International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 2, pages 167–180, July 2016

Translation of Large-Scale Simulation Codes for an OpenACC Platform Using the Xevolver Framework

Kazuhiko Komatsu, Ryusuke Egawa
Cyberscience Center, Tohoku University
6-3, Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi, 980-8578, Japan

Shoichi Hirasawa, Hiroyuki Takizawa
Graduate School of Information Sciences, Tohoku University
6-6-01, Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi, 980-8578, Japan

Ken'ichi Itakura
CEIST, Japan Agency for Marine-Earth Science and Technology
3173-25, Syowa-machi, Kanazawa-ku, Yokohama, Kanagawa, 236-0001, Japan

Hiroaki Kobayashi
Graduate School of Information Sciences, Tohoku University
6-6-01, Aramaki Aza Aoba, Aoba-ku, Sendai, Miyagi, 980-8578, Japan

Received: February 15, 2016
Accepted: April 7, 2016
Communicated by Reiji Suda

Abstract

As the diversity of high-performance computing (HPC) systems increases, even legacy HPC applications often need to use accelerators for higher performance. To migrate large-scale legacy HPC applications to modern HPC systems equipped with accelerators, a promising way is to use OpenACC because its directive-based approach can prevent drastic code modifications. This paper shows translation of a large-scale simulation code for an OpenACC platform by keeping the maintainability of the original code. Although OpenACC enables an application to use accelerators by adding a small number of directives, it requires modifying the original code to achieve a high performance in most cases, which tends to degrade the code maintainability and performance portability. To avoid such code modifications, this paper adopts a code translation framework, *Xevolver*. Instead of directly modifying a code, a pair of a custom code translation rule and a custom directive is defined, and is applied to the original code using the Xevolver framework. This paper first shows that simply inserting OpenACC directives does not lead to high performance and non-trivial code modifications are required in practice. In addition, the code modifications sometimes decrease the performance when migrating a code to other platforms, which leads to low performance portability. The direct code modifications can be avoided by using pairs of an externally-defined translation rule and a custom directive to keep the original code unchanged as much as possible. Finally, the performance evaluation shows that the performance portability can be improved by selectively applying translation with the Xevolver framework compared with directly modifying a code.

Keywords: Code Translation Framework, Performance Portability, Maintainability

1 Introduction

Nowadays, high-performance computing (HPC) systems are equipped with so-called accelerators such as graphics processing units (GPUs) and Intel Xeon Phi. The latest TOP500 list has shown that more than 30% of the TOP500 peak performance is occupied by accelerator-based systems [5]. Because the peak performance and energy efficiency of an accelerator are high, an accelerator is attractive for building HPC systems in the situation where the power budget of an HPC system is strongly limited. Therefore, an accelerator becomes a strong candidate to accelerate not only newly-developed HPC applications but also existing HPC applications, so-called *legacy HPC applications* that have been maintained for a long time.

One of obstacles to migrate a legacy HPC application to an accelerator is low code maintainability. Because a legacy HPC application tends to be complicated and large, neither rewriting the code nor drastically modifying the code with low-level programming models such as CUDA [10] and OpenCL [3][9] is a realistic solution. In addition, such a low-level programming degrades the code maintainability because an application developer has to maintain different versions of an application code; the original version and the low-level programming version.

Directive-based approaches that employ compiler directives such as OpenMP [4] [8] have widely been used to optimize an HPC application code while keeping its code maintainability. Various features of directives can be utilized for annotating a code. Because inserted directives can also be treated as comment lines by disabling the directive functions, only one version of the code can be maintained. Moreover, directives can be inserted incrementally during the development and porting process. Thus, directive-based approaches are promising to easily migrate a legacy HPC application to modern HPC platforms.

For migration of a legacy HPC application to an accelerator by directive-based approaches, OpenACC [2] has been an attractive candidate. OpenACC provides directives to execute computation on an accelerator. However, simply inserting OpenACC directives cannot always exploit the potential of an accelerator. To exploit the potential, an OpenACC code needs to be modified so that the compiler can understand the structure of the code. These modifications often degrade the code maintainability for an application developer, and also the performance on other platforms called *performance portability.*

This paper discusses translation of a large-scale simulation code for an OpenACC platform in order to keep the maintainability of the original code. To avoid code modifications, a code translation framework, *Xevolver* [15], is adopted in addition to employing a directive-based approach. The framework can avoid modifying the original code by using a pair of an external user-defined rule and a custom directive to keep the maintainability of the original code. In this paper, the effectiveness of our approach through a case study of migration of an atmospheric simulation code to an accelerator are examined.

The rest of this paper is organized as follows. Section 2 describes the related work on migration of HPC application codes to modern HPC systems and code translation frameworks for migration. Section 3 explains the translation of a large-scale atmospheric simulation code for an OpenACC platform by using the Xevolver framework. Section 4 shows performance evaluations to demonstrate the effectiveness of the code translation. Section 5 gives concluding remarks and future work.

2 Migration of HPC Application Codes

Directive-based approaches are often used in the field of HPC. For example, compiler pragmas can give additional information to a compiler for effective analysis and optimizations of a code. OpenMP helps use a serial code as a multithreaded one running on a shared memory system.

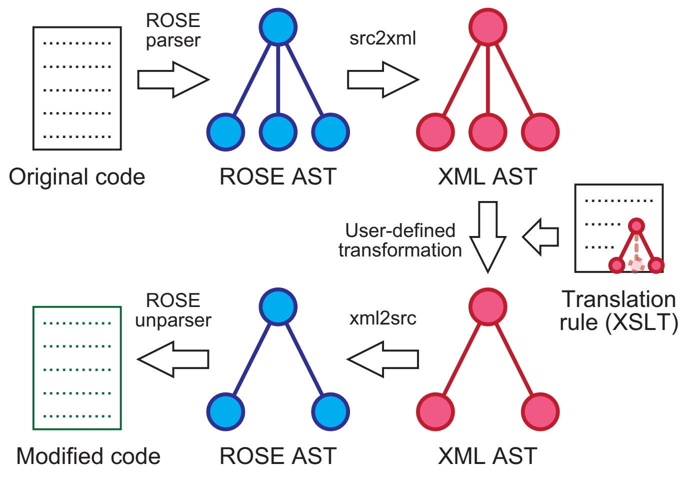

Figure 1: An overview of translation using the Xevolver framework.

As OpenMP specification version 4.0 includes the support of accelerators, compilers that support OpenMP 4.0 specification are becoming more popular for programming accelerators. This paper focuses on OpenACC that has been firstly released as a directive-based programming model for an accelerator because OpenACC is the most popular directive at present.

There are several advantages of the directive-based approaches in terms of maintainability. First, the structure of the original code need not be modified when the features of directives are utilized. Simply inserting directives into a code realizes naive code migration. As no drastic code modification is necessary, an application developer can easily maintain the code after the migration using directives. Second, migration of an application code using directives can be incremental. Even when an application code is partially migrated and the migration processing is ongoing, the application is executable for checking its behaviors. Third, only one unified code can be maintained because directives are treated as comment lines when a compiler disables the directive functions. From these reasons, directive-based approaches generally lead to high maintainability of an application.

However, simply inserting directives is often unable to exploit the potential of HPC systems. In HPC application development, the highest priority is given to the performance. As a result, further optimizations by modifying the original code are frequently required. It is common that many code modifications are necessary for the optimizations. Therefore, there is a demand for helpful tools that can keep the original code unchanged as much as possible for code modifications.

Code translation frameworks such as ChiLL [7], POET [16], and Xevolver [15] can be used for migration of HPC applications. These frameworks can perform code translation according to custom translation rules. ChiLL and POET are, in particular, specialized for combinations of pre-defined loop translation rules. In addition to combinations of pre-defined rules, Xevolver can easily define custom code translation rules using XML (eXtensible Markup Language) [1]. Thus, more flexible code modifications can be achieved by the Xevolver framework.

Figure 1 shows an overview of translation by the Xevolver framework. First, an application code is parsed by using the ROSE compiler infrastructure [11], and then its AST (Abstract Syntax Tree) is converted to an XML document by the *src2xml* command of the Xevolver framework. As an AST is represented as an XML document called an *XML AST*, many XML techniques can be applied to the XML AST. To describe a translation rule in the Xevolver framework, XSLT (XML Stylesheet Language Transformations) to describe XML data conversion is utilized [6]. A translation rule can also be generated from Fortran codes [12]. Furthermore, the Xevolver framework provides basic translation rules as pre-defined rules that can also be utilized as a template of a new custom rule. After a transformation of the XML AST, the modified XML AST is converted by the *xml2src* command of the Xevolver framework. Then, it is unparsed through the ROSE compiler infrastructure. Finally, a modified version of the application code is generated.

```
1401:  !$acc kernels                             &
1402:  !$acc present(r2cp, sqrtg2, dzs2)         &
1403:  !$acc present(dphi, dlam)
1404:        do k = ks , ke
1405:           zscl01 = cintp_z_scl(k,1)
1406:           zscl02 = cintp_z_scl(k,2)
1407:           c01 = zscl01*0.5_DP
1408:           c02 = zscl02*0.5_DP
1409:           do j = js , je
1410:              detj_h = r2cp(j,k)*dphi*dlam
1411:              tmp3 = ksmagp2 *detj_h
1412:              do i = is , ie
1413:                 detj_v = sqrtg2(i,j)*dzs2(k)
```

(a) Insertion of the directives into the code.

```
   1404, Loop is parallelizable
         Accelerator kernel generated
       1404, !$acc loop gang ! blockidx%x
       1412, !$acc loop vector(128) ! threadidx%x
```

(b) Compile log of the code by the PGI compiler.

Figure 2: Insertion of the OpenACC directives and its compile log.

The Xevolver framework can define custom translation rules of code modifications and apply the pre-defined and newly-defined translation rules to a code. Thus, by using the Xevolver framework, code modifications to achieve high performance can be replaced with externally defined custom rules to keep the maintainability of the original code.

3 Translation of an Atmospheric Simulation Code for an OpenACC Platform Using the Xevolver Framework

This section discusses migration of a large-scale atmospheric simulation code to an OpenACC platform as a case study of HPC application migration. MSSG (Multi-scale Simulator for the Geo-environment) has been developed for a global geo-environment simulation that incorporates non-hydrostatic atmosphere, ocean, and sea-ice components [14][13]. MSSG can conduct seamless comprehensive simulations at different scales from the entire global area to an urban area. The MSSG code is written in Fortran 90 and parallelized using MPI and OpenMP. The size of the code is about 420K lines. This paper focuses on migration of the atmosphere component in MSSG into an OpenACC platform because the atmosphere simulation is the most crucial part to achieve a high performance.

3.1 Migration to an OpenACC Platform

The most naive code modification is to insert OpenACC directives into the code. OpenACC *kernels* directives are inserted into loop nests whose calculations can be offloaded to an accelerator. Figure 2(a) shows an example of the insertion of an OpenACC *kernels* directive into a code. In Line 1401, the OpenACC *kernels* directive is inserted because the following do-loop does not have inter-iteration data dependency and can be offloaded to an accelerator. Figure 2(b) shows the compile log of the code by using the PGI compiler that supports OpenACC. The compile log indicates that the do-loop in Line 1404 is translated to an *accelerator kernel*, which is a part of the application code executed on an accelerator.

When an OpenACC compiler cannot detect parallelizable loop nests, OpenACC *loop* directives with *gang* and *vector* clauses need to manually be inserted. These clauses can allow a compiler to know parallelism of loop nests. An *independent* clause of the *loop* directive is sometimes required to

```
1401:  !$acc kernels                               &
1402:  !$acc present(r2cp, sqrtg2, dzs2)           &
1403:  !$acc present(dphi, dlam)
1404:        do k = ks , ke
1405:           do j = js , je
1406:              zscl01 = cintp_z_scl(k,1)
1407:              zscl02 = cintp_z_scl(k,2)
1408:              c01 = zscl01*0.5_DP
1409:              c02 = zscl02*0.5_DP
1410:              detj_h = r2cp(j,k)*dphi*dlam
1411:              tmp3 = ksmagp2 *detj_h
1412:              do i = is , ie
1413:                 detj_v = sqrtg2(i,j)*dzs2(k)
```

(a) Modification of the loop-invariant code motion.

```
   1404, Loop is parallelizable
   1405, Loop is parallelizable
         Accelerator kernel generated
       1404, !$acc loop gang ! blockidx%x
       1405, !$acc loop gang ! blockidx%y
       1412, !$acc loop vector(128) ! threadidx%x
         Loop is parallelizable
```

(b) Compile log of the modified code by the PGI compiler.

Figure 3: Modification of the code and its compile log.

indicate that there is no data dependency among loop iterations. By using these basic OpenACC *kernels* and *loop* directives, accelerator kernels are generated in order to offload computationally-intensive parts of the application to an accelerator.

In addition to the basic OpenACC directives, an OpenACC *data* directive is appropriately inserted considering the data location to avoid redundant data transfers between a host and an accelerator. Otherwise, the whole data required for the execution of a kernel are transferred every kernel execution even if the same data are already stored in the memory of a host and an accelerator.

However, simply inserting OpenACC directives is not always enough to exploit the potential of an accelerator. Figure 3(a) shows a modified version of the code of Figure 2(a). The loop-invariant code fragment from Lines 1405 to 1408 in Figure 2(a) is moved into the body of the do-loop of j to generate a perfectly-nested loop as shown in Figure 3(a). By generating the perfectly-nested loop, the OpenACC compiler successfully detects the parallelizable do-loops of both k and j. The compiler utilizes the do-loops of k and j as block indices of x and y in the accelerator kernel, respectively. The compiler log in this case is shown in Figure 3(b). Although this modification generally increases the computation amount, it is expected to improve performance due to an increase in the number of threads that can be executed in parallel on an accelerator. As shown in this example, code modifications are still necessary for an OpenACC code to achieve a higher performance.

3.2 Translation Using the Xevolver Framework

Table 1 shows code modifications necessary for migration of the MSSG code to an OpenACC platform. The numbers in the table show how many times each optimization technique is required in the code. For a basic optimization such as loop interchange, the pre-defined rule provided by the Xevolver framework can be utilized. By using the pre-defined loop interchange rule as a template of a new custom rule, loop interchange customized for migration can easily be defined.

This paper also focuses on the repetitive modifications. One of such modifications is replacement of pointer variables declared in every derived type. The modification is indispensable due to the limitation of OpenACC 2.0. As the OpenACC 2.0 specification does not fully support derived types in Fortran, members of a derived type with an allocatable or pointer attribute must be avoided. The translation to replace the pointer members of the derived type with the Fortran basic type has

Table 1: OpenACC optimizations that require code modifications.

OpenACC optimizations	# of modifications
Replacement of pointer variables	793
Loop interchange	1
Code motions for a perfectly-nested loop	20
Replacement of data transfers to calculations	1
Combination of loop interchange and loop fusion	5
Combination of loop interchange and loop fission	6

been examined in [15]. To avoid a number of code modifications scattered over a code, the Xevolver framework has taken a role of translating all the pointer members to the Fortran basic types.

Another repetitive modification is the loop-invariant code motion that appears 20 times in the migration. Because there are many loop nests that are similar code structures in the MSSG code, the similar modifications are applied to the loop nests. The repetitive modifications over the code could trigger bugs and cause a maintenance problem. To effectively avoid the manual modifications, the code motion is implemented by defining a pair of a custom translation rule and a custom directive using the Xevolver framework.

To enable an OpenACC compiler to understand the parallelism of the nested loops, even the loop-invariant calculation moves into the body of the inner loop in the example of modifying the code in Figure 2(a) to that in Figure 3(a). The repetitive translation can be expressed as a common pair of one translation rule written in XSLT and one custom directive. Figure 4 shows the custom translation rule of the code motion. From Lines 1 to 3, the type of this document is defined. These lines indicate that this document is described in XSLT. Line 5 searches for the target AST pattern from an XML AST by using the *match* attribute of the XSLT template rule. The *SgFortranDo* statement in an XML AST matches *Do* statements in a Fortran code. Thus, Line 5 searches for *Do* statements with the *code_motion* mode attribute. The *code_motion* mode attribute is given to the *SgFortranDo* statement by inserting a custom directive *"!$xev code_motion"* just before the target *Do* statement in the code. The directive indicates that the following *Do* statement has another *Do* statement in its body. Namely, it is followed by a nested loop. However, the loop is not perfectly-nested because there are other statements between those two *Do* statements. Thus, a perfectly-nested loop needs to be generated by moving the statements between the two *DO* statements to the body of the innermost loop. From Lines 6 to 27, the transformation is applied to the sub-tree of the target AST pattern according to the rule. Lines 6 to 11 generate the same loop structure of the *Do* statement with the *code_motion* mode attribute by using the *copy* and *copy-of* elements of XSLT. Then, from Lines 13 to 18, the inner *Do* loop is generated right after the target *Do* loop. Lines 19 and 20 generate the loop body statements of the first and second loops in the nested loop, respectively. Because a perfectly-nested loop can be generated, an OpenACC compiler exploits the parallelism of the perfectly-nested loop for efficient parallel processing.

The other code modifications shown in Table 1 are also implemented by defining pairs of a custom translation rule and a custom directive. Even though the number of occurrences that each code modification is applied to the code is small, it is important to define these code modifications separately from the original code in the aspect of the code maintainability. For example, loop fusions and loop fissions generally need drastic changes of the code structure. Thus, the maintainability of the original code may decrease. By implementing such code modifications as external translation rules by the Xevolver framework, the original code can be kept unchanged as much as possible, resulting in keeping the maintainability of the original code.

As all code modifications in Table 1 are defined separately from the original code, the code modifications can easily be enabled or disabled. In advance of the compilation, by considering a target platform, a user can choose whether each code modification should be applied or not. Therefore, the approach to code modifications by a custom translation with a custom directive is

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:exslt="http://exslt.org/common">

  <xsl:template match="SgFortranDo" mode="code_motion">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:copy-of select="./*[1]" />
      <xsl:copy-of select="./*[2]" />
      <xsl:copy-of select="./*[3]" />
      <xsl:element name="SgBasicBlock">

        <xsl:element name="SgFortranDo">
          <xsl:copy-of select="SgBasicBlock/SgFortranDo/@*" />
          <xsl:copy-of select="SgBasicBlock/SgFortranDo/*[1]" />
          <xsl:copy-of select="SgBasicBlock/SgFortranDo/*[2]" />
          <xsl:copy-of select="SgBasicBlock/SgFortranDo/*[3]" />
          <xsl:element name="SgBasicBlock">
            <xsl:copy-of select="SgBasicBlock/SgExprStatement" />
            <xsl:copy-of select="SgBasicBlock/SgFortranDo/SgBasicBlock/*" />
          </xsl:element>
          <xsl:copy-of select="SgBasicBlock/SgFortranDo/PreprocessingInfo" />
        </xsl:element>

      </xsl:element>
      <xsl:copy-of select="PreprocessingInfo" />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Figure 4: The translation rule for the code motion.

expected to enhance performance portability.

4 Performance Evaluation

In this section, we demonstrate the results of performance evaluation in order to clarify the necessities of the non-trivial code modifications. Then, the effects of translation by using the Xevolver framework are discussed through the performance evaluation.

4.1 Experimental Environment

In the evaluation, a hybrid computing system that is equipped with two Intel Xeon E5-2630 processors, Nvidia Tesla K20, and 64GB main memory is utilized. The PGI compiler 14.02 on CentOS 6.2 is used as an OpenACC compiler. The other systems used in the evaluation are a vector supercomputing system NEC SX-ACE equipped with an SX-ACE processor, 64 GB main memory, and an Intel-based scalar system LX equipped with two Intel Xeon E5-2695v2 processors and 128 GB main memory, which are installed in Cyberscience Center of Tohoku University. NEC SX Fortran90 Rev.501 compiler and Intel Fortran Compiler 15.0.2.164 are utilized, respectively.

4.2 Performance of the Optimized Code on GPU

Figure 5 shows the performance of three versions of the OpenACC codes on a GPU. The first version is a basic code, in which OpenACC directives are simply inserted to the original code. The second version is an optimized code, to which all the code translations in Table 1 are applied by using the Xevolver framework. The third version is another optimized code, to which the same code modifications are directly applied by hand without the Xevolver framework. The x-axis indicates the five major kernels in MSSG. The y-axis indicates the GPU performance of each code normalized

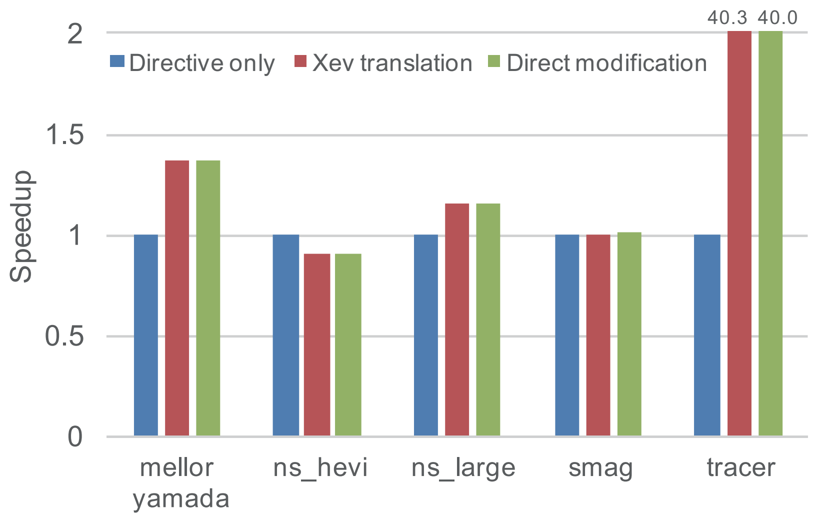

Figure 5: The effects of all code modifications on GPU.

by that of the first version of the code. This figure shows that the second and third versions of the code are accelerated by GPU except for the *ns_hevi* kernel. In both the second and third versions, the speedup ratios of the *mellor_yamada*, *ns_large*, and *tracer* kernels are about 37%, 16%, and 400%, respectively. These results indicate that simply inserting OpenACC directives could not always exploit the potential of an accelerator because the OpenACC compiler might be unable to effectively parallelize the code. However, by appropriately modifying the code, the OpenACC compiler could successfully parallelize the code, and hence a higher performance can be obtained. Therefore, it is clarified that, even if directive-based programming models such as OpenACC are adopted, code modifications are still necessary to achieve high performance on an accelerator.

As shown in Figure 5, the second and third versions have the same performance because an identical code is eventually passed to the compiler in both cases. Because code modifications in Table 1 usually help the compiler find the loop parallelism, the second and third versions can usually achieve a higher performance than the first one. However, in the case of the *smag* kernel, all the three versions have the same performance because code transformations in Table 1 do not affect the performance.

One exception is the *ns_hevi* kernel, for which the performance of the second and third versions is lower than that of the first one. This means that code modifications in Table 1 might degrade the performance in some cases. In such a case, the performance portability of the second version can be higher than that of the third version. In the case of translating the code with the Xevolver framework, i.e., the second version, all code modifications are represented as pairs of a custom translation rule and a custom directive. Thus, it is easy to cancel the effects of code modifications by just disabling the code translation, resulting in the same performance as the first version. On the other hand, in the case of directly modifying the original code, i.e., the third version, it is difficult to undo the code modification. To achieve high performance portability without code translation, an application developer would need to maintain multiple versions, i.e., the first and third versions.

4.3 Maintainability of the Code Translation

In order to clarify the code maintainability, the number of modified lines is examined. Figure 6 shows the number of modified code lines. The x-axis indicates the five kernels in MSSG. The y-axis indicates the numbers of modified code lines in the second and third versions of the code. This figure clearly shows that the number of modified code lines in the second version is smaller than that in the third version. In the cases of the *mellor_yamada* and *tracer* kernels, the difference of the

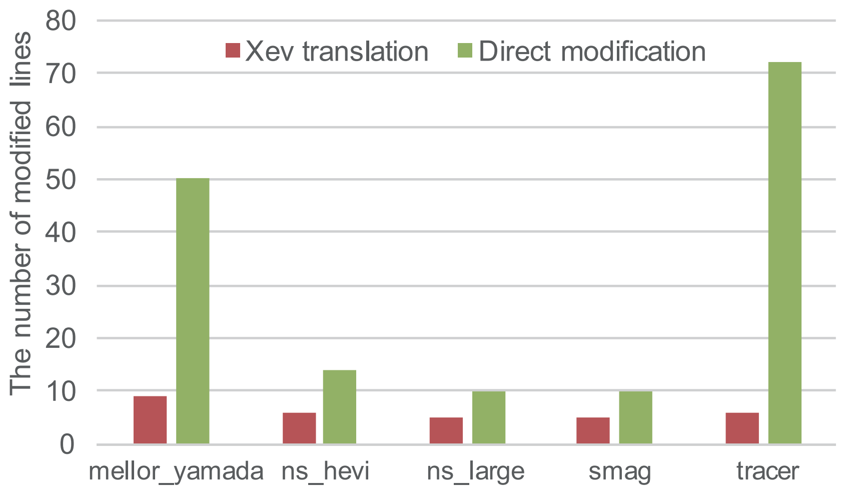

Figure 6: The number of modified code lines.

numbers of modified code lines becomes large because the complicated modifications such as loop fissions and loop fusions are included for the kernels. In the cases of the other *ns_hevi*, *ns_large*, and *smag* kernels, the difference is not so large because most of code modifications are the loop-invariant code motions.

The total numbers of modified code lines in the second and third versions are 31 and 156, respectively. The total number of modified code lines in the second version is about one fifth of that in the third version. In the case of the second version, only inserting custom directives into the original code is necessary because the code modifications can be externally defined by the Xevolver framework. For a custom code translation, although a user needs to write an externally-defined translation rule, the user can reuse the rule. Furthermore, if the rule is written in general enough, the user can utilize the same rule even for another application. For basic modifications that the Xevolver framework provides as pre-defined rules, a user does not need to write the rules. On the other hand, in the case of the third version, the direct modifications are necessary to the original code. Thus, the number of modified code lines in the original code is different between two versions. From the result, it is clarified that translation with the Xevolver framework can successfully keep the maintainability of the original code.

4.4 Performance Portability of the Code Modifications

In order to clarify the performance portability of the code modifications, Figure 7 shows the performance of the second version on multiple platforms. The x-axis indicates the five kernels in MSSG. The y-axis indicates the performance of the second version normalized by that of the first version on each platform. This figure shows that code modifications beneficial to achieve a higher performance on one platform might be harmful on another platform. The performance impact depends on the combination of a kernel and a platform. In the cases of the *mellor_yamada*, *ns_large*, and *tracer* kernels, only the GPU performance improves while the others remain unchanged. On the other hand, in the case of the *ns_hevi* kernel, only the GPU performance decreases while the others remain unchanged. In the case of the *smag* kernel, only the performance of LX decreases. From the above the results, it is shown that the best code among the three codes to achieve high performance could change depending on the target platform. Because the Xevolver framework can separate the code modification from the original code, an appropriate translation can be selectively applied by considering the characteristics of a target platform.

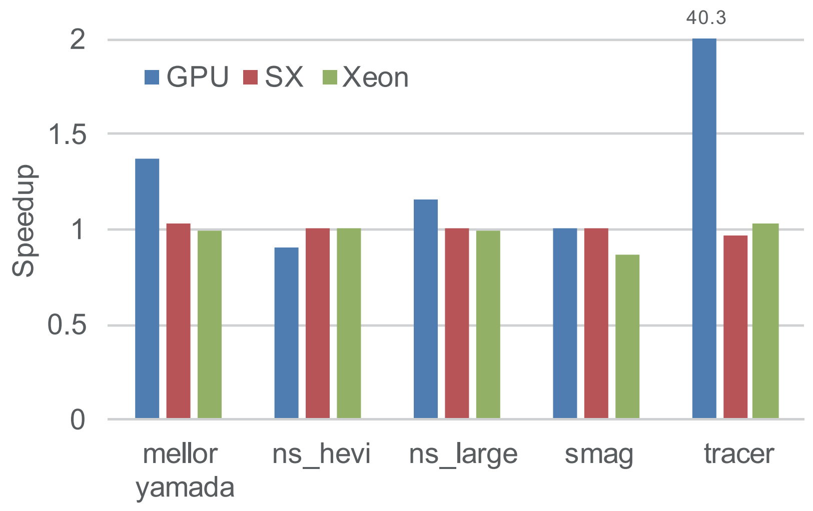

Figure 7: The effects of all code modifications on various platforms.

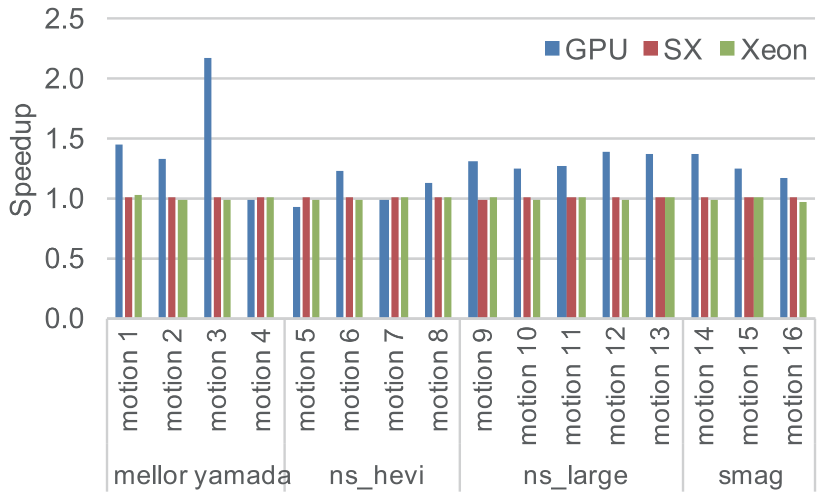

Figure 8: The effects of the loop-invariant code motions on various platforms.

4.4.1 Performance Portability of the Loop-invariant Motions

In order to further analyze the effects of each code modification, the performance is examined in more detail. Figure 8 shows the performance improvement by the loop-invariant code motion on three platforms. The x-axis indicates the loop nests, to which the code motion is applied by the Xevolver framework. Note that the number of modified loops is different from the numbers in Table 1 because all of the modified loops are not always executed due to the conditional branches. The y-axis indicates the performance of each loop nest normalized by that of the first version. The figure shows that the performance on the GPU is improved in most cases. This is because the number of parallelizable loop iterations increases more than a hundred times by the translation and results in efficient parallel execution. On the other hand, in a few cases, the GPU performance is unchanged or becomes even worse, because there is a trade off between increasing loop parallelism and increasing the computation amount. If the loop parallelism is not significantly increased, an increase in the

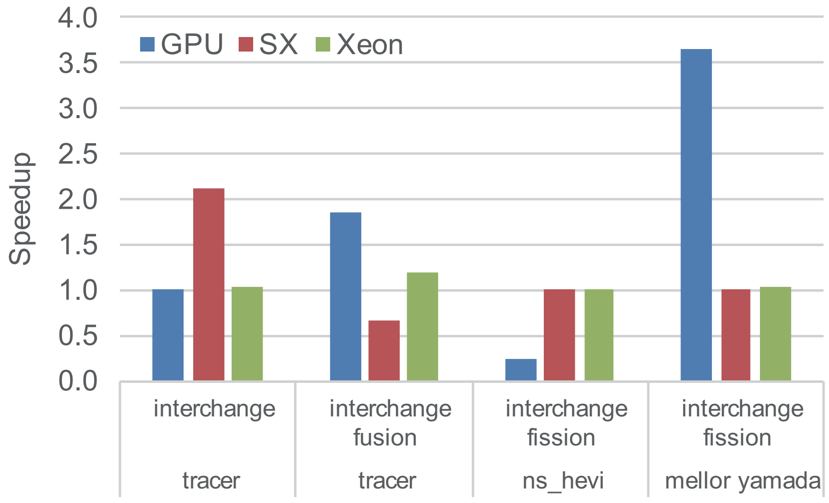

Figure 9: The effects of the loop interchange and its variants.

computation amount by the loop-invariant code motion cannot be amortized. This example clearly indicates that it is important to avoid directly modifying a code for such a code optimization.

Figure 8 also shows that performances of the other platforms do not change in any cases. This is because the compilers of each platform, the SX and Intel compilers, could detect the loop-invariant code, and then move it back to the original position in order to avoid redundant calculations.

Furthermore, Figure 8 shows that the performance improvements of loop nests in the same kernel are similar. The variance of the speedup ratios of loop nests in each kernel is less than 0.18. For example, the speedup ratios of all loop nests in the *ns_large* kernel range from 25% to 40%. This is because the loop nests in the same kernel are likely to have the same or similar structures. For those loop nests, the same code modifications become effective. As a result, the similar speedup can be obtained for the loop nests in each kernel. From these results, it is clarified that the use of the code translation framework is effective for such repetitive code modifications in the same kernel even though a certain labor is necessary for defining a pair of a new translation rule and a new directive.

4.4.2 Performance Portability of the Loop Interchanges

Figure 9 shows how the loop interchange and its variants affect the performance. The x-axis indicates the code modification applied to various kernels according to the translation rule by the Xevolver framework. The y-axis indicates the performance of each code modification normalized by that of the first version. This figure shows that the effect of each code modification is completely different. Different from the case of the loop-invariant code motion, the speedup ratio strongly depends on the combination of a kernel and a platform. In the case of the loop interchange for the *tracer* kernel, the speedup ratio on SX-ACE reaches about 2.1 times. In the case of the combination of the loop interchange and the loop fusion for the *tracer* kernel, the performances of GPU and LX increase, and the performance on SX-ACE decreases. In the case of the combination of the loop interchange and the loop fission for the *ns_hevi* kernel, the performance on the GPU decreases by more than 75%. On the other hand, in the case of the *mellor_yamada* kernel, the speedup ratio of GPU reaches 3.6 times even by applying the same code modification. These results clarify that an appropriate selection of the translation by the framework is important for high performance portability because the same code modification does not always achieve high performance on all platforms.

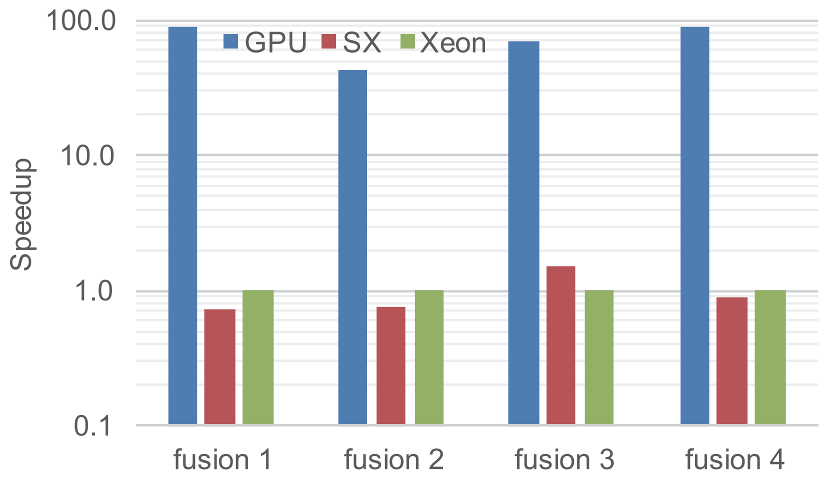

Figure 10: The effects of the loop fusions in the *tracer* kernel.

4.4.3 Performance Portability of the Loop Fusions

Figure 10 shows the effects of the combination of the loop fusion with the loop interchange on the performance of the *tracer* kernel. The x-axis indicates the loop nests, to which the combination of the loop fusion and the loop interchange is applied. The y-axis indicates the speedup ratio. This figure shows that the performances on the GPU drastically increase. The speedup ratio reaches about 91 times at maximum. This is because the replacement of a temporal array with a temporal scalar variable is applied in addition to the loop fusion and the loop interchange. Because the access to the temporal array prevents parallelization for GPU, the replacement leads to effective parallel execution using more threads on GPU, and results in an incredible speedup. However, the code modification causes performance degradation on SX-ACE. This is because the concentrated accesses to the temporal scalar variable prevent vectorization. These results also indicate that only appropriate translation considering the target platform should be applied to achieve high performance portability among multiple platforms. Therefore, it is clarified that the Xevolver framework is suitable to achieve high performance portability among multiple platforms.

5 Conclusions

This paper shows translation of a large legacy HPC application code for an accelerator platform in order to keep code maintainability and achieve high performance portability. To avoid drastic code modification to the original code, a code translation framework, *Xevolver*, is adopted in addition to employing a directive-based programming OpenACC. The Xevolver framework can replace code modifications necessary to achieve a high performance with pairs of a custom directive and a custom code translation rule. Because the translation rule is separately defined from the original code, the code modifications to the original code can be minimized to keep its code maintainability. Furthermore, by representing all code modifications as externally-defined translation rules and their corresponding directives, translation can easily be enabled or disabled. Therefore, by appropriately select whether the translation should be applied considering a target platform, high performance portability can be achieved.

Preliminary performance evaluation firstly shows that simply inserting OpenACC directives is not always enough to exploit the potential of an accelerator. The result clarifies that non-trivial code modifications are necessary even in a directive-based programming. In addition, the code modifications sometimes degrade the performance of other platforms and leads to low performance portability. Then, the Xevolver framework avoids such code modifications by defining pairs of a translation rule and a directive in order to keep the code maintainability. Finally, it is clarified that our approach can achieve high performance portability by selectively applying translation with the Xevolver framework though the performance evaluation.

Our future work includes the cooperation of translation and auto-tuning. An application user may not judge which translation should be applied to a given application for a particular target system when the number of pre-defined and custom translation rules is large. In such a situation, auto-tuning techniques may be effective to find appropriate translation from a translation rule database.

Acknowledgment

This research was partially supported by Core Research of Evolutional Science and Technology of Japan Science and Technology Agency (JST CREST) "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems." This research uses the SX-ACE and LX systems in Cyberscience Center of Tohoku University.

References

[1] Extensible markup language (XML) 1.1 (second edition). https://www.w3.org/TR/xml11/.

[2] OpenACC directives for accelerometers. http://www.openacc-standard.org/.

[3] OpenCL - the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/.

[4] The OpenMP API specification for parallel programming. http://openmp.org/.

[5] Top500 supercomputer sites. http://www.top500.org/.

[6] XSL transformations (XSLT) version 2.0. https://www.w3.org/TR/xslt20/.

[7] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[8] Kazuhiko Komatsu, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A compiler-assisted OpenMP migration method based on automatic parallelizing information. In *Proceedings of 29th International Supercomputing Conference*, volume 8488, pages 450–459. 2014.

[9] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of OpenCL programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[11] Dan Quinlan and Chunhua Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[12] Reiji Suda, Hiroyuki Takizawa, and Shoichi Hirasawa. Xevtgen: fortran code transformer generator for high performance scientific codes. In *Proceedings of the Third International Symposium on Computing and Networking*, pages 528–534, Dec 2015.

[13] Keiko Takahashi, Akira Azami, Yuki Tochihara, Yoshiyuki Kubo, Ken'ichi Itakura, Koji Goto, Kenryo Kataumi, Hiroshi Takahara, Yoko Isobe, Satoru Okura, Hiromitsu Fuchigami, Jun-ichi Yamamoto, Toshifumi Takei, Yoshinori Tsuda, and Kunihiko Watanabe. World-highest resolution global atmospheric model and its performance on the Earth Simulator. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011*, pages 1–12, Nov 2011.

[14] Keiko Takahashi, Ryo Onishi, Takeshi Sugimura, Yuya Baba, Koji Goto, and Hiromitsu Fuchigami. Seamless simulations in climate variability and HPC. In Michael Resch, Sabine Roller, Katharina Benkert, Martin Galle, Wolfgang Bez, and Hiroaki Kobayashi, editors, *High Performance Computing on Vector Systems 2009*, pages 199–219. Springer Berlin Heidelberg, 2010.

[15] Hiroyuki Takizawa, Shoichi Hirasawa, Yasuharu Hayashi, Ryusuke Egawa, and Hiroaki Kobayashi. Xevolver: An XML-based code translation framework for supporting HPC application migration. In *21st International Conference on High Performance Computing (HiPC 2014)*, pages 1–11, Dec 2014.

[16] Qing Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Journal of Software - Practice & Expererience*, 42(6):675–706, June 2012.