not loops should be offloaded. Through repeated measurements in a verification environment, high-speed offloading patterns are gradually discovered.

2.2 Outline of environment-adaptive software

In Figure 1, we propose a seven-step process for adaptive software. The adaptive software process centers around the adaptive functionality provided by cloud service providers, linking the verification environment, production environment, code pattern database, test case database, and facility resource database.

Step 1: Analyze user-provided code:

Step 2: Extract offloadable parts:

Step 3: Search for appropriate offloadable parts:

Step 4: Adjust commercial resource amounts:

Step 5: Adjust commercial deployment locations:

Step 6: Commercial binary file deployment and verification:

Step 7: Reconfiguration during commercial operation:

In Steps 1-6, as part of the pre-operation process, we first analyze the user-provided code, conduct repeated performance measurement tests in the verification environment, convert it into appropriate code, determine the resource amounts and deployment locations, and verify proper operation. In Step 7, as part of the post-operation process, we analyze trends in actual usage data and reconfigure the production configuration if it is clear that changes to the operating code, resource amounts, deployment locations, or other aspects of the production configuration are appropriate.

2.3 The purpose of this paper

The challenges of this paper are outlined below. Accelerating applications using diverse hardware is typically achieved through manual modification by experts. I proposed the concept of environment-adaptive software and have implemented automatic offloading to many-core CPUs, FPGAs, GPUs, and other technologies. However, until now, automation has primarily focused on offloading individual for statements. As a result, while it is possible to achieve speedups of around ten times faster than with a few-core CPU, this does not match the speedup achieved by manually creating OpenMP, which takes into account the hardware and algorithms tailored to the type of calculation. This paper focuses on automatic offloading to many-core CPUs for calculations such as block matrix calculations and trivially parallel processing, depending on the type of calculation. By using pattern matching to replace the implementation with one packed with existing know-how depending on the type of calculation, automatic speedup is achieved taking the algorithm into account. The effectiveness of the proposed method is demonstrated on an actual many-core CPU, the AMD Ryzen Threadripper PRO 3995WX, a 64-core machine.

3 Offloading computational processing to many-core CPUs based on computation type

In this section, we propose offloading computational processing to many-core CPUs according to the computation type. Section 3.1 reviews the automatic offloading method for loop statements to many-core CPUs proposed in a previous paper. Section 3.2 discusses ideas for offloading at a larger granularity, based on the offloading of individual loop statements. Section 3.3 proposes a computational processing determination method for the computation type using pattern matching. Section 3.4 proposes speedup by using OpenMP substitution according to the determined computational processing.

3.1 Automatic many-core CPU offloading of loop statements

Many-core CPUs utilize their numerous computational cores to increase speed by parallelizing processing. Unlike GPUs, many-core CPUs share common memory, eliminating the overhead of data transfer between CPU and GPU memory, a common issue when offloading to a GPU. Furthermore, the OpenMP specification is used to parallelize program processing on many-core CPUs. OpenMP uses directives such as #pragma omp parallel for to specify parallel processing for for statements. The OpenMP programmer is responsible for parallelizing processing in OpenMP. If a loop that cannot be parallelized is parallelized, the compiler does not output an error; instead, it outputs incorrect calculation results.

Based on this, automatic offloading of loop statements to many-core CPUs takes an evolutionary computing approach, creating multiple patterns for specifying whether the loop can be parallelized using OpenMP #pragma, and repeatedly measuring actual performance in a verification environment to gradually increase speed. The PGI compiler used for automatic GPU offloading output an error if parallelization was not possible. However, with OpenMP compilers such as gcc, such errors are the programmer's responsibility. Therefore, to automate this process, the processing performed by OpenMP directives is simplified to simply determining whether or not to parallelize a loop statement on a many-core CPU. Furthermore, a check for the correct final calculation result when parallelized is also performed during performance measurement. This ensures that only patterns that produce correct calculation results remain in the evolutionary computation.

Specifically, when code is entered, it is parsed using Clang[27] or similar to identify the loop statement. OpenMP code specifying parallel processing is created for the loop statement by adding #pragma omp parallel for. Here, a genetic pattern is created, with a value of 1 for parallel processing on a many-core CPU and a value of 0 for no parallel processing. The prepared patterns are then compiled using an OpenMP compiler such as gcc, and performance is measured on a verification environment machine equipped with a many-core CPU. Based on the performance measurement results, patterns with high processing speeds are assigned a high fitness, and patterns with low processing speeds are assigned a low fitness. Next-generation patterns are then created using processes such as elite selection, crossover, and mutation in a genetic algorithm. When measuring performance, the final calculation result is compared with the result obtained without parallel processing when the original code is processed on a CPU with fewer cores, and if the difference is unacceptably large, the fitness of that pattern is set to 0 and it is not selected for the next generation.

3.2 Ideas for many-core CPU offloading of middle size computation

Using the method described in Section 3.1, it is possible to create many-core CPU offload patterns and perform high-speed pattern search through verification environment measurements. However, while determining whether to offload each individual loop statement can achieve a certain degree of speedup (around ten times), achieving extremely large speedups is difficult.

This is because many-core CPUs often achieve speedups by taking advantage of the characteristics of multiple core processing and making full use of pipeline processing, so most speedups are done manually, considering many-core CPU processing algorithms for each calculation type, such as block matrix calculations and trivially parallel processing. Therefore, instead of determining each individual loop statement, we achieve automatic speedup by applying many-core CPU processing algorithms that have been previously

studied by many authors in other papers to calculations corresponding to larger-granularity calculation types.

The operation overview consists of the following two steps. First, we analyze whether the code to be offloaded contains calculations of a type that can be offloaded to a many-core CPU. If so, we speed up the processing by replacing it with an implementation that incorporates existing know-how applicable to many-core CPU processing. Here, the first step is explained in detail in Section 3.3, and the second step is explained in Section 3.4.

3.3 Searching for computational processing based on computation type

Firstly, we analyze the code to determine whether it contains computations of a type that can be offloaded. To determine the type of computation, pattern matching (e.g., [28] explanation) can be used. Pattern matching is a technique for searching for specific patterns. To determine the computation type, it is necessary to semantically match the program structure corresponding to the computation type in an abstract syntax tree using abstract terms that are independent of individual variable or function names. Commercially available tools capable of this type of pattern matching include Semgrep[29], which is available as an open-source tool.

For pattern matching searches, a code pattern database is created in advance to store code for computations of types that can be offloaded to many-core CPUs (e.g., block matrix calculations, trivially parallel processing calculations), their search patterns, and the OpenMP code for processing them on many-core CPUs. This database information is used to accelerate many-core CPU offloading, so it is expected that cloud providers offering many-core CPU VMs will prepare it with the aim of activating many-core CPU VM usage. Search patterns are created by replacing the variable and function names of the main computational processing units in the code with abstract terms. For OpenMP, which processes each type of calculation on a many-core CPU, we use open-source versions that have been studied and implemented in other papers.

One example of OpenMP that has been studied for speedup is the open-source OpenMP provided by NASA. Many existing studies have also explored speedups for various types of calculations on many-core CPUs using OpenMP.

When the user specifies the code to be offloaded, a pattern matching search is performed to see if it contains an offloadable calculation type. If no offloadable calculation type is found, we move on to Section 3.1, which describes the many-core CPU offload acceleration of loop statements. The cases where a pattern is found are explained in detail below.

The pattern matching tool automatically searches for search patterns registered in the code pattern database in order. The code to be offloaded provided by the user becomes the search target, and pattern matching is performed using abstract terms.

Step 1: Syntax Analysis of the Code to Be Offloaded The code to be searched is converted into an abstract syntax tree using a parser. Step 2: Abstract Syntax Tree Generation of the Search Pattern Like the code, the search pattern is also converted into an abstract syntax tree. Step 3: Traverse and Match the Tree Structure of the Abstract Syntax Tree Determine whether the search pattern abstract syntax tree matches the target abstract syntax tree as a subtree. Specifically, the abstract syntax tree subtree matching algorithm traverses the pattern abstract syntax tree against the target abstract syntax tree and checks for subtree isomorphism.

This allows us to determine whether the code stored in the code pattern database contains computational processes that can be offloaded to a many-core CPU.

3.4 Acceleration through OpenMP substitution

Since it is possible to determine whether the code contains calculations that can be offloaded to a many-core CPU, the found parts can be replaced with OpenMP code for processing on a many-core CPU, thereby increasing speed. OpenMP code is used for calculations such as block matrix calculations and trivially parallel processing calculations, and its speed has been improved through manual modifications in other papers. This implementation is packed with the accumulated know-how of experts.

The system searches for code patterns stored in the database. When a pattern is matched, the corresponding OpenMP implementation associated with that pattern is retrieved and used to replace the code. For example, in the evaluation in Section 5, Semgrep detects a C code pattern corresponding to the BT (Block Tri-Diagonal Solver), and the OpenMP implementation based on the NASA benchmark implementation is obtained and replaced.

However, since the code to be offloaded is pattern-matched and offloadable calculation types are replaced with OpenMP code from the code pattern database, there is no guarantee that the number and types of arguments and return values will match the user's requirements. If they do not match, since OpenMP is based on existing know-how and cannot be changed frequently, the user requesting offloading is asked whether they would like to change the number and types of arguments and return values in the original code to match OpenMP. After confirmation and approval, an offload performance test is conducted. Regarding type differences, if automatic casting between float and double is sufficient, the trial can proceed without user confirmation. Also, if the number of arguments or return values differs between the original code and OpenMP, for example, if arguments 1 and 2 are required in the user code and 3 is optional, and arguments 1 and 2 are required in OpenMP, and there is no problem if they are omitted, the optional arguments can be automatically omitted without prompting the user.

4 Implementation

4.1 Tools

This section describes the implementation used to verify the effectiveness of the proposed technology. To verify the effectiveness of many-core CPU offloading of larger-scale computations rather than individual loop statements, the target application is a C/C++ application. The many-core CPU used is an AMD Ryzen Threadripper PRO 3995WX with 64 cores. We also measure performance during GPU offloading to confirm its suitability for many-core CPUs. The GPU used is an NVIDIA GeForce RTX 3090. The machine used for compilation and execution is Iiyama LEVEL-R6W8-LCTP39-XAX (OS: Ubuntu 24.04.3 LTS, RAM: 128GB DDR4-3200). The 64-core CPU AMD Ryzen Threadripper PRO 3995WX used in our evaluation has a nominal power consumption (TDP) of 280W, while the GPU NVIDIA GeForce RTX 3090 has a nominal power consumption (TDP) of 350W.

The many-core CPU processing uses gcc/g++ 13.3.0[23]. While gcc can also execute standard C code, it also interprets and compiles code with #pragma directives added according to the OpenMP specification, outputting a binary file. This binary file is then executed on a many-core CPU.

For GPU processing, where performance is measured in advance, NVIDIA's compiler nvc/nvc++ 25.9.0 is used. nvc/nvc++ interprets and compiles code with #pragma directives added according to the OpenACC specification, outputting a binary file. This binary file is then executed on a GPU.

In addition, existing loop statement offloading performs C/C++ syntax analysis separately from pattern matching, using the LLVM/Clang 6.0 syntax analysis library (libClang's python binding) [27].

We use MySQL 8.0 for the code pattern database. The database stores computational code that can be offloaded to many-core CPUs, their search patterns, and the OpenMP code for processing them on many-core CPUs.

We use Semgrep 1 as the pattern matching tool. Semgrep is a tool that statically analyzes source code and can detect specific code patterns in multiple languages using pattern matching. By using abstract syntax trees to semantically match the search patterns in the code pattern database with the code to be offloaded, the system finds computational processes that can be offloaded.

The implementation is done in Python 3. Python handles all processing, including pattern matching and replacing the matching results.

4.2 Implementation operation

The implementation works as follows: When the code to be offloaded is entered, Semgrep searches to see if it contains an offloadable computation type.

Semgrep's search patterns are YAML files in which the names of variables and functions in the computation processing unit are capitalized and prefixed with $ to form abstract terms. Search patterns are stored in a code pattern

database, and the registered search patterns are used in order as search conditions. However, since searching through all of the registered code pattern databases is inefficient, users may specify search conditions. In this case, the search patterns are also assigned identifying information such as trivially parallel computation.

Semgrep's search target is the code to be offloaded specified by the user. Pattern matching involves the following steps: Step 1: Syntax analysis of the code to be searched; Step 2: Abstract syntax tree construction of the search pattern; and Step 3: Traversal matching of the tree structures of the abstract syntax trees.

If no offloadable computation type is found, the tool moves on to loop statement offloading, performing loop statement many-core CPU offloading using the same [19] operation as the previous implementation tool.

The newly added many-core CPU offloading for medium-grain computation is performed when Semgrep finds a computation type that can be offloaded.

If an offloadable computation type is found, OpenMP for processing on a many-core CPU is obtained from the code pattern database, and the part found by Semgrep is replaced to perform the processing using an OpenMP implementation that makes use of other people's know-how.

If the number and types of arguments and return values differ before and after the replacement, the user requesting the offload is asked whether they would like to change the number and types of arguments and return values in the original code to match OpenMP.

After verifying the arguments and return values, a many-core CPU offload performance test is conducted, and the performance, including the price of the many-core CPU VM, is presented to the user to confirm whether the performance is sufficient. After confirmation, the system is deployed to a commercial environment and actual use begins.

5 Evaluation

Rather than determining the offloading of individual loop statements and automatically offloading them to a many-core CPU, we evaluate the effectiveness of the proposed method of automatically offloading computations to a many-core CPU according to the computation type at a larger granularity.

5.1 Evaluation conditions

5.1.1 Evaluation applications

The evaluation applications are block matrix calculations and trivially parallel calculations that users are expected to use on many-core CPUs. We have proposed environment-adaptive software and implemented [19] technology for loop offloading, which verifies and selects the fastest offload destination in a

mixed environment of many-core CPUs, FPGAs, and GPUs. Many-core CPUs are fastest for block matrix calculations and trivially parallel calculations. Simple Fourier transforms and simple matrix calculations are faster on GPUs, so there is no point in offloading them to many-core CPUs, which consume high amounts of power.

Block tridiagonal matrix calculations are a numerical calculation method for quickly solving simultaneous linear equations whose coefficient matrices have a block tridiagonal structure. The block Thomas algorithm is used. This method utilizes the structure to quickly solve simultaneous linear equations with a tridiagonal structure and small matrices as components, and requires the use of a specific structure. The data size of the block tridiagonal matrix calculation BT (Block Tri-Diagonal Solver) [30] used in verification is Class = B.

Trivially parallel calculations provide an estimate of the achievable upper bound on floating-point performance, performance that requires little interprocessor communication. Among various use cases, this EP is a benchmark derived from a computational fluid dynamics application and is used in fluid dynamics. The data size of the trivially parallel calculation EP (Embarrassingly Parallel) [31] used in verification is Class = B.

5.1.2 Evaluation methods

The user specifies the application they wish to offload. Semgrep 1 performs pattern matching and automatically offloads the computation to the many-core CPU according to the computation type. After offloading, the search criteria and results are logged, and the processing time is measured for both single-core and many-core CPU offloading to assess the effectiveness of the offloading. Additionally, for comparison, when no computation type suitable for offloading using pattern matching is found, loop statements are offloaded to the many-core CPU and GPU using existing methods.

The conditions for offloading to the many-core CPU and GPU using existing genetic algorithms are as follows:

Number of loop statements: BT 120, EP 12

Number of individuals M: Less than or equal to the number of loop statements (BT 60, EP 12)

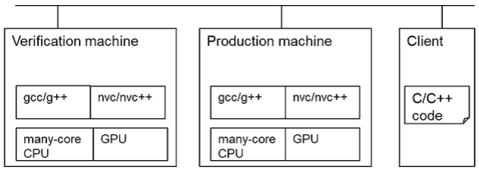
Number of generations T: Less than or equal to the number of loop statements (BT 60, EP 12)

Fitness: (Processing time) $^{-1/2}$ The shorter the processing time, the higher the fitness. Additionally, by using the (-1/2) power, we prevent the fitness of certain individuals with short processing times from becoming too high, narrowing the search range.

Selection: Roulette wheel selection. However, elite preservation is also performed, in which the genes with the highest fitness in one generation are preserved in the next generation without crossover or mutation.

Crossover rate Pc: 0.9

Mutation rate Pm: 0.05



Name	Hardware	CPU	RAM	GPU	OS	gcc/g++	nvc/nvc++
Verification machine for many-core CPU and GPU	LEVEL-R6W8-LCTP39-XAX	AMD Ryzen Threadripper PRO 3995WX (64 Core)	128 GB	NVIDIA GeForce RTX 3090	Ubuntu 24.04.3 LTS	13.3.0	25.9.0
Production machine for many-core CPU and GPU	LEVEL-R6W8-LCTP39-XAX	AMD Ryzen Threadripper PRO 3995WX (64 Core)	128 GB	NVIDIA GeForce RTX 3090	Ubuntu 24.04.3 LTS	13.3.0	25.9.0
Client	Lenovo ThinkPad X1 Carbon Gen10	Intel Core i7-1260P	32 GB		Windows 11 Pro		

Fig. 2 Performance measurement environment

5.1.3 Evaluation environment

The 64-core AMD Ryzen Treadripper PRO 3995WX was used as the many-core CPU for evaluation. The machine equipped with the AMD Ryzen Treadripper PRO 3995WX was the Iiyama LEVEL-R6W8-LCTP39-XAX (OS: Ubuntu 24.04.3 LTS, GPU: NVIDIA GeForce RTX 3090, RAM: 128GB DDR4-3200). The many-core CPU was controlled using gcc/g++. The C language program was extended according to the OpenMP grammar, and many-core CPU offloading was performed using OpenMP. Additionally, performance of GPU offloading was measured in advance, and nvc/nvc++ was used to control the GPU.

The evaluation environment and specifications are shown in Figure 2. The laptop PC specifies the application code to be offloaded. Performance measurements in the verification environment confirm the offload pattern, and the laptop is then deployed to the commercial environment.

5.2 Results

Figure 3 shows the search criteria and search result log for EPs searched using Semgrep pattern matching. While the search pattern itself uses abstract variable and function names, it can be seen that EPs with specific variables can be found by subtree matching of the abstract syntax tree. BT can also find EPs using abstract syntax tree matching in abstract terms.

Figure 4 shows the processing time for a single CPU, the processing time for a 64-core CPU after replacing with the obtained OpenMP, and the processing performance multiplier relative to a single CPU when pattern matching and offloadable calculation types are automatically offloaded to a many-core CPU using BT and EP. It also shows the processing performance multiplier when loop statement offloading is performed on a 64-core CPU using a genetic algorithm, a previous method.

For BT, the processing time for a single CPU is 110 seconds, and the processing time for a 64-core CPU after replacing with the obtained OpenMP is 6.64 seconds, resulting in a processing performance multiplier of 16.6x. Furthermore, the processing performance multiplier when loop statement offloading is performed on a 64-core CPU is 8.46x, demonstrating that the proposed method is superior in terms of performance multiplier. For EP, the processing time for a single CPU is 66.2 seconds, and the processing time for a 64-core CPU after replacing with the obtained OpenMP is 2.08 seconds, resulting in a processing performance multiplier of 31.8x. Furthermore, the processing performance multiplier when loop statement offloading is performed on a 64-core CPU is 16.6x, demonstrating that the proposed method is superior in terms of performance multiplier.

For example, Amazon offers VMs with multi-core CPUs, FPGAs, and GPUs in addition to low-core CPUs on its cloud service [3]. Typical monthly costs are approximately:

- 60 USD/month for a standard VM instance (m7g.large),
- 420 USD/month for a 16-core VM instance (c7g.4xlarge),
- 1,200 USD/month for an FPGA instance (f1.2xlarge),
- 470 USD/month for a GPU instance (g2.2xlarge).

Despite the difference in the number of VM cores, these two applications, which achieve over 15x performance, can be said to have a positive cost effect.

Through experiments, we found computational targets that can be offloaded to many-core CPUs using pattern matching, and showed that offloading to many-core CPUs makes offloading cost-effective, and that the method is effective.

5.3 Discussion

In our previous research on offloading loop statements to many-core CPUs, we measured the performance of multiple offload patterns in a testing environment and automatically selected the fastest pattern. This method was used to automatically accelerate many-core CPUs with a 32-core processor. Many-core CPUs require programs that take into account the hardware characteristics to achieve high speeds, and manual design is the norm. Automatic offloading represents a major departure from this approach. This time, instead of determining whether to offload individual loop statements, we focused on automatic offloading to many-core CPUs for computations based on the type of computation, such as larger-granularity block matrix calculations and trivially parallel computations. This approach promises greater performance improvements than loop offloading and represents a major advancement in many-core CPU offloading.

Consider cost. Many-core CPUs such as the AMD Ryzen Threadripper 2990WX and 3995WX cost around 5,000 USD. Therefore, in terms of hardware price alone, the price of a machine equipped with a many-core CPU is

Search pattern

```
EP rules.yaml
rules:
  - id: detect-EP
    pattern-either:
      - pattern: |     # pattern #1    [175]
          for (...) {
            ...
            for (i=1; i<=100; i++) {
              ...
              t3 = randlc(&t2, t2);
              ...
            }
            ...
            vranlc(2 * NK, &t1, A, x);
            ...
            for (i=0; i<NK; i++) {
              x1 = 2.0 * $X_0 - 1.0;
              x2 = 2.0 * $X_1 - 1.0;
              t1 = pow2(x1) + pow2(x2);
              if (t1 <= 1.0) {
                t2 = sqrt($CON_M2 * log(t1) / t1);
                t3 = (x1 * t2);
                t4 = (x2 * t2);
                l = max(fabs(t3), fabs(t4));
                q[l] += 1.0;
                sx = sx + t3;
                sy = sy + t4;
              }
            }
            ...
          }
    message: "The formula for EP has been found."
    languages: [c, cpp]
    severity: INFO
```

Search result

```
❱ detect-EP
   The formula for EP has been found.

      for(k=1; k<=np; k++){
        kk = k_offset + k;
        t1 = S;
        t2 = an;

        for(i=1;i<=100; i++){
              ik = kk / 2;
              if((2*ik)!=kk)(t3=randlc(&t1,t2);}
              if(ik==0){break;}
              t3=randlc(&t2,t2);
              kk=ik;
        }

        if(timers_enabled){timer_start(2);}
        vranlc(2*NK, &t1, A, x);
        if(timers_enabled){timer_stop(2);}

        if(timers_enabled){timer_start(1);}

        for(i=0;i<NK; i++){
              x1 = 2.0 * x[2*i] - 1.0;
              x2 = 2.0 * x[2*i+1] - 1.0;
              t1 = pow2(x1) + pow2(x2);
              if(t1 <= 1.0){
                    t2 = sqrt(-2.0 * log(t1) / t1);
                    t3 = (x1 * t2);
                    t4 = (x2 * t2);
                    l = max(fabs(t3), fabs(t4));
                    q[l] += 1.0;
                    sx = sx + t3;
                    sy = sy + t4;
              }
        }
        if(timers_enabled){timer_stop(1);}
      }
```

Fig. 3 EP (Embarrassingly Parallel) pattern matching search conditions and search results

Applications	Single core CPU processing time	Proposed method processing time (Change searched OpenMP)	Proposed method improvement ratio	Loop statements offloading improvement ratio
BT (Block Tri-Diagonal Solver	110 sec	6.64 sec	16.6	8.46
EP (Embarrassingly Parallel)	66.2 sec	2.08 sec	31.8	16.6

Fig. 4 Many-core CPUs offloading processing time results

roughly three times that of a machine equipped with only a few cores. However, in general, datacenter costs are less than one-third of those of conventional datacenters, with hardware and cloud system development costs exceeding one-third, electricity costs and operational costs such as maintenance and operations costs exceeding one-third, and other costs such as service orders remaining at roughly one-third. Therefore, even if the hardware costs themselves are roughly three times higher, this technology is considered cost-effective, achieving over 15 times the performance of processes that previously required time-consuming processing using a limited number of CPU cores. Incidentally, Amazon, a major commercial cloud service provider, offers multi-core CPU VMs at roughly two times the price of standard CPU VMs, yet still remains cost-effective.

Consider the time it takes to provide the service. Pattern matching searches and OpenMP substitutions take almost no time, measured in seconds, and compiling OpenMP and making it usable also takes almost no time, measured in seconds. Offloading based on the type of calculation is not possible. When attempting to offload loop statements, a genetic algorithm is used to measure multiple patterns and select the fastest one, which takes approximately eight hours. However, these processes are performed in a verification environment, not a commercial environment used by other users, so there is no impact on other users. Since cloud computing and other services are often not immediately available, we believe it is acceptable to require a day or so of testing before provisioning.

To offload computational processes to many-core CPUs according to their computational type, a code pattern database must be maintained in advance, containing the code for computational types that can be offloaded to many-core CPUs, their search patterns, and the OpenMP code for processing them on many-core CPUs. We expect cloud providers offering many-core CPU VMs to prepare this database to promote usage, but they may also do so for a fee. Accelerated OpenMP implementations have been implemented by other experts in separate papers, so it is possible to use the open-source versions.

This paper proposes an automatic acceleration method for many-core CPUs by taking advantage of the characteristics of the hardware and applying many-core CPU processing algorithms to the computational processes. While most acceleration methods are manually implemented, most are performed by applying many-core CPU processing algorithms that many users have considered. However, since using pattern matching to identify computational types and replacing them with libraries with validated processing algorithms is likely possible with GPUs as well, it is conceivable that a similar concept could be applied to other hardware.

6 Related work

[32] compared the performance of OpenMP implementations and applications on Intel Xeon Phi processors with over 60 cores. They analyzed the effects of the number of threads, memory mode, and thread placement. [33] performed hybrid MPI+OpenMP parallelization of Hartree-Fock calculations on a 64-core node. They reported scaling and memory reduction effects on processors with 64 cores per node, and demonstrated OpenMP on large-scale nodes. [34] evaluated the SPEC CPU/SPEC OMP benchmarks on a 48-core A64FX processor. Specifically, they examined thread scaling and memory performance of OpenMP benchmarks on the A64FX.

OpenMP is also frequently used for offloading to FPGAs and GPUs. [35][36][37] used OpenMP for FPGA offloading. [37] can interpret OpenMP code and perform FPGA offloading. [38][39][40] use OpenMP for GPU offloading. These methods require manual addition of instructions for parts to be parallelized using OpenMP or other specifications. There is no research on the authors'

goal of automatically converting existing code for FPGAs or GPUs without a new development model or manual insertion of directives.

OpenMP essentially controls kernel processing within a node, but when processing on multiple nodes, MPI is typically used to process between nodes. However, like OpenMP, MPI requires a high level of knowledge. Therefore, MPI hiding techniques have emerged that allow devices on different nodes to be exposed as local node devices and used without writing MPI code itself.[41] We are also considering using these MPI hiding techniques for processing on multiple nodes.

SYCL[42] is a single-source programming model for diverse hardware. While OpenCL requires separate code for host and kernel execution, SYCL allows them to be written in a single source. DPC++[43] is Intel's SYCL compiler. Both OpenCL and SYCL require new programs to use diverse hardware. SYCL is designed to run a single piece of code on multiple pieces of hardware, which makes it an improvement over OpenCL. However, the single piece of code must be newly created by the programmer. This differs from the goal of this paper, which does not require the manual creation of new code for OpenCL or SYCL.

As mentioned above, there have been many efforts to speed up processing by offloading to many-core CPUs, but most of these efforts involve manually adding instructions on which parts to parallelize and then offloading accordingly; offloading existing code to many-core CPUs and similar devices is mostly done manually, with few automated efforts. While our research has previously focused on accelerating individual loop statements, this paper expands on this to include automatic offloading of computational processes according to the computation type, which is a larger unit of effort.

7 Conclusion

In this paper, as an extension of our proposed environment-adaptive software, we propose a method to analyze user-provided applications, regardless of individual loop statements, and automatically offload them to a many-core CPU using an appropriate processing algorithm depending on the computation type (e.g., block matrix calculation, trivially parallel computation, etc.).

First, the user application is analyzed. The pattern matching tool Semgrep is used to search for processing patterns corresponding to computation types (e.g., block matrix calculation, trivially parallel computation, etc.). To perform matching searches using Semgrep, the code, search patterns, and corresponding OpenMP statements are stored in a code pattern database in advance. Semgrep's pattern matching uses a semantic search using an abstract syntax tree to search for computations of a certain computation type that contain a replaceable OpenMP statement. If no OpenMP statement is found, we attempt to speed up the loop statement using a genetic algorithm, as previously investigated. If a replaceable OpenMP statement is found, we replace it with that OpenMP statement and measure performance improvement. If offloading

to a many-core CPU makes sense from a cost perspective, users will approve to use.

In this verification, we used Semgrep to analyze the computational types BT, for block matrix calculations, and EP, for trivially parallel calculations, and then replaced them with the corresponding OpenMP to measure performance. We confirmed a performance improvement of more than 15 times, which makes offloading worthwhile considering the price of multi-core CPU VMs, demonstrating the effectiveness of our method.

We plan to take two directions in the future. One is to trial automatic many-core CPU offloading according to computational type for more practical applications. The other is to apply automatic offloading according to computational type to other hardware such as GPUs, although currently offloading using accelerators is performed on many-core CPUs, FPGAs, and GPUs.

Declarations

Ethical Approval and Consent to participate

Not Applicable.

Consent for publication

Not Applicable. Only Yoji Yamato did all of this research and Yoji Yamato approved the publication of this manuscript.

Availability of supporting data

Not Applicable.

Competing interests

The author declares no competing interest with this manuscript.

Funding

The author declares no funding with this manuscript.

Authors' contributions

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

Acknowledgments

Not Applicable.

References

1. 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.
2. 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.
3. AWS EC2 web site, https://aws.amazon.com/ec2/instance-types/
4. M. Hermann, T. Pentek and B. Otto, "Design Principles for Industrie 4.0 Scenarios," Rechnische Universitat Dortmund. 2015.
5. T. Sterling, M. Anderson and M. Brodowicz, "High performance computing: modern systems and practices," Cambridge, MA: Morgan Kaufmann, ISBN 9780124202153, 2018.
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 and E. Kandrot, "CUDA by example: an introduction to general-purpose GPU programming," Addison-Wesley, 2011.
8. Y. Yamato, "Proposal of Automatic GPU Offloading Method from Various Language Applications," The 9th International Conference on Information and Education Technology (ICIET 2021), pp.400-404, Mar. 2021.
9. 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.
10. Y. Yamato, "Automatic Verification Technology of Software Patches for User Virtual Environments on IaaS Cloud," Journal of Cloud Computing, Springer, Vol.4, No.4, DOI: 10.1186/s13677-015-0028-6, Feb. 2015.
11. Y. Yamato, "A study for environmental adaptation of IoT devices," 2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW 2023) pp.14-19, Nov. 2023.
12. Y. Yamato, "Study of software reconfiguration after adapted service start," 2023 5th International Electronics Communication Conference (IECC 2023), pp.63-68, July 2023.
13. Y. Yamato, "Evaluation of GPU Logic Reconfiguration after Service Start," The 11th International Conference on Information and Education Technology (ICIET 2023), pp.551-556, Mar. 2023.
14. Y. Yamato, "Study and Evaluation of Automatic Offloading for Function Blocks of Applications," Automatika, Taylor & Francis, Vol.65, Issue.1, pp.387-400, DOI: 10.1080/00051144.2024.2301888, Jan. 2024.
15. Y. Yamato, "Proposal and evaluation of GPU offloading parts reconfiguration during applications operations for environment adaptation," Journal of Network and Systems Management, Springer, DOI: 10.1007/s10922-023-09789-2, Nov. 2023.
16. Y. Yamato, "Study and Evaluation of FPGA Reconfiguration during Service Operation for Environment-Adaptive Software," International Journal of Parallel, Emergent and Distributed Systems, Taylor & Francis, DOI: 10.1080/17445760.2023.2242639, Aug. 2023.
17. Y. Yamato, "Study and Evaluation of Optimum Location Deployment for Environment Adaptive Applications," International Journal of Parallel, Emergent and Distributed Systems, Taylor & Francis, DOI: 10.1080/17445760.2022.2088749, June 2022.

18. Y. Yamato, "Proposal and Evaluation of Adjusting Resource Amount for Automatically Offloaded Applications," Cogent Engineering, Taylor & Francis, Vol.9, Issue 1, DOI: 10.1080/23311916.2022.2085467, June 2022.
19. Y. Yamato, "Study and Evaluation of Automatic Offloading Method in Mixed Offloading Destination Environment," Cogent Engineering, Taylor & Francis, Vol.9, Issue 1, DOI: 10.1080/23311916.2022.2080624, June 2022.
20. Y. Yamato, "Study and evaluation of automatic division of general-purpose programs to facilitate addition of user functions," International Journal of Parallel, Emergent and Distributed Systems, Taylor & Francis, DOI: 10.1080/17445760.2024.2375650, Aug. 2024.
21. Y. Yamato, "Study and evaluation for adopting environmental adaptation of low-resource devices," IEEE Access, DOI: 10.1109/ACCESS.2024.3440918, Aug. 2024.
22. AMD Ryzen website, https://www.amd.com/ja/products/processors/desktops/ryzen.html
23. gcc website, https://gcc.gnu.org/
24. S. Wienke, P. Springer, C. Terboven and D. an Mey, "OpenACC-first experiences with real-world applications," Euro-Par 2012 Parallel Processing, pp.859-870, 2012.
25. M. Wolfe, "Implementing the PGI accelerator model," ACM the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp.43-50, Mar. 2010.
26. 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.
27. Clang website, http://llvm.org/
28. Haskell description website, https://wiki.haskell.org/Declaration_vs._expression_style
29. Semgrep website, https://github.com/semgrep
30. Block Tri-Diagonal Solver website, https://www.nas.nasa.gov/software/npb.html
31. Embarrassingly Parallel website, https://www.nas.nasa.gov/assets/nas/pdf/techreports/1994/rnr-94-007.pdf
32. T. Cramer, D. Schmidl, M. Klemm and D. an Mey, "OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison," Many Core Appl. Res. Community (MARC) Symp, pp.38-44, 2012.
33. V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky and M. S. Gordon, "An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor," International conference for high performance computing, networking, storage and analysis, pp.1-12, Nov. 2017.
34. Y. Kodama, M. Kondo and M. Sato, "Evaluation of spec cpu and spec omp on the a64fx," 2021 IEEE International Conference on Cluster Computing (CLUSTER), pp.553-561, Sep. 2021.
35. J. Huthmann, L. Sommer, A. Podobas, A. Koch and K. Sano, "OpenMP device offloading to FPGAs using the Nymble infrastructure." International Workshop on OpenMP, pp.265-279, Cham: Springer International Publishing, Sep. 2020.
36. M. Knaust, F. Mayer and T. Steinke, "OpenMP to FPGA offloading prototype using OpenCL SDK," 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp.387-390, May 2019.
37. 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.
38. G. T. Bercea, C. Bertolli, A. C. Jacob, A. Eichenberger, A. Bataev, G. Rokos and K. O'Brien, "Implementing implicit OpenMP data sharing on GPUs," The Fourth Workshop on the LLVM Compiler Infrastructure in HPC, pp.1-12, Nov. 2017.
39. C. Bertolli, S. F. Antao, G. T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans and K. O'Brien, "Integrating GPU support for OpenMP offloading directives into Clang," ACM Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM'15), Nov. 2015.
40. S. Lee, S.J. Min and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'09), 2009.
41. 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.

42. SYCL web site, https://www.khronos.org/sycl/
43. DPC++ web site, https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-library.html#gs.flx6xq