論文 / 著書情報
Article / Book Information

題目(和文)	メニーコアプロセッサにおける高性能計算のための高レベル抽象化
Title(English)	High-level Abstractions for High Performance Computing on Many-core Processors
著者(和文)	星野哲也
Author(English)	Tetsuya Hoshino
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第10941号, 授与年月日:2018年9月20日, 学位の種別:課程博士, 審査員:松岡 聡,増原 英彦,遠藤 敏夫,額田 彰,横田 理央
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第10941号, Conferred date:2018/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Category(English)	Doctoral Thesis
種別(和文)	要約
Type(English)	Outline

High-level Abstractions for High Performance Computing on Many-core Processors (メニーコアプロセッサにおける高性能計算のための高レベル抽象化)

by

Tetsuya Hoshino
hoshino@matsulab.is.titech.ac.jp

Submitted to the
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Science

June 26, 2018

Abstract

Over the last decade, processor performance has mainly been improved by increasing the number of cores, and the high-performance computing field is correspondingly shifting from multi- to many-core processors. Many-core processors can combine great computational power with excellent energy efficiency; however, programming them effectively takes significant effort. To reduce the difficulty of programming for many-core processors, several high-level programming models have been proposed.

One approach, commonly used in scientific applications, is to rely on compiler-directive-based language extensions such as OpenMP and OpenACC. These provide sets of directives, namely language extensions that abstract away the architectural characteristics of many-core processors. However, although these abstractions allow compiler-directive-based approaches to simplify the writing of parallel programs for many-core processors, they cannot bring out the best possible performance in many-core processors. This is because some of the essential difficulties in programming for many-core processors still remain, such as load balancing and vectorization, which are intimately dependent on the algorithms and data structures used by the target application.

As an alternative, many domain-specific frameworks have also been proposed. Unlike compiler-directive-based approaches, domain-specific approaches can overcome the above issues by focusing on the specific characteristics of their target applications. However, this also means they lack versatility, and a separate framework must be developed for each application domain.

To address these difficulties, we consider both compiler-directive-based and domain-specific approaches in this dissertation. To evaluate the performance, productivity, and portability of the OpenACC directive-based approach, we port and optimize both kernel benchmarks and real-world application code to both OpenACC and CUDA. We also use the UPACS com-

putational fluid dynamics (CFD) application as a case study. We find that data structure transformations, which require structural changes to the code, even with a directive-based approach, can effectively create efficient vectorized versions of applications.

We also propose extensions to the OpenACC directives for creating efficient vectorizations. The extended directives provide data structure abstraction, which is very important for efficient vectorization. We then develop a source-to-source translator that supports the proposed directives and evaluate it using two real-world applications, namely UPACS and CCS-QCD. The results show that the performance improved by 23% and 20% compared with the baseline for UPACS and CCS-QCD, respectively.

Moving on to domain-specific approaches, we enhance the $\mathcal{H}$ACApK open-source $\mathcal{H}$-matrix framework, originally developed for symmetric multiprocessing clusters, to work well on many-core processors. In particular, we propose many-core-focused load-balancing-aware parallel adaptive cross approximation (ACA) algorithms for working with $\mathcal{H}$-matrices. Existing parallel algorithms apply the ACA process to each of the $\mathcal{H}$-matrix 's submatrices independently. However, since the computational load of each ACA process depends on the sub-matrix 's size, which is undefined before the ACA process is applied, it is difficult to balance the load. Because our algorithms are implemented as part of the $\mathcal{H}$ACApK framework, framework users can reap the benefits of these changes without even realizing it. We then evaluate the proposed algorithms using boundary element method (BEM) problems on both an NVIDIA Tesla P100 GPU (P100) and an Intel Xeon Broadwell (BDW) CPU. The results show that our algorithms improved performance in all GPU cases.

In addition, we propose a design of framework for abstracting the vectorization process. We then adopt this framework design to the BEM-BB framework, a $\mathcal{H}$ACApK wrapper framework for BEM analyses. We then evaluate the adapted framework using two BEM problems, namely, static electric field analysis with a perfect conductor and with a dielectric, on BDW and Intel Xeon Phi Knights Landing (KNL) processors. The results show that this approach can offer good vectorization performance while requiring little vectorization knowledge. Specifically, in perfect conductor analyses conducted using $\mathcal{H}$-matrices, the new framework improved performance by 2.22 and 4.34 times compared with the original BEM-BB framework on the BDW and KNL processors, respectively.

In summary, we hide the difficulties involved in developing efficient pro-

grams for many-core processors by proposing abstractions based on both compiler-directive-based and domain-specific approaches. These abstractions also enable programs to be ported to different processors with different numbers of cores and vector lengths without sacrificing performance. Even though the optimal data layout, load-balancing algorithm and vector length may differ among processors, our proposals enable the most suitable approaches to be easily selected. We therefore believe that they are one possible answer to the multi- to many-core paradigm shift.

Acknowledgement

Paper List

Conference and Workshop (reviewed)

[1] Tetsuya Hoshino, Akihiro Ida, Toshihiro Hanawa and Kengo Nakajima, “ Design of Parallel BEM Analyses Framework for SIMD Processors ”, The International Conference on Computational Science 2018 (ICCS 2018), Wuxi, China (June 11-13, 2018) (Accepted)

[2] T. Hoshino, N. Maruyama and S. Matsuoka, ”A Directive-Based Data Layout Abstraction for Performance Portability of OpenACC Applications,” 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, 2016, pp. 1147-1154.

[3] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka, “ An OpenACC extension for data layout transformation ”, In Proceedings of the First Workshop on Accelerator Programming using Directives (WACCPD ’14), pp.12-18, New Orleans, USA, November, 2014

[4] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, Ryoji Takaki, “ CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application ”, In Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013), pp.136-143, Delft, the Netherlands, May, 2013

Conference and Workshop (unreviewed)

[5] Tetsuya Hoshino, Akihiro Ida, Toshihiro Hanawa, and Kengo Nakajima,“ Performance Evaluations and Optimizations of H-Matrices for Many-Core Processors ”, SIAM Conference on Parallel Processing for Scientific Computing 2018(SIAM PP 18), Tokyo, Japan (March 2018).

[6] 星野哲也、伊田明宏、塙敏博、中島研吾:階層型行列法ライブラリ$\mathcal{H}$ACApKを用いたアプリケーションのメニーコア向け最適化, 情報処理学会研究報告 (HPC-160), 2017-07-19

[7] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. A Directive-Based Data Layout Auto-Tuning for OpenACC Applications (SIAM CSE17) Mar. 2017, Atlanta, USA

[8] 星野哲也、大島聡史、塙敏博、中島研吾、伊田明宏：OpenACC を用いた ICCG 法ソルバーの Pascal GPU における性能評価, 情報処理学会研究報告 (HPC-158), 2017-03-01

[9] 星野哲也、松岡聡：圧縮性流体解析プログラムの OpenACC による高速化, 情報処理学会研究報告 (HPC-153), 2016-02-23

[10] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. Evaluations of Directive Based Programming Model for GPUs and Extensions for Performance Portability (SIAM CSE15) Mar. 2015, UT, USA

[11] 星野哲也, 丸山直也, 松岡聡：OpenACC ディレクティブ拡張によるデータレイアウト最適化 情報処理学会研究報告, (HPC-145), 2014-07-21

Poster

[12] Tetsuya Hoshino, Satoshi Ohshima, Toshihiro Hanawa, Kengo Nakajima, Akihiro Ida. Pascal vs KNL: Performance Evaluation with ICCG solver (ISC 17) June 2017, Frankfurt, Germany

[13] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. An OpenACC extension for data layout transformation.(GTC 15) Mar. 2015, CA, USA

Contents

1 Introduction **1**
- 1.1 Computational Environments for the Many-Core Era 1
- 1.2 High-Level Programming Models for Many-Core Processors . . 3
- 1.3 Problem Statement 4
- 1.4 Approach 5
- 1.5 Contributions 6
- 1.6 Thesis Outline 8

2 Background **9**
- 2.1 Many-core Processors 9
  - 2.1.1 GPUs 9
- 2.2 Programing Models for Many-core Processors 10
  - 2.2.1 CUDA 10
  - 2.2.2 OpenACC 12
  - 2.2.3 Comparisons of Programing Models 13

3 Evaluations of Directive-based Programming Models for Many-core Processors **17**
- 3.1 Introduction and Motivation 17
- 3.2 Evaluation Methodology 18
  - 3.2.1 Kernel Benchmarks 19
  - 3.2.2 Application Case Study 20
- 3.3 Optimizations 24
  - 3.3.1 Matrix Multiplication 24
  - 3.3.2 Shared Memory Blocking 24
  - 3.3.3 7-point Stencil 25
  - 3.3.4 UPACS 25
- 3.4 Performance Evaluation 27

3.4.1 Matrix Multiplication . . . 28
3.4.2 7-point Stencil . . . 28
3.4.3 UPACS . . . 30
3.5 Related Work . . . 31
3.6 Chapter summary . . . 32

4 Extensions of OpenACC Directives for Efficient Vectorizations 34
4.1 Introduction . . . 34
4.2 Background . . . 35
4.2.1 Data Layout and Performance Characteristics of Devices 35
4.3 Related Work . . . 36
4.4 Abstraction of the data layout . . . 36
4.4.1 Proposal of the transformation directive . . . 37
4.5 Implementation . . . 40
4.5.1 Implementation of the source-to-source translator . . . 40
4.5.2 Runtime Library . . . 41
4.6 Evaluations of Translator . . . 41
4.6.1 Micro benchmark . . . 42
4.6.2 UPACS_turbo . . . 44
4.6.3 CCS-QCD . . . 45
4.7 Chapter summary . . . 46

5 Framework Design for Many-core Processors 54
5.1 Introduction . . . 54
5.2 BEM-BB framework . . . 55
5.3 Framework Design for SIMD Vectorization with OpenMP SIMD Directives . . . 58
5.3.1 New interface definition for compiler vectorization . . . 59
5.3.2 Using the framework . . . 60
5.4 Numerical Evaluations . . . 63
5.4.1 Test Model and Processors . . . 63
5.4.2 Hand Tuning Using OpenMP SIMD Directives . . . 65
5.5 Related Work . . . 68
5.6 Chapter summary . . . 68

6 Conclusion 70

List of Figures

1.1 40 years of microprocessor trend (cited from [31]) 2

2.1 Full chip block diagram of the Pascal GP100 architecture with 60 SM Units (cited from NVIDIA Pascal whitepaper[28]) . . . 11
2.2 Diagram of the Pascal GP100 Streaming Multiprocessor (SM) Unit (cited from NVIDIA Pascal whitepaper[28]) 11
2.3 CUDA Fortran matrix multiplication using shared memory code 15
2.4 OpenACC directives 16
2.5 OpenACC matrix multiplication 16

3.1 OpenACC baseline implementation of matrix multiplication . 20
3.2 OpenACC baseline implementation of 7-point stencil 21
3.3 UPACS computation phases, illustrated in 2-D for simplicity. Each filled circle or rectangle represents a grid point or cell face updated by the stencil. Arrows are used to express read-write data dependencies from source to destination. 27
3.4 Performance of matrix multiplication of 2048^2 matrices. 29
3.5 Throughput of 7-point stencil with 256^3 grids. 29
3.6 UPACS performance relative to the CPU performance. Non-applicable optimizations are left blank. 30
3.7 Elapsed time of each UPACS phase. Non-applicable optimizations are left blank. 31

4.1 The specification of *acc transform transpose*. 38
4.2 Data Layout Transformation Process 38
4.3 The usage of the *present_transform* directive. 39
4.4 The overview of our translator. Our translator consists of source-to-source translator and runtime libraries. 48

4.5 Data layout transformation process. This example transforms the target array A to B that has optimal data layout for device process. 48
4.6 An example of acc transform directive. 49
4.7 The output of our translator using 4.6as an input. 49
4.8 Runtime transposition kernel bandwidth (without PCIe transfer). 50
4.9 The elapsed time of the PCIe transfer time + transposition kernel time. 50
4.10 Convection (left) and viscosity (right) computation phases, illustrated in 2-D for simplicity. Each filled circle or rectangle represents a grid point or cell face updated by the stencil. Arrows are used to express read-write data dependencies from source to destination. 51
4.11 The target array of structures used in UPACS_turbo convection phase. Each structure defined at cell face illustrated as a filled circle in Fig.4.10. 51
4.12 Elapsed time of the convection and viscosity phases of UPACS 52
4.13 Applying *loop collapse* extension to a loop nest of CCS-QCD. . 52
4.14 The performance of CCS-QCD 53
4.15 Elapsed time of CCS-QCD 53

5.1 The design of BEM-BB framework 57
5.2 Parallel generation of coefficient dense matrix and $\mathcal{H}$-matrix. . 58
5.3 An interface of a user-defined function to calculate the i-th row and the j-th column element of the coefficient matrix. The function arguments after i and j are used as input variable of the calculation. 59
5.4 User-defined function caller for dense matrix. Here, a(j,i) is a coefficient dense matrix. The ranges of i and j are assigned to each thread adequately. 60
5.5 User-defined function caller for sub-matrix of $\mathcal{H}$-matrix. Here, HACApK_entry_ij is a wrapper function of ppohBEM_matrix_element_ij. The structure st_bemv contains the variables required as arguments of the user-defined function. 61

5.6 New interface for data access. The former arguments are the same as ppohBEM_matrix_element_ij. The latter arguments are the scalar variables used in vectorize_func. The number of arguments depends on the target application. 62
5.7 New calculation interface. This function should be called after the set_args subroutine and vectorized. All arguments of this function should have intent(in) attribute. 62
5.8 User-defined function using new interface caller for dense matrix. 63
5.9 Dummy function of user-defined function. Although the function is not used in the framework, users are required to implement this function correctly. 64
5.10 The users program automatically transformed at the compile time. 65
5.11 Solving perfect conductor problem using KNL processor 67
5.12 Solving dielectric problem using KNL proccesor 67
5.13 Solving perfect conductor problem using BDW processor . . . 67
5.14 Solving dielectric problem using BDW processor 67

List of Tables

2.1 OpenACC Limitations 14

3.1 Details of each major phase in UPACS. 23
3.2 Number of modified lines of code in matrix multiplication. . . 28
3.3 Details of each major phase in UPACS. 29
3.4 Details of each major phase in UPACS. 31

4.1 Evaluation environment 42
4.2 All target data layout and transformation rule 46

5.1 Processor Specifications 65
5.2 The elapsed times of coefficient generation component of original implementation (H1) and implementation of proposed framework (H7) 68

Chapter 1

Introduction

1.1 Computational Environments for the Many-Core Era

Scientific numerical analyses, such as earthquake, weather prediction, and engineering simulations, have now become increasingly common due to the increased computational resources available. These improvements originate from progress in microminiaturization technology. Figure 1.1 shows the microprocessor trends over the last 40 years based on data produced by Kari Rupp [31]. This graph shows that even though the number of transistors is continuing to increase linearly, both the single-thread performance and frequency have become saturated. Instead, the number of logical cores is increasing, indicating that the improvements in processor performance in recent years originate from increases in the number of cores. Herein, we will call processors comprising many cores as many-core processors.

Increasing numbers of supercomputers are powered by many-core processors due to their high energy efficiency, which is one of the key factors for current supercomputers. The top 10 places in November 2017 's Green500 list [19], which is a list of the world 's greenest supercomputers, were occupied by machines based on many-core processors. The top Japanese supercomputers also use many-core processors. For example, Oakforest-PACS [23], which is installed at The University of Tokyo, comprises 8,208 Intel Xeon Phi Knights Landing (KNL) processors [34], while TSUBAME3 [40] has 2,160 NVIDIA Tesla P100 GPUs (P100s) [28]. The KNL and P100 processors, the main targets of this dissertation, have 68 physical cores and 56 streaming

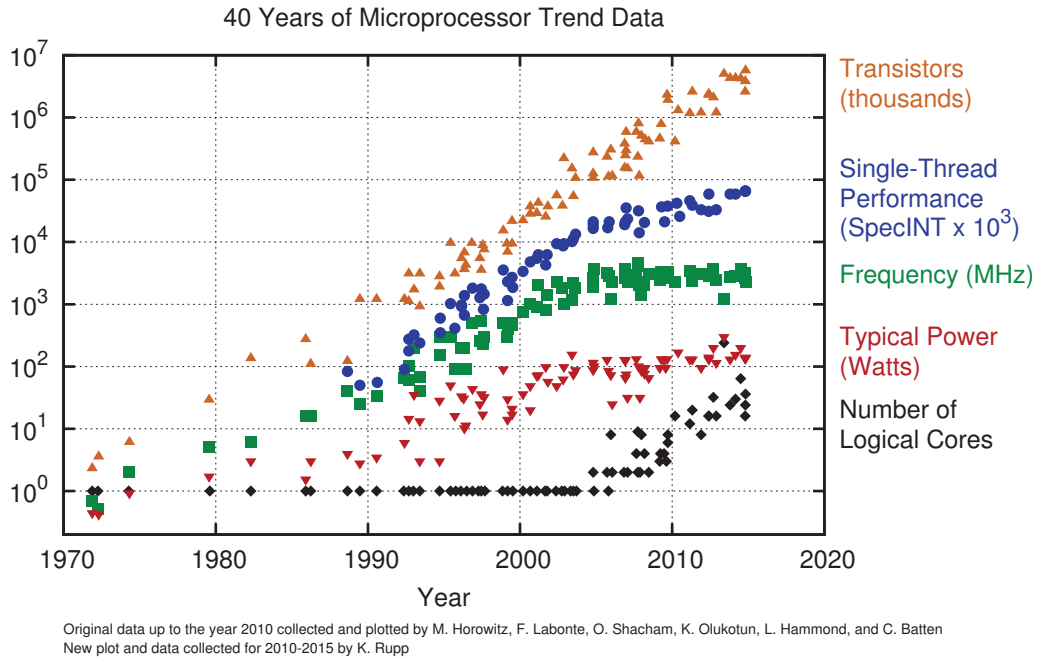

Figure 1.1: 40 years of microprocessor trend (cited from [31])

multiprocessors, respectively.

In addition to having large numbers of cores, the cores in many-core processors also generally include vector execution units. For example, the Intel Xeon Phi series has 512-bit single instruction, multiple data (SIMD) units, while NVIDIA GPUs have warps, which are execution units that handle groups of 32 threads. In other words, two-level parallelization is required to take full advantage of the computational resources of many-core processors. Furthermore, all current many-core processors have their own memory, which has relatively small capacity but high bandwidth, in addition to the main memory. For example, KNL has 16 GB of MCDRAM, which can also be used as a last-level cache, while P100 has 16 GB of HBM2 memory. The memory hierarchies of many-core processors are thus becoming increasingly complex.

1.2 High-Level Programming Models for Many-Core Processors

To obtain the best possible performance from many-core processors, application programmers have had to understand their characteristics and optimize their applications accordingly. Although many scientific applications [45, 1, 33] have been optimized for many-core processors using explicit parallel programming models, such as CUDA [22] and OpenCL [35], these models require significant programmer effort to implement.

To reduce this burden, several high-level implicit parallel programming models and frameworks have been proposed for many-core processors [42, 18, 5, 10]. In particular, compiler directive-based parallel programming models allow for considerable simplification and increases in productivity when hybridizing existing applications. These models abstract the characteristic processes of many-core processors, such as managing the data in their local memory and parallelizing tasks for vector execution units, so that they can be implicitly handled by the compiler.

Among these compiler-directive-based models, OpenMP [39] and OpenACC [27] are widely used in scientific computing, where parallelism often appears in the form of regular looping constructs such as Fortran DO loops. Although OpenMP 's directives were originally defined for multi-core processors, additional directives aimed at many-core processors are included in OpenMP 4.x and later. In contrast, OpenACC is a new compiler directive specification that allows for annotation of the computation and data regions that are offloaded to accelerators such as GPUs. Unlike CUDA and OpenCL, which require more explicit computation and data management, porting legacy CPU-based applications to OpenACC only requires code annotations and does not involve any significant structural changes to the original code.

Domain-specific frameworks [20, 15, 14] are a powerful alternative approach to high-performance computing on many-core processors. In contrast to directive-based approaches, domain-specific methods abstract the computational patterns of specific applications. For example, Physis [20] is a domain-specific framework for computing stencils, which often appear in scientific applications such as computational fluid dynamics (CFD). Physis abstracts the communication and computation patterns involved in stencil computation and applies auto-tuning to GPU-accelerated clusters.

1.3 Problem Statement

Although compiler-directive-based programming models greatly simplify many programming tasks, they are not as flexible as explicit programming models. For example, both CUDA and OpenCL provide fine-grained synchronization primitives such as thread synchronization, whereas OpenACC does not. Efficient application implementations may depend on the ability to manage on-chip memory in software, which can be done directly with CUDA and OpenCL, but not with OpenACC. These differences may prevent applications from taking full advantage of the available architectural resources, potentially resulting in substantially inferior performance when compared with highly tuned CUDA or OpenCL code.

In addition, even though compiler-directive-based programming models simplify the writing of parallel programs, these simplifications do not always bring out the best possible performance in many-core processors. In this dissertation, we focus particularly on the following issues.

Load balancing

Balancing computational loads is an important issue for using many-core processors efficiently. OpenMP and OpenACC balance loads using dynamic loop scheduling functions. However, such dynamic scheduling methods assume that the number of loads to be allocated is much larger than the number of threads, so this approach is not always helpful for many-core processors, which have large numbers of threads. As a result, achieving effective load balancing often requires both significant structural changes to the algorithms involved and an understanding of many-core processors, despite the use of compiler-directive-based programming.

Vectorization

As discussed above, vector execution units are important elements of current many-core processors. Even though vectorization can be abstracted by directives, such as the SIMD directive in OpenMP or the VECTOR clause of the LOOP directive in OpenACC, it can still be difficult to vectorize programs effectively. This is because efficient vectorization requires two constraints to be met: (1) there should be no data dependencies among the elements of the target vector and (2) vector elements should be stored contiguously. In addition, generating efficiently vectorized code using compiler directives requires the compiler

to identify the code as being vectorizable. In other words, programmers are often required to change the data structures, potentially affecting the whole program, based on their knowledge of the way the compiler operates.

Ideally, high-level programming models should be able to implicitly perform any algorithmic or data structure changes that are required but it is difficult to handle such abstractions using a general directive-based approach because they depend intimately on the target application. In contrast, domain-specific approaches can deal with these issues but they are obviously less versatile. Furthermore, a different domain-specific framework would be required for each application domain.

1.4 Approach

The aim of this dissertation is to hide the difficulties of working with many-core processors using high-level programming models while attempting to mitigate the issues discussed above. In particular, we discuss both directive-based and domain-specific approaches.

First, we consider the OpenACC directive-based approach. To evaluate its performance, productivity, and portability, we port and optimize both kernel benchmarks and real-world application code for use in case studies comparing OpenACC and CUDA. We use matrix multiplication and 3D stencil kernel benchmarks to compare their performance on NVIDIA GPUs, as well as conduct an application study using the UPACS CFD application [44]. Then, we extend the OpenACC directives to accelerate vectorization by abstracting the data structures involved. We develop a source-to-source translator using the ROSE compiler infrastructure [32] and evaluate the results using several applications.

Moving on to domain-specific approaches, we enhance the BEM-BB boundary element method (BEM) framework for use with many-core processors. In particular, we propose load-balancing-aware algorithms for working with $\mathcal{H}$-matrices using adaptive cross approximation (ACA) on many-core processors. In addition, we propose and design a framework for abstracting the vectorization process and evaluate its performance on electrostatic field problems. Even though this framework targets BEM applications, the underlying design could be re-used by other frameworks.

1.5 Contributions

The main contributions of this dissertation are as follows.

1. **The performance of directive-based programming models for many-core processors is evaluated using application case studies.**

   We evaluate OpenACC using several applications. The results of experiments using NVIDIA GPUs with multiple production OpenACC compilers show that the OpenACC versions of micro-benchmarks performed on average approximately half as well as the corresponding CUDA versions when the same manual optimizations were applied, although the performance could be as high as 98% depending on the compiler and benchmark. A similar average performance trend was also observed for the UPACS application, although here the OpenACC performance was at most 64% of that of the CUDA version. We also found that the highly tuned versions exhibited larger performance gaps, as some of the optimizations, particularly those based on shared memory, could not be applied due to the limitations of the current OpenACC version 's programming interface. In particular, the highly tuned CUDA versions of matrix multiplication, a 7-point stencil, and UPACS were faster than the best OpenACC results by factors of 2.7, 1.2, and 2.4, respectively.

2. **The OpenACC directives are extended to handle vectorization.**

   We propose a set of OpenACC directive extensions that allow programmers to declaratively adapt the data structures, which are very important for efficient vectorization, to suit the target device. We also implement and evaluate a source-to-source translator that automatically generates data structures optimized for a given target processor based on the information supplied by the extended directives. To evaluate our proposal, we adapted two existing real-world OpenACC applications, UPACS and CCS-QCD to use our proposed directives. Applying our prototype source-to-source translator to the extensions yielded performance improvements of 23% and 20%, respectively, compared with the baseline, similar to the performance of manually tuned versions of the applications.

3. **Many-core-optimized load-balancing-aware algorithms are proposed for working with $\mathcal{H}$-matrices using ACA.**

   We propose load-balancing-aware algorithms that take a fine-grained parallelization approach to handling $\mathcal{H}$-matrices using ACA. We also propose a method of storing $\mathcal{H}$-matrices for use with the algorithms. Next, we implement these algorithms for GPUs as part of $\mathcal{H}$ACApK [14, 15], an open-source $\mathcal{H}$-matrix library that was originally developed for symmetric multiprocessing clusters based on a locality-aware approach. We then compare the performance of the proposed load-balancing-aware algorithms with those of existing locality-aware algorithms using electrostatic field problems on P100 and Intel Xeon Broadwell (BDW) platforms. The results show that the load-balancing-aware algorithms were able to achieve up to 12.9 times the performance on the P100. To investigate the effect of the $\mathcal{H}$-matrix storage method, we also evaluate the performance of $\mathcal{H}$-matrix-vector multiplication (HMVM), proposing an optimized version of HMVM for GPUs. The proposed implementation achieved a speed-up of up to 3.3 times on the P100.

4. **A Framework is proposed for SIMD processors.**

   We propose and design a framework for abstracting SIMD-related issues. Next, we adapt the BEM-BB framework to include SIMD vectorization using the proposed design. We then evaluate the proposed framework 's performance by solving two problems, namely static electric field analysis with a perfect conductor and with a dielectric, which require different user-defined functions for BEM-BB, on BDW and KNL processors. We compare the performance of the proposed framework with those of the original framework and of hand-tuned user functions. The results show that the proposed framework achieved performance improvements of 2.22 and 4.34 times compared with the original framework for the BDW processor and the KNL processor, respectively. Furthermore, the results demonstrate that the framework 's performance was comparable to that achieved by hand-tuned programs.

In summary, this dissertation presents both directive-based and domain-specific approaches to hide the difficulties of programming on many-core processors. We propose directive-based and domain-specific methods for handling vectorization and implement them using a source-to-source translator

and framework, respectively. We then evaluate the performance of the proposed implementations using real-world applications.

1.6 Thesis Outline

The rest of this dissertation is organized as follows.

chapter2:

In Chapter 2, we introduce background materials about this dissertation. First, we give overviews about the architectures of P100 and KNL as many-core processors. Then, we focus the abstractions of high-level programming models.

chapter3:

In Chapter 3, we evaluate the OpenACC as a directive-based programming model with kernel benchmarks and a real world application.

chapter4:

In Chapter 4, we propose a set of directive extensions of OpenACC for abstraction of the data layout.

chapter5:

In Chapter **??**, we introduce the load-balancing methodology for $\mathcal{H}$-matrices.

chapter6:

In Chapter 5, we propose a framework design to encapsulate the vectorizations.

chapter7:

Finally, we argue the conclusion and future work.

Chapter 2

Background

In this chapter, we describe the architectural characteristics of many-core processors and the abstractions of high-level programming models as the bases of discussions. In particular, we introduce the architecture of P100 and KNL, which are the main target processors in this dissertation. Also, we mainly focus on the abstraction of OpenACC, which is new specification for accelerator, such as GPUs.

2.1 Many-core Processors

2.1.1 GPUs

First, we describe the architectures of NVIDIA Tesla GPUs that are most widely used in HPC field. Pascal is the code name of the current mainstream of NVIDIA GPUs. The peak performance of NVIDIA Tesla P100 is 5.3 TFlops in double precision and the peak memory bandwidth is 732 GB/s. This excellent performances are produced by the parallel execution of many cores of P100.

To obtain a good performance of GPUs, it is important to understand the core hierarchy. Figure 2.1 shows overview of the Pascal architecture. P100 includes 56 Streaming Multiprocessors (SM, Fig. 2.2), which has 64 CUDA cores, that is, P100 has 3,584 cuda cores in total. Actually, P100 has 60 SMs shown in Fig. 2.1. But the number of available SMs is limited to obtain high yield ratio. Each SM can execute concurrently a large amount of threads up to 2,048. These threads are grouped, managed, scheduled, and executed

as sets of 32 threads called warps. All 32 threads within a warp execute the same instruction at the same time, this execution model is called SIMT (Single Instruction, Multiple Thread). Several warps are also grouped as a thread block and thread blocks are assigned to SMs in batches.

Load balancing

For load balancing, the core hierarchy is important. There are 56 SMs and each SM has 64 CUDA cores. Therefore, at least 56 thread blocks including 64 threads are required to fill the SMs. In addition, to hide the latency of memory accesses, GPUs provide very low cost context switch. Thus, the suitable number of threads are generally much larger than the number of CUDA cores. As a result, for load balancing, parallel algorithms that is scalable for more than 10 thousand of threads.

2.2 Programing Models for Many-core Processors

GPU programming requires using language extensions since no standard programming language natively supports GPU programming. CUDA is one of the most widely used extensions but it runs only on NVIDIA GPUs. It provides flexible programming constructs that allow fine-grained control of various functions of GPU. On the other hand, OpenACC is a new extension for C and Fortran. It provides directives like OpenMP for offloading of computation and data to accelerators. In this section, we describe a brief overview of OpenACC and CUDA.

2.2.1 CUDA

One of the most widely used programming models for GPUs is CUDA [22] developed by NVIDIA. It enables to program GPU applications with slightly extended C++ language. And CUDA Fortran [30] is developed by The Portland Group (PGI). Most of legacy applications are written in Fortran and target application of this thesis UPACS is also written in Fortran. Figure 2.3 shows the matrix multiplication program written in CUDA Fortran. Both CUDA program consists of host cords and device cords. Host cords run on host CPU and call device codes run on GPU. In CUDA Fortran, a subroutine

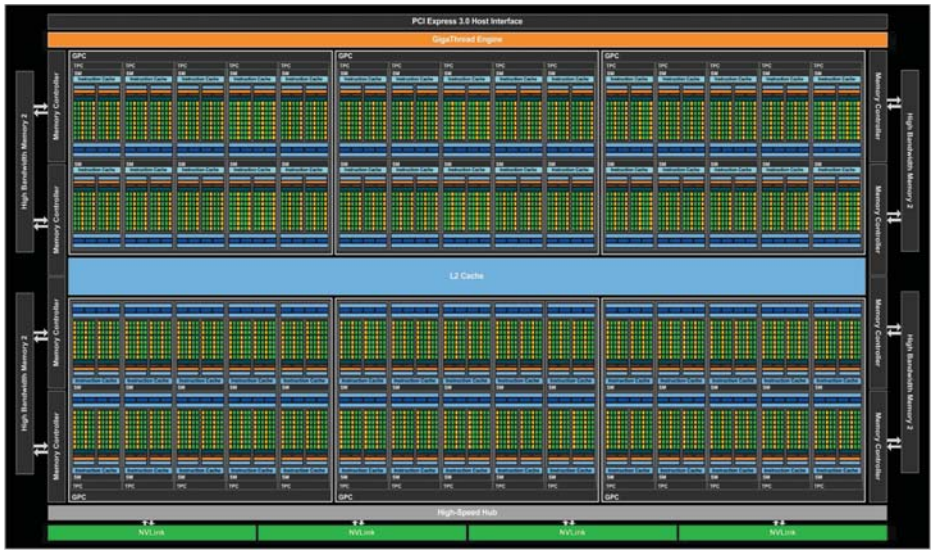

Figure 2.1: Full chip block diagram of the Pascal GP100 architecture with 60 SM Units (cited from NVIDIA Pascal whitepaper[28])

Figure 2.2: Diagram of the Pascal GP100 Streaming Multiprocessor (SM) Unit (cited from NVIDIA Pascal whitepaper[28])

added `attribute(global)` is a device code. In device code, programmer uses `blockidx`, `blockdim`, and `threadidx`, for example, to specify array indexes. Threads of CUDA programing model are managed with two levels of hierarchy called grid and thread block. Grid has two-dimensional (x, y) thread blocks and each thread block has three-dimensional (x, y, z) threads. Each thread blocks are executed on a single SMX which has user addressable on-chip memory called shared memory. Threads within a single thread block can communicate via shared memory and synchronize by calling `syncthreads()` function. The size of each dimension can be specified with `dim3` type in host codes (but grid doesn't have z-dimension so the third index of gird is always 1). Programmer appoints sizes of grid and thread block in `<<< >>>` and call a device code. GPU has a separate memory space, so programmer has to allocate memory for exclusive use of GPU by using `device` modifier and transfer data between host CPU memory and GPU device memory explicitly. Programmer has to be careful that data transfer speed is limited by PCI-express and it becomes the overhead.

2.2.2 OpenACC

OpenACC [27] is developed by NVIDIA and multiple compiler venders, PGI, CAPS, and Cray, in November 2011. Although CUDA only supports NVIDIA GPUs, OpenACC provides portability across operating systems, host CPUs and many core accelerators from multiple venders. It provides OpenMP-like directives to define compute and data regions in standard C and Fortran programs. Figure 2.4 shows syntax of the OpenACC directives, and figure 2.5 shows OpenACC matrix multiplication program. Compute regions can be defined with either `parallel` or `kernels` directives. The code inside a `parallel` or `kernels` region is offloaded and executed on an accelerator. C and Fortran programs with OpenACC directives can be compiled into a hybrid code by OpenACC-compliant compilers. Similar to OpenMP, in which the directives can be safely ignored, non-supporting compilers can still generate correct CPU-only code without interpretation of the directives. Similar to CUDA, a hybrid parallelism of SIMD and SPMD is used in OpenACC. The code inside the compute region is executed in parallel by multiple workers, each of which can also have SIMD operations. Similar to the thread blocks in CUDA, a group of workers is managed as a gang in OpenACC. The numbers of workers and gangs as well as the length of the SIMD operations can be configured through the `loop` directive and its options. Parallel

loops can also have cache directives, which specify that data objects should be kept on the highest level of the cache memory hierarchy for the body of the loop. Because OpenACC does not have synchronization primitives for the parallelized loop iterations, such as barrier in OpenMP or CUDA's `syncthreads()`, the directive can only be used for read-only data. The host and accelerator memory spaces are assumed to be separate, and the data objects used in the compute regions must be resident on the accelerator memory. However, unlike CUDA, they are automatically copied between the host and accelerator memories as necessary by the compilers such that their coherency is obtained at the region boundaries. Explicit control of the data transfer is also possible with directive options such as `copyin`, `copyout`, and `present`.

2.2.3 Comparisons of Programing Models

Productivity

CUDA has been the most widely used programming interface for scientific computing on GPUs, but its explicit, low-level programming abstraction results in relatively low programming productivity. Porting of existing CPU-based programs to CUDA requires identification of the bottleneck regions, which must then be rewritten into CUDA kernel code, which often causes significant structural changes in the original code. In contrast, although OpenACC still requires the programmer to identify the bottleneck regions, the original code can be reused with much fewer changes than required with CUDA, because the minimum porting requirement is annotation of the regions with the OpenACC directives. This simplification is particularly important when porting large legacy applications.

Portability and Performance

Unlike CUDA, OpenACC is designed to be portable across devices from multiple vendors. However, this design decision limits the use of vendor-specific architectural features. For example, OpenACC does not have software-addressable on-chip memory, which is available as shared memory in CUDA. The cache directive may be used to fetch data to the shared memory, but the lack of synchronization with respect to the shared memory effectively limits the use of this directive to read-only data. This limitation prohibits

Table 2.1: OpenACC Limitations

	OpenACC	CUDA
Vendor-specific architectural features	Depends	✓
Using structure including pointer member	N/A	✓
Getting thread ID	N/A	✓
Atomic operation	N/A	✓

some manual code transformations for memory access optimizations, such as temporal blocking using the shared memory [21], leaving them completely to the compiler. The other limitations are listed in table 2.1. OpenACC doesn't allow using structure including pointer member. This is because deep-copy, it is very challenging for compiler's automatic analysis, is necessary when the structure is copied to accelerators. Most of real world applications use structure, so this limitation may reduce portability of them. Furthermore, OpenACC doesn't have atomic operation and cannot get thread ID. OpenMP has these operations and most of legacy applications have been already parallelized with OpenMP for multi-core CPUs. So when a application already parallelized with OpenMP is ported to OpenACC, these limitations prevent it.

```
attribute(global) subroutine matmul_func(a, b, c, n)
   integer, value :: n
   double precision, dimension(n,n) :: a, b, c
   double precision, shared, dimension(16, 16) :: a_shared, b_shared
   integer :: i, j, k1, k2
   double precision :: cc
   tx = threadidx%x
   ty = threadidx%y
   i = (blockidx%x-1) * 16 + threadidx%x
   j = (blockidx%y-1) * 16 + threadidx%y
   cc = 0.0
   do k2 = 1, n, 16
      a_shared(tx, ty) = a(i, k2+ty)
      b_shared(tx, ty) = b(k2+tx, j)
      do k1 = 1, 16
         cc = cc + a_shared(tx,k) * b_shared(k,ty)
      end do
      call syncthreads()
   end do
   c(i,j) = cc
end subroutine matmul_func

subroutine matmul(a, b, c, n)
   double precision, dimension(n,n) :: a, b, c
   integer :: i, j, k, n, stat
   double precision :: cc
   double precision, device, allocatable, dimension(:,:) :: a_dev, b_dev, c_dev
   type(dim3) :: dimGrid, dimBlock

   allocate(a_dev(n,n) b_dev(n,n), c_dev(n,n))
   a_dev(:,:) = a(:,:)
   b_dev(:,:) = b(:,:)
   dimGrid = dim3(n/16, n/16, 1)
   dimBlock = dim3(16, 16, 1)
   call matmul_func<<<dimGrid, dimBlock>>>(a, b, c, n)
   c(:,:) = c_dev(:,:)
end subroutine matmul
```

Figure 2.3: CUDA Fortran matrix multiplication using shared memory code

```
C
#pragma acc directive [clause]
     {
          // body
     }

Fortran
!$acc directive [clause]
     ! body
!$acc end directive
```

Figure 2.4: OpenACC directives

```
subroutine matmul(a, b, c, n)
   double precision(n,n) :: a, b, c
   integer :: i, j, k, n
   double precision :: cc
!$acc data copyin( a(1:n, 1:n), b(1:n, 1:n)) copyout( c(1:n))
!$acc kernels
!$acc loop
   do j = 1, n
!$acc loop
      do i = 1, n
         cc = 0
         do k = 1, n
            cc = cc + a(i,k) * b(k,j)
         end do
         c(i,j) = cc
      end do
   end do
!$acc end kernels
!$acc end data
end subroutine matmul
```

Figure 2.5: OpenACC matrix multiplication

Chapter 3

Evaluations of Directive-based Programming Models for Many-core Processors

3.1 Introduction and Motivation

Parallel programming with compiler directives, such as OpenMP, is widely used in scientific computing, where parallelism often appears in regular repetition constructs such as Fortran DO loops. OpenACC is a new specification for compiler directives that allow for annotation of the compute and data regions that are offloaded to accelerators such as GPUs. In contrast to current mainstream GPU programming, such as CUDA [22] and OpenCL [35], where more explicit compute and data management is necessary, porting of legacy CPU-based applications with OpenACC requires only code annotations without any significant structural changes in the original code, which allows considerable simplification and productivity improvement when hybridizing existing applications.

Programming with OpenACC directives, while greatly simplified, is not as flexible as using CUDA or OpenCL. For example, both CUDA and OpenCL provide fine-grained synchronization primitives, such as thread synchronization and atomic operations, whereas OpenACC does not. Efficient implementations of applications may depend on the availability of software-managed on-chip memory, which can be used directly in CUDA and OpenCL, but not in OpenACC. These differences may prevent full use of the available archi-

tectural resources, potentially resulting in greatly inferior performance when compared to highly tuned CUDA and OpenCL code.

To understand the performance implications of programming accelerators with OpenACC, this paper presents case studies of porting and optimization of kernel benchmarks and a real-world application code. As kernel benchmarks, we use matrix multiplications and a 3-D stencil and compare their performances on NVIDIA GPUs using both OpenACC and CUDA. As an application case study, we use a computational fluid dynamics (CFD) application, called Unified Platform for Aerospace Computational Simulation (UPACS) [37] . The code was originally written in Fortran and was parallelized with MPI, so we first identify the bottleneck loops and then rewrite part of the application for GPU execution in both CUDA and OpenACC. Also, because the performance of UPACS is usually bound by the memory throughput, we apply a series of typical memory access optimizations, including several blocking transformations and loop fusions. Throughout this optimization study, we attempt to implement each optimization in both CUDA and OpenACC and compare their applicability and effectiveness in a fair setting.

Experimental results measured using NVIDIA GPUs with multiple production OpenACC compilers show that in the microbenchmark studies, the performances of the OpenACC versions are on average approximately half of the corresponding CUDA versions when the same manual optimizations are applied, with performance reaching up to 98% depending on the compiler. A similar trend in average performance is also observed for the UPACS application, but the OpenACC performance relative to CUDA reaches only 64% at best. We also find that the highly tuned versions have a larger performance gap, as some of the optimizations, particularly those based on shared memory, cannot be applied because of the limitations of the programming interface of the current OpenACC specification. Specifically, the highly tuned CUDA versions of matrix multiplication, a 7-point stencil, and UPACS are faster than the best performing OpenACC results by factors of 2.7, 1.2, and 2.4, respectively.

3.2 Evaluation Methodology

The goal of this paper is to understand the performance of OpenACC-based hybrid codes on GPU accelerators. To that end, we develop two OpenACC-

based kernel benchmarks, a matrix multiplication and a 7-point stencil, and compare them with the CUDA-based versions. We choose these two kernels to study the performance implications of both compute and memory intensive code, respectively. We also study an application performance by extending UPACS with both OpenACC and CUDA.

For each of the benchmarks and application, starting from a CPU-based reference code, we incrementally develop multiple versions that run on the GPU with varying degrees of optimizations. The baseline version has minimum extensions to enable it to run on the GPU. The rest of this section describes each of the benchmarks and their baseline implementations in OpenACC and CUDA, followed by Section IV, which describes the optimized versions.

3.2.1 Kernel Benchmarks

Matrix Multiplication

We use a double-precision multiplication of $C = A \times B$, where each dimension is n. The OpenACC baseline implementation is shown in Figure 3.1. A `kernels` directive is used in this implementation. We assume that all arrays have already been made available on the GPU device memory by a data region directive that is not shown here. The routine is parallelized by using the two `loop` directives associated with the `i` and `j` loops.

Our CUDA implementation of the matrix multiplication kernel is very straightforward. We map multiple threads and thread blocks to the `i` and `j` loops and let each thread compute the innermost `k` loop. We selected a thread block size of 16×16.

7-point Stencil

We use a single-precision 7-point stencil kernel with the Dirichlet boundary condition. The OpenACC baseline implementation is illustrated in Figure 3.2. As we did in the matrix multiplication, we again assume that all necessary data exist on the GPU memory, having been transferred by a data region directive that is not shown here. We annotate all the three loops with the `loop` directive.

Our CUDA implementation of the stencil divides the `x` and `y` loops by all of the threads across the thread blocks, but lets each thread compute the

```
1  ! a, b, c : 2-D n*n matrices of double values
2  !$acc kernels present (a, b, c)
3  !$acc loop
4     do j = 1, n
5  !$acc loop
6        do i = 1, n
7           cc = 0
8           do k = 1, n
9              cc = cc + a(i,k) * b(k,j)
10          end do
11          c(i,j) = cc
12       end do
13    end do
14 !$acc end kernels
```

Figure 3.1: OpenACC baseline implementation of matrix multiplication

z loop sequentially. We use a thread block size of 64×4, which typically performs efficiently in stencil kernels on NVIDIA GPUs.

3.2.2 Application Case Study

We use the CFD software package, called UPACS, which has been under development at the Japan Aerospace Exploration Agency since late 1990 ' s [37]. UPACS consists of nearly one hundred thousand lines of Fortran 90 code, providing a large number of CFD solvers and their supporting components. It uses a multi-block and overset structured grid system with an underlying data structure that is a collection of loosely-connected rectangular 3-D grids. The size of each grid is typically 603, but can be chosen adaptively depending on the simulation settings and the execution environment.

In the original UPACS, a block-wise parallelization scheme is implemented with MPI, where each MPI process sequentially processes the assigned blocks with optional compiler-based automatic parallelization of the nested DO loops inside the blocks.

In this application case study, we selected a flux-based Navier-Stokes solver, which is one of the most important components in UPACS, and consists of three major computation phases: convection, viscosity, and time integration. Each phase performs a standard stencil computation for each 3-D block, as illustrated in Figure 3. The convection and viscosity phases have no loop carried dependencies, which allows all of the grid points to be computed in parallel, whereas the time integration phase has a diagonal data

```
1  ! f1, f2: 3-D nx*ny*nz arrays of float values
2  !$acc kernels present(f1 ,f2)
3  !$acc loop
4     do z = 1, nz
5  !$acc loop
6        do y = 1, ny
7  !$acc loop
8           do x = 1, nx
9              x = -1; e = 1; n = -1;
10             s = 1; b = -1; t = 1;
11             if(x == 1)   w = 0
12             if(x == nx)  e = 0
13             if(y == 1)   n = 0
14             if(y == ny) s = 0
15             if(z == 1)   b = 0
16             if(z == nz) t = 0
17             f2(x,y,z) = cc*f1(x,y,z) + cw*f1(x+w,y,z)
18                + ce*f1(x+e,y,z) + cs*f1(x,y+s,z)
19                + cn*f1(x,y+n,z) + cb*f1(x,y,z+b)
20                + ct*f1(x,y,z+t)
21          end do
22       end do
23    end do
24 !$acc end kernels
```

Figure 3.2: OpenACC baseline implementation of 7-point stencil

dependency, which requires wavefront parallelization. As shown in Table 3.1, the code size of each phase is approximately six hundred lines, consisting of up to 10 Fortran subroutines and 7 loop nests. Each phase updates 5 data objects using 20 to 33 read-only objects and 5 to 25 temporary objects. Note that each data object is a double- precision 3-D array or multiples of them aggregated to an array of structures.

The three phases are iteratively executed a given number of times, which is the most time-consuming part of the solver. For example, a sequential run with 1203 blocks using a recent Intel Xeon processor spends approximately 25% of the total time in the convection phase, 37% in the viscosity phase, and 28% in the time integration phase; as a result, 90% of the total time is consumed by just these three phases. Therefore, in this case study comparing CUDA and OpenACC, we only port the three phases to the GPU without modifying the remaining part. Details of the baseline versions are as follows.

OpenACC Version

Unlike the porting of microbenchmarks, where addition of the OpenACC directives to the CPU code is the only necessary change, we found that the original UPACS code uses Fortran features that are not currently supported by any of the OpenACC compilers used in this work. The development of the baseline version therefore consists of the following two steps.

As a first step, we replace any use of non-supported features in the original code with an equivalent representation. We found that two features that are both related to Fortran derived data types are not supported: arrays of derived data types and derived data types with variable- length arrays, both of which are used extensively in UPACS. More specifically, all data defined at cell faces, such as fluxes, were originally expressed as a single object of a derived data type. We change the variable declarations to a set of separate 3-D arrays in the OpenACC version. Also, because each block originally had a derived-type object with variable-length arrays that represent the data defined at the cell center, we copy each array out of the object into a separate array with the same length but defined independently. There are eleven such arrays in the original code, and we copy all of them to separate arrays and change their references as appropriately. In addition to these changes, we found that one subroutine did not yield correct results with any of the used compilers. It defines a stencil on each of the six boundary planes of a 3-D grid with a different stencil shape parameterized with a subroutine argument. We replace it with six unique subroutines defined for each of the six boundary planes with the non-parameterized, equivalent stencil.

Once the removal of unsupported features is complete, we offload the three phases to the GPU by defining the OpenACC compute and data regions as we did in the stencil case study. To offload computation, we add a `kernels` directive to each stencil loop nest that does not have loop carried dependencies and annotate each loop with a `loop` directive. For the time integration phase, which has a loop-carried diagonal dependency, we first change the loop nest so that the innermost loop calculates the stencil of a wavefront, and annotate the loop with a `loop` directive to parallelize it.

To save data transfers between the host and GPU memories as much as possible, we define a data region that encompasses the main loop of the three phases. We use the `copyin` option for each array variable that is used in the loop but not after the loop, and the `copy` option for variables that need the latest copy after the loop. Also, for each variable that required MPI

Table 3.1: Details of each major phase in UPACS.

Phase	LoC	Number of subroutines	Number of loop nests	Number of 3-D data		
				Read	Update	Temporary
Convection	682	10	7	29	5	15
Viscosity	599	5	5	33	5	25
Time integration	622	5	5	20	5	5

communication for boundary exchange, we transfer the whole array before and after each exchange by using the `update` directive. Although this method is in fact suboptimal, because only the thinner sub- region of the boundary and halo requires the host-GPU data transfer, we transfer the entire array for simplicity in this work. In UPACS, there are five such 3-D data: the density, the momentum for each dimension, and the energy.

CUDA Version

In the baseline CUDA version, we create a new CUDA kernel for each of the loop nests with no data dependency by following the same parallelization scheme as used in the 7-point stencil benchmark. For the loop nest with a loop carried dependency in the time integration phase, we again use the wavefront parallelization scheme as per the OpenACC version. We use the thread block size of 64×4, which is the same as that for the 7-point stencil, for all kernels in the baseline version. Also, to make the performance comparison with OpenACC consistent, we change the array of structure data to a structure of array form in the baseline CUDA version as well.

Although CUDA does not allow direct reuse of the original loop nest, our baseline kernel implementation is relatively simple because it does not have the limitations of OpenACC. Instead, the separation of the host and GPU memory spaces has a greater impact on the porting cost, because all variables used in the loop nests must be identified and must have their copies explicitly transferred to the GPU memory. It also effectively doubles the number of variable declarations, since CUDA requires a separate variable for each of the host and GPU memories, whereas in OpenACC, the same variable can be used transparently in both memories. We will compare the code sizes of the OpenACC and CUDA implementations in Section 3.4.

Note that in this case study, we develop the kernel functions and related host control programs in CUDA C and used them from the original Fortran

code. We see that the same implementation strategy can also be used with CUDA Fortran.

3.3 Optimizations

For both CUDA and OpenACC, we apply a series of manual optimizations to evaluate their effectiveness and programming costs. This section describes both the CUDA and OpenACC versions with manual optimizations, while the next section assesses their effectiveness by comparing them against the baseline versions.

3.3.1 Matrix Multiplication

Thread Mapping

Our baseline OpenACC versions do not explicitly control the mapping of the available hardware parallelism to the application-level parallelism; instead, they only have a `loop` directive on each parallel loop to instruct the compilers to parallelize the loop. Since the `loop` directive also allows explicit mapping through its options, we create a modified version with manually-tuned thread mapping in OpenACC. The tuning process involves the selection of parallelism and its degree, which is performed by using a semi-automated method that searches exhaustively for the most efficient mapping among a predefined set. We apply this optimization to both OpenACC and CUDA. Note that since CUDA always requires a mapping specification, a specification chosen on the basis of known heuristics is used in the baseline CUDA version.

3.3.2 Shared Memory Blocking

We optimize the baseline CUDA version by using the shared memory. Specifically, in the optimized version, all threads in the same thread block cooperatively load input sub matrices into the shared memory by all threads in the same thread block and use the on-chip data in the loop to compute the inner products. This scheme reduces the number of global memory accesses, if no hardware caching is performed, by a factor of T_x for the multiplicand matrix and a factor of T_y for the multiplier, where T_x and T_y are the dimensions of the 2-D thread block used in our benchmark implementation. Although our code is not fully optimized as the DGEMM routine in the CUBLAS library,

we use the performance number as a reference to evaluating the optimization effect.

The same optimization cannot be expressed directly in OpenACC since it does not have programming constructs to access the shared memory. One indirect way would be to use the `cache` directive, which can be used to request data objects to be cached on the shared memory. Among the compilers used in this work, only the latest PGI compiler (version 12.10) claims to fully support the directive. We were, however, unsuccessful in using the directive because of compilation errors, and thus our optimization evaluation for the matrix multiplication only includes the CUDA case.

3.3.3 7-point Stencil

Thread Mapping

As in the matrix multiplication, we evaluate the effectiveness of thread mapping optimization.

Branch Hoisting

As shown in Figure 3.2, the stencil loop has six branches inside the innermost loop. Since the branches using variables y and z are loop invariant, we simply move them outside of the loop body. To completely eliminate the branches from the innermost loop, we move the remaining loops by peeling the first and last iterations of the inner loop.

Register Blocking

Since the slowest changing dimension is updated by a single thread and there is a data reuse along the dimension, we can use three local variables to hold the elements of the input grid at coordinates $(x, y, z-1)$, (x, y, z), and $(x, y, z+1)$. We implement this optimization in both OpenACC and CUDA.

3.3.4 UPACS

In UPACS, we also apply the thread mapping and register blocking optimizations to both OpenACC and CUDA. We also evaluate the following code transformations that are not applicable to the microbenchmarks.

Kernel Specialization

A common pattern in the original UPACS code defines a stencil for each of the three dimensions with a similar computation pattern. The original UPACS has a common stencil subroutine for these stencils, and indirectly accesses neighbor elements by using a sub- routine parameter that specifies the stencil offsets. While this coding practice is desirable from a software engineering perspective, we found that it increased the register pressure in both the OpenACC and the CUDA code. Our kernel specialization optimizes each stencil subroutine with this pattern by creating a separate subroutine for each dimension.

Loop Fusion

Because stencils are typically memory bound, minimization of memory accesses is often a effective optimization method. As shown in Table I, there are multiple loop nests in each UPACS phase; therein, some pairs have a producer-consumer data flow with temporary 3-D array variables. To reduce the memory access pressure, we fuse these loop pairs and allocate their temporary variables on registers or shared memory, depending on the existence of an inter-thread data dependency. In CUDA, we manually apply this code transformation to six loop pairs in the convection phase. Since OpenACC does not have inter-thread data communication methods, such as the shared memory in CUDA, only two of the loops are fused in OpenACC.

Fine-Grained Parallelization in the Matrix Free Gauss-Seidel Method

The main bottleneck routine in the time integration phase uses a Matrix Free Gauss-Seidel method (MFGS), where each point uses six neighbor points as illustrated in Figure 3.3c. The MFGS method has a computation defined at each of the neighbor points, which is computed serially in the original UPACS code as well as in our baseline versions. The neighbor results are then used to update the central point. From our code inspection, we speculate that the computations for the six neighbor points are actually expensive enough to compute in parallel by allocating one thread per neighbor point. This fine-grained parallelization requires inter-thread data communications to update the central point, which is possible in CUDA by using the shared memory, but not in OpenACC.

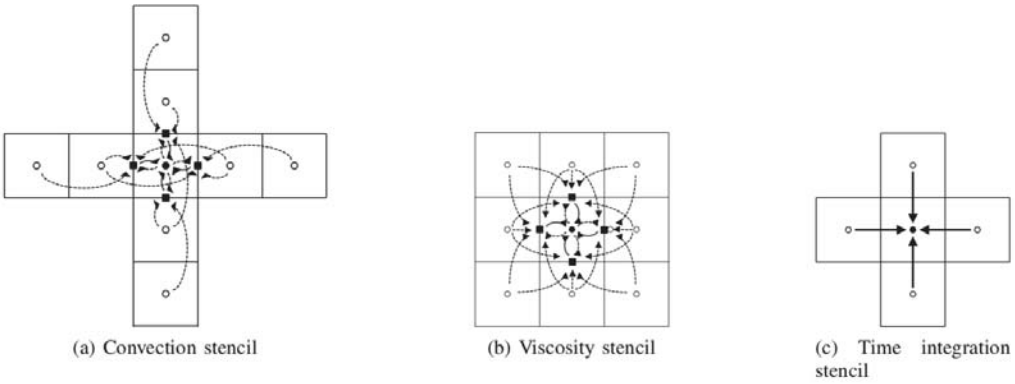

Figure 3.3: UPACS computation phases, illustrated in 2-D for simplicity. Each filled circle or rectangle represents a grid point or cell face updated by the stencil. Arrows are used to express read-write data dependencies from source to destination.

3.4 Performance Evaluation

In this section, we give performance results using a single GPU with three OpenACC production compilers, including the PGI, HMPP, and Cray compilers. Although UPACS is parallelized with MPI, and thus is able to run on multiple GPUs over distributed nodes, we use only a single GPU for comparing CUDA and OpenACC performances since inter-GPU communications should have the same effect on the overall performance of each version. We also present the number of modified or inserted lines of code in each version as a rough approximation of the programming costs.

We use a single node of the TSUBAME2.0 supercomputer, which consists of compute nodes with two Intel Xeon Westmere-EP 2.9 GHz CPUs and three NVIDIA M2050 GPUs. We use PGI CUDA Fortran 12.10 and Intel Compiler 11.1 for the CUDA and host codes. To compile the OpenACC codes, we use PGI Compiler 12.10, HMPP 3.2.4, and Cray Compiler 8.1.0.165 with CUDA 4.1. However, because the HMPP and Cray compilers were unable to compile the UPACS code, even without our OpenACC extensions, only the PGI compiler is used for the application case study. Optimization option -O3 is used with all compilers as well as -static -xP -openmp with the Intel compiler.

Table 3.2: Number of modified lines of code in matrix multiplication.

	Baseline	Thread mapping	Shared blocking
OpenACC	9	11	
CUDA	26	26	45

3.4.1 Matrix Multiplication

Figure 3.4 shows the performance of double-precision matrix multiplication of two 20482 matrices on an M2050 GPU without counting the PCI transfer overhead. We use Fortran in both OpenACC and CUDA. Table 3.2 shows the number of modified or inserted lines of code of each version to the original CPU code. From these results, we see that the baseline OpenACC versions with the three OpenACC compilers achieve approximately 50% to 85% of performance of the baseline CUDA version with less than half of code changes. The thread mapping optimization is effective for the PGI and HMPP compilers, and improves the baseline performance by 1.6 and 3.2 times, respectively. These results indicate that thread mapping should be carefully tuned when using these two compilers. The HMPP compiler in particular generates a poorly performing mapping by default: Only sixteen thread blocks of 256 threads are spawned. Because there are fourteen SMs in Tesla M2050, this configuration would cause a load imbalance.

The shared memory blocking optimization implemented in the optimized CUDA version significantly improves the performance by a factor of three. As discussed in Section 3.3.2, all three OpenACC compilers were unable to successfully compile the benchmark program with the cache directive. While the performance of the optimized CUDA version is still far behind that of the fully tuned DGEMM, these results indicate that a relatively simple optimization can still have a large performance impact and is not yet automated in the OpenACC compilers.

3.4.2 7-point Stencil

Figure 3.5 shows the performance in terms of achieved memory throughput for a single-precision 7-point stencil on a 2563 grid. The number of modified or inserted lines of code is shown in Table 3.3. We use the same GPU and compiler configuration as used in the matrix multiplication experiment. The best performance achieved is 81.79 GB/s when the CUDA version is fully

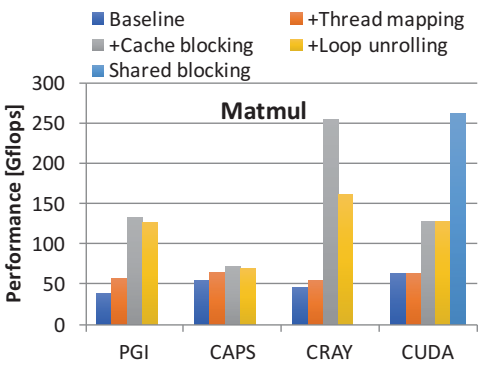

Figure 3.4: Performance of matrix multiplication of 2048^2 matrices.

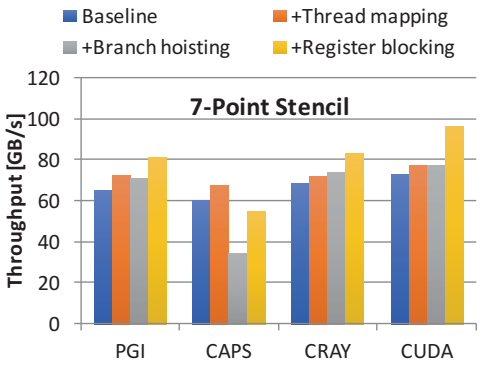

Figure 3.5: Throughput of 7-point stencil with 256^3 grids.

Table 3.3: Details of each major phase in UPACS.

	Baseline	Thread mapping	Branch hoisting	Register blocking
OpenACC	7	10	18	29
CUDA	35	35	45	56

optimized. An experiment using the bandwidthTest program included in the CUDA SDK shows that the on-memory data throughput is approximately 108 GB/s, indicating that our optimized CUDA stencil is well tuned.

Among the three OpenACC compilers, the PGI compiler performs best with a performance 19% lower than the corresponding CUDA versions. The thread mapping optimization does not yield much of a performance improvement with PGI and Cray, whereas it produces a substantial improvement with HMPP. This result is consistent with the matrix multiplication results on the thread mapping effect. Both the branch hoisting and register blocking optimizations have mixed effectiveness. This result indicates that the manual optimizations that have been known to be effective in CUDA can also be important in OpenACC, but their effectiveness depends on the actual compilers used to generate the final code. In addition, as expected, the number of modified or inserted lines of code in OpenACC is significantly smaller than that of CUDA.

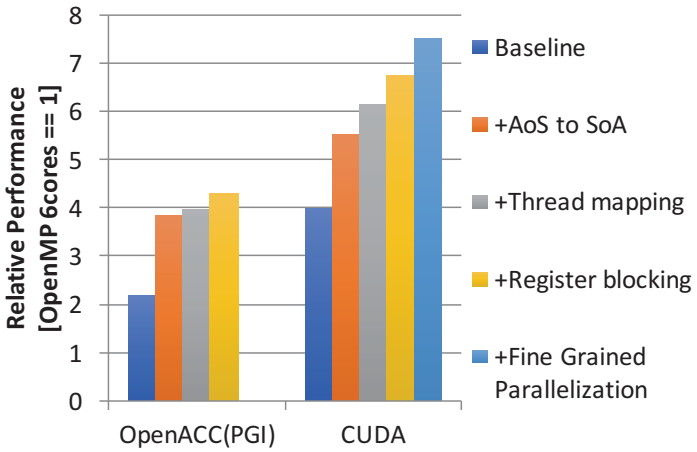

Figure 3.6: UPACS performance relative to the CPU performance. Non-applicable optimizations are left blank.

3.4.3 UPACS

Figure 3.6 shows the overall performance of the GPU UPACS versions when using the PGI OpenACC and CUDA Fortran compilers relative to OpenMP executed on a six-core Xeon CPU. The GPU performances include the overhead of host-device data transfer. The phase-wise performance is shown in Figure 3.7. Note that the versions where each optimization is not applicable are left blank in the figures. As shown in Figure 3.6, the baseline GPU performance with the PGI OpenACC compiler is 1.4 times higher than that of the CPU version. Similar to the 7-point stencil with the PGI Compiler, the thread mapping optimization slightly improves the performance, achieving 1.48 times faster performance when compared to the CPU performance. With the remaining optimizations in place, the final performance is 1.8 times faster than the CPU performance.

When compared to OpenACC, even the baseline CUDA version is faster than the fully optimized OpenACC version, reaching more than 2.1 times faster performance than that of the CPU version. Among the five optimizations, the shared-memory based MFGS optimization is particularly effective, further improving the performance by a factor of 1.5.

Table 3.4 shows the number of modified or inserted lines of code. The

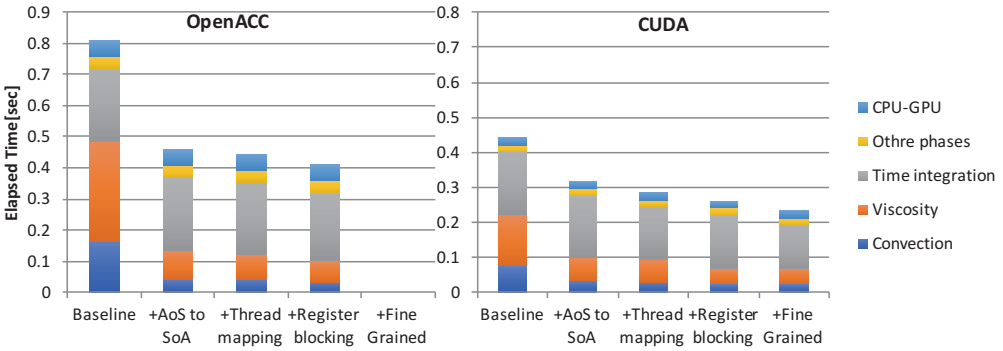

Figure 3.7: Elapsed time of each UPACS phase. Non-applicable optimizations are left blank.

Table 3.4: Details of each major phase in UPACS.

	Baseline	AoS to SoA	Thread mapping	Register blocking	Fine-Grained
OpenACC	1623	1788	1809	2940	
CUDA	5447	5607	5743	7641	7755

manual optimizations significantly increase the code changes in both OpenACC and CUDA, although the former is significantly smaller than the latter. Note that because of the minimum changes required in UPACS with the PGI OpenACC compiler, we see that even the baseline OpenACC version has almost the same degree of modifications as that of the original CPU UPACS code. Overall, both the OpenACC and CUDA versions successfully achieve faster performances when compared to the original CPU version. However, the fully tuned CUDA version is faster than the corresponding version of OpenACC by a factor of 2.7, although with a much larger degree of code changes. The results indicate that OpenACC still has a considerable room for performance improvement.

3.5 Related Work

The PGI Accelerator Model is also an extension of OpenMP for accelerator programming, and is a precursor of the current OpenACC specification [26].

It basically presents the same architecture model to the programmer, i.e., multiple levels of different parallelism with the memory space separated from the host memory. The directives provided by the PGI Accelerator model also resemble the OpenACC directives, with several minor differences. Another precursor of the OpenACC specification is the CAPS HMPP Workbench [5], which is also a directive-based accelerator programming framework.

In contrast to the high-level abstractions provided in the work discussed above, hiCUDA prioritizes achievable performance with relatively lower level abstractions [10]. It specifically targets the NVIDIA GPU architectures, and by doing so allows the programmer to directly control the data movement between the various CUDA-specific memories, including the global, constant, shared memories. As shown in our evaluation, this level of flexibility can sometimes be critical to enable optimized performance to be achieved in real-world applications, especially for experienced programmers. It would be useful if a compiler-based automated approach such as OpenACC and a lower-level explicit model could coexist within the same unified programming model.

Another extension of OpenMP for OpenACC is presented by Bayer et al. [2]. The extension is quite similar to the OpenACC specification, and a preliminary performance evaluation with matrix multiplication shows a similar performance result to that of our matrix multiplication evaluation.

Performance of an earlier version of the Cray OpenACC compiler are reported by Wienke et al. [41]. Similar to our work, their results indicate that OpenACC performance is generally similar to or lower than that of OpenCL, partly because of less efficient memory access. Our work presents more detailed performance studies with multiple levels of optimizations, and identifies that the lack of interface to ac- cess the on-chip memory can severely limit the performance when compared to hand-tuned CUDA code.

3.6 Chapter summary

This paper studied the performance implications of OpenACC for two microbenchmarks and one real-world CFD application. We first explored the baseline porting strategies of matrix multiplication, a 7-point stencil, and the UPACS application. We then studied the effectiveness of the common and application-specific optimization techniques in both OpenACC and CUDA. The evaluations indicate that the current OpenACC compilers achieve ap-

proximately 50% of performance of the CUDA versions, reaching up to 98% depending on the compiler. However, the limitation on the on-chip memory causes a significant performance gap when compared to the shared-memory optimized CUDA code. The lack of programmable control of the on-chip memory in OpenACC may be alleviated by introducing low-level abstractions, such as those explored in the hiCUDA project [10]. However, low-level interfaces are typically specific to particular architectures, and may lose program portability. The question of how performance and program portability can coexist remains an open question. One promising approach would be to improve compiler-based optimizations by auto- tuning as is partially done in [18].

Chapter 4

Extensions of OpenACC Directives for Efficient Vectorizations

4.1 Introduction

As an implicit and portable programming interface for porting legacy CPU-based applications to modern computational environment accelerated with many-core accelerators such as GPUs and Intel Xeon Phi, OpenACC [26] is receiving a lot of attention. OpenACC provides a set of OpenMP-like loop directives for the parallelization and also to manage data movement between host CPU memory and accelerator device memory. Although CUDA, which is one of the most widely used extension for GPU programming, supports only NVIDIA GPUs, OpenACC provides functional portability across different heterogeneous devices. However, the performance portability of OpenACC is known to be insufficient because different target devices often require different optimization strategies.

In our previous work [13], as a result of porting and optimizing a real-world application with OpenACC and CUDA, one of the most effective optimizations was the data layout transformation to obtain optimal performance for GPUs. However, as the appropriate data layouts for GPUs were not often suitable for multi-core CPUs, a significant manual effort was necessary for modifying the data layout, resulting in a completely different version of the code with non-localized changes. This is the performance portability

problem of OpenACC program proclaims *write once run anywhere*.

This paper presents a set of directive extensions for OpenACC that allow the programmer to declaratively adapt data layouts to target devices. While it is not completely automated as it still requires appropriate directives to be inserted into the original code, it significantly simplifies such optimization by automating most of the manual burden using the directives. We extend our prior proposal [12], where *acc transform* was introduced to transform multi-dimensional arrays, so that it can be applied to more complex program code appearing in production applications. Notably, the new proposal allows for data objects to be transformed in a more flexible way, which follows the similar concept as the data transfer directives in the original OpenACC specification. Furthermore, function calls are also supported as they are in the original OpenACC. Our prototype implementation is also extended significantly to support the new extensions and Fortran as a host language, whereas previously only the C language was supported.

To evaluate our proposal, we extend existing two real-world OpenACC applications, UPACS and CCS-QCD, with our proposed directives. Our prototype source-to-source translator for the extensions achieves 123% and 120% of the baseline performance, respectively, which are comparable to manually tuned versions.

4.2 Background

4.2.1 Data Layout and Performance Characteristics of Devices

Although the data layout has a significant influence on the performance, it can take various forms for each target application. For example, Array of Structures (AoS) data layout is often used in Computational Fluid Dynamics (CFD) applications. The space that should be calculated in CFD program is divided into discrete cells generally to deal with fluid flow computationally. These cells usually contain several physical quantities such as pressure, velocity, and so on. The set of the cells including several physical quantities is expressed intuitively as AoS by regarding the physical quantity as an element of structure, the cell as a structure, and the set of cells as an array. It is ideal that the data layout of the program is decided without realizing target devices, however, many-core accelerators are not good at AoS data layout

because of their relatively less amount of on-chip cache memories, which are relatively good at non-sequential memory access pattern, per a core. Furthermore, to get enough performance, the data layout modification may be required every time the target device is changed because the suitable data layout is different for each target device.

4.3 Related Work

Sung et al [36] propose and evaluate the effectiveness of the data layout called Array-of-Structure-of-Tiled-Array(ASTA) for GPUs and the parallel in-place transposition algorithms for ASTA. Although they focused on the data layout itself and its transformation algorithms, we focus on the performance portability of applications between different devices.

Shuai et al [4] propose Dymaxion++ that is a set of directives for data layout transformation for CUDA and OpenCL. Dymaxion++ provides three data layout transformation rule; *transpose*, *diagonal*, and *indirect*. These transformations are hidden within the PCI-e data transfer. On the other hand, in our research, we focus on high-level programming model and aim to improve the performance portability between models.

Yamada et al [43] demonstrate the effectiveness of the data layout transformation by using Xevolver framework [38]. Xevolver framework allows user-defined code transformations including derived type transformation that is not supported in our translator. Users can realize various transformations by describing transformation codes of XSLT, which is internally used for code transformation in Xevolver. Although Xevolver supports flexible transformation, it requires users to describe detailed transformation codes every data layout. Our framework is specialized for data layout transformation and has highly abstracted data layout that is suitable for auto-tuning.

4.4 Abstraction of the data layout

In this section, we briefly explain about our data layout abstraction idea. As we mentioned in the above section, OpenACC has the concept that abstracted independent memory called Host and Device. Our idea is to allow users to have different data layout on Device side from Host side and to optimize it for executed devices. The main idea of this paper is almost

same as [12]. However, our proposal was mainly data layout transformation methodology, and insufficient for porting real-world applications, for example, there was no support for function calls.

4.4.1 Proposal of the transformation directive

We propose the data layout abstraction OpenACC extension directive again to optimize applications for executed devices. Our proposal directive called *acc transform* directive is shown in Fig.4.1. We explain again the information that user should give the *transform* directive is following.

- Transformation target array name.
- The size of the array that should be transformed.
- The rule of transformation (optional).

The *transpose* shown in Fig.4.1 is one of the usages of *acc transform* directive's clause *transpose*. It indicates that the target arrays are multi-dimensional array and should be used as transposed array in the specified region. The *array_name* is the name of target multi-dimensional array. The array size can be specified with *lower* and *upper* in the same way as `acc data` directive for Fortran. The rule of transformation is unique specification in this directive. Users can explicitly chose the data layout after the transformation. It must have the number of digit same as the number of dimensions of the multi-dimensional array. The digits in the transformation rule correspond to the dimensions, which are numbered from the left starting with 1, of the multi-dimensional array. In addition, the digits show positions of dimensions after the transformation. The compiler transposes the multi-dimensional array according to the rule. For example, when array(I,J,K) specified with transformation rule (2,1,3) like fig4.1, the first dimension is changed to the second dimension, the second dimension is changed to the first dimension, and the third dimension is just as it is. So the array(I,J,K) converts array(J,I,K) in the specified region. Also, *acc transform* has another two clauses in specification. The *redim* and *expand* clauses are used to transform 1-dimensional array to multi-dimensional array and derived type array to multi-dimensional array, respectively. But both clauses are not implemented in our translator yet.

The concept of OpenACC memory model is separated `host` and `device` and users have to take responsibility of the coherency between `host` and

```
1 !$acc transform transpose(array_name(l1:u1,l2:u2,l3:u3)::(r1,r2,r3))
2    !structured block
3 !$acc end transform
```

Figure 4.1: The specification of *acc transform transpose*.

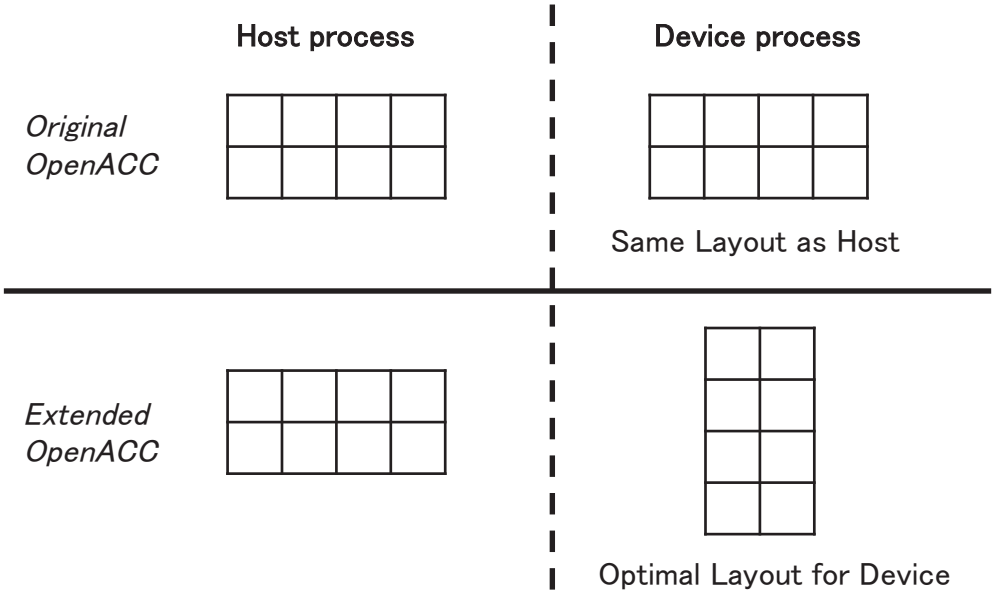

Figure 4.2: Data Layout Transformation Process

`device` by using data movement directives. Standard OpenACC allows to the `device` side having only same data layout as `host` side. Our simple idea is allowing different data layout to `device` side like fig4.2.

Although there are several minor changes such as directive name from *acc trans* to *acc transform* and difference between C and Fortran, the above explanations are almost same as our past research [12]. In this paper, to apply the directive to real world applications, we also propose following additional directives and clauses.

- !$acc *present_transform*
- !$acc transform *transpose_create*

```
1  subroutine caller
2     double precision, dimension(3,100,100) :: a
3     !$acc transform transpose(a(1:3,1:100,1:100)::(2,1,3))
4     ...
5     call callee(a)
6     ...
7     !$acc end transform
8  end subroutine caller
9
10 subroutine callee(a)
11    double precision, dimension(3,100,100) :: a
12    !$acc present_transform transpose(a(3,100,100)::(2,1,3))
13    ...
14 end subroutine callee
```

Figure 4.3: The usage of the *present_transform* directive.

- !$acc transform *transpose_in*
- !$acc transform *transpose_out*
- !$acc transform *transpose_inout*
- !$acc loop collapse(n)*::*$(r_1, r_2, ..., r_n)$

The *present_transform* directive is mainly used for procedure call like Fig.4.3. It indicates that the target arrays are already transformed and exist on device memory, that is, the transformation is unnecessary. Unlike the *transform* directive, there is no *acc end present_transform* directive. Although the target region of *acc transform* is the inside between *!$acc transform* and *!$acc end transform*, the target region of the *present_transform* directive is a given procedure like the routine directive. Thus, the subroutine `callee`'s argument `a` is also transposed in accordance with the transposition rule.

The *transpose_create*, transpose_in, transpose_out, and transpose_inout clauses indicate behavior of initialization and termination of the *transform* directive. If the *transpose_create* clause is specified, transformed array is just created on Device side without initialization and it is just deleted without copyout the result. The *transpose_in* clause indicates the transform array should be initialized based on target array but the result is not necessary. Conversely, the *transpose_out* clause indicates the initialization is not needed but the result should be copied out. Similarly, the *transpose_inout* needs both the initialization and the result. The *collapse* extension is for loop

interchange. To obtain optimal performance, optimizing memory access pattern is important as well as memory layout.

We consider that these additional directives and clauses enable users to obtain optimal performance.

4.5 Implementation

4.5.1 Implementation of the source-to-source translator

To experiment the directives mentioned above, we implement a source-to-source translator on top of the Rose Compiler Infrastructure [32]. Fig.4.4 is the overview of our translator. Rose compiler converts standard C/C++ and Fortran code into Abstract Syntax Tree(AST). Our translator is implemented as one of the optimization pass of Rose compiler by transforming the generated AST. Our translator output OpenACC code from extended OpenACC code that includes *acc transform* directives as an input like Fig.4.6. Note that the *redim* and *expand* clauses are not implemented yet. Also some specifications for C are not supported. At this time, our translator requires double space for transformed array on device memory like Fig.4.5. Although the methodology of the translator implementation is almost same as [12], we support additional directives mentioned in above section and Fortran program.

As an example of the *acc transform* directive, we show that the transformation of our translator. Fig.4.6 and Fig.4.7 are the input and output of our translator, respectively. The target arrays `A` and `B` appear in inside of the *acc transform* directive are replaced with `A_tp` and `B_tp`, respectively. `A_tp` and `B_tp` are generated transformed arrays. Transformed arrays are allocated on both side of Host and Device memory at line 3,4 and in a library called at line 17,18, respectively. As shown in Fig.4.2, although transformed arrays are not necessary for Host process, OpenACC requires managing data as pair of Host and Device so the translator allocates those on both processes. Meta information for transformation that should be given to library is secured at line 5-16 and is handed as arguments. Meta information includes the size of target array, transposition rule, and a flag to control transposition. The control flags correspond to *transpose_create/in/out/inout* clauses and appear as last arguments of library call at line 17,18 and 31,32. For example `A` is

applied *transpose_in* in Fig.4.6. That results in the control flag at line 18 is 1, that is, `A_tp` is initialized with `A`. In contrast, `B_tp` is not initialized at line 17 because `B` is specified *transpose_out*.

4.5.2 Runtime Library

We explain the implementation of the runtime library. As shown in Fig.4.2, data layout transformation executes on device process. The runtime library is implemented with OpenACC + CUDA. The transposition kernels are implemented with CUDA and the interface called from the input OpenACC program implemented with OpenACC. CUDA kernels are used for transposition kernels so this runtime library is only for NVIDIA GPUs. It is not good for OpenACC whose merit is write once run anywhere that supported devices are limited. It is future work to support other accelerator devices such as Xeon Phi.

The transposition kernels are implemented naively to support an arbitrary transposition rule for a multi-dimensional array. However, typical transposition patterns used in real-world applications are limited. For example, if the point on $X \times Y \times Z$ 3-D space has the n physical quantities, it is natural to express the space with 4-dimensional array phys(n,X,Y,Z). At this time, if the length of n is insufficient as vector parallelization length for the target accelerator device, it is also natural that the target array is transformed to phys'(X,Y,Z,n) or phys"(X,n,Y,Z). We can consider the transformation from phys(n,X,Y,Z) to phys'(X,Y,Z,n) as the 2-dimensional array transposition phys(n,X*Y*Z) to phys'(X*Y*Z,n). By regarding multi-dimensional array as 2 or 3-dimensional array, we optimize the typical transposition kernels.

4.6 Evaluations of Translator

To evaluate our translator, we use a simple kernel benchmark and two real-world applications, CCS-QCD and UPACS. The purposes of the evaluations are following,

- To evaluate the cost of the transformation of runtime library, we use a kernel benchmark.
- To evaluate the cost of the *present_transform* and *transpose_create* directives, we use a CFD application UPACS.

Table 4.1: Evaluation environment

CPU	Intel Xeon X5670 2.93 GHz (6 cores) × 2 (Hyperthreading enabled) 54 GB Memory
GPU	NVIDIA Kepler K20X 2688 CUDA cores 6GB Memory PCI-e Gen2 x16

- CCS-QCD with *transform* directives and *collapse* extension.

The evaluation environment is shown in Table 4.1.

4.6.1 Micro benchmark

To evaluate the cost of the runtime library, we use a kernel benchmark. The kernel benchmark transposes a 3 dimensional array with 6 transposition rules by using our library. The runtime library consists of two kinds of kernels; one is *transpose* that transpose the target array from the original data layout to the optimal data layout, and another is *retranspose* that is the opposite of *transpose*. Two transposition kernels that implements with CUDA are compiled with nvcc (version 7.5) compiler with -O3, -arch=sm_35 flags, and the interface of runtime library that implements with OpenACC is compiled with pgfortran (version 16.4) compiler with -acc -Mcuda -O3 -ta=tesla,pinned flags. Fig.4.8 shows the performance of the transposition kernels on K20 GPU shown in Table 4.1. The x-axis and y-axis of the graph show the transposition rules and the throughput of the kernels, respectively. The target array `org` is double precision `org`(1:10, 1:1000, 1:2000). The first dimension of `org`, which is arranged in a continuous address space on memory, is shorter than the vector length required GPUs, so it is a typical example needing transformation. The data layout of the transformed array `opt` is different every transposition rule. For example, if the transposition rule is (2,3,1), the form of `opt` is `opt`(1:1000,1:2000,1:10). The transposition rule (1,2,3) means that the forms of `opt` is same as `org`, that is, both kernels are simple memory copy. In addition, the results are the average of the 100 times evaluations of each kernel and includes host-device transfer time of Meta information such as a transposition rule or the target array size, but doesn't include the transfer time of target array. To support an arbitrary transposition rule for a

multi-dimensional array, the implementation of the transposition kernels are naive except the rule (2,1,3) and (2,3,1). The rule (2,1,3) and (2,3,1) are typical transposition patterns for GPUs so we optimize these kernels for typical target arrays whose first dimension is relatively short. Thus, these kernels achieve relatively good performance. If the target array is n-dimensional array (n > 3), we can apply the same kernel as far as the transposition pattern is same as the rules by considering plural dimensions without any order changes to be one dimension. For example, `org`(10,1000,2,4,5,50)::(2,1,3,4,5,6) can be considered as `org`(10,1000,2000)::(2,1,3) because the order of 3 to 6 dimension of `org` does not change. Actually, the performance difference between `org`(10,1000,2,4,5,50)::(2,1,3,4,5,6) and `org`(10,1000,2000)::(2,1,3) is less than 1%. The (2,3,1) *transpose* kernel achieves 75.6% of the (1,2,3) performance. In contrast, the performance of *retranspose* is only 46.3% of the (1,2,3) performance. Furthermore, the series of optimization is effective only when the first dimension size of target array is smaller than 32 (GPU warp size).

However, It can be supposed that the timing of calling of the transposition kernels in real-world applications is basically same as the Host-Device transfer, so it is important that the relative performance compared with Host-Device transfer. Fig.4.9 shows the overhead of transposition kernels. The execution times of (2,1,3) and (2,3,1) transposition kernels with Host-Device transfer are 110.9% and 112.6% of pure Host-Device transfer, respectively. The slowest is (3,1,2) pattern and its overhead achieves 36.3% of pure Host-Device transfer. Additionally, our evaluation environment uses PCI-e Gen2 generation and the throughput of Host-Device transfer is 6.4GB/s. If the PCI-e is newest one, the overhead of (3,1,2) kernels should reach to over 70%. To reduce the overhead, we consider hiding the transposition kernels with Host-Device transfer by using additional directive extension. At this time, the *acc transform* directive is completely separated from *acc data* directive and supposes that the target array is already shipped on Device side. Therefore, the *acc transform* directive doesn't know when the data transfer occurred so the directive cannot hide its cost with Host-Device transfer. As a result, we consider extending *acc data* directive like *transpose_copy(org(X,Y,Z)::(2,1,3))*. Although we anticipate further optimization and improvement of the flexibility of the description by the extension, it is future work.

4.6.2 UPACS_turbo

UPACS is the CFD software package, which has been under development at the Japan Aerospace Exploration Agency since late 1990 's [37]. It consists of nearly one hundred thousand lines of Fortran 90 code, providing a large number of CFD solvers and their supporting components. It uses a multi-block and overset structured grid system with an underlying data structure that is a collection of loosely-connected rectangular 3-D grids. In the original UPACS, a block-wise parallelization scheme is implemented with MPI; where each MPI process sequentially processes the assigned blocks with optional compiler-based automatic parallelization of the nested DO loops inside the blocks. In this evaluation, we use UPACS_turbo that is extended by IHI Corporation based on UPACS. Although UPACS_turbo includes some original solvers for their analysis, the main program structure is almost same as original UPACS. Also, we use real data offered from IHI Corporation. Experimental data is approximately 4 million points 3D grid to analyze the flow of the surrounding of single wing. The 3D gird is divided into 7 blocks to fit the wing. The biggest block size is $83 \times 96 \times 120$ and the smallest block size is $110 \times 26 \times 120$. Although this 3D grid can be divided to 7 MPI processes, we calculate with 1 MPI process in this evaluation.

Porting and optimizing of UPACS with OpenACC was done in our previous work [13]. We also port and optimize UPACS_turbo with OpenACC almost same way. We evaluated the effectiveness and the cost of data layout transformation by manual porting. As a result, we studied that the data layout transformation is very effective but the porting cost is also huge. Besides, an optimization for specific device such as GPUs can cause performance degradation for other devices such as multi-core CPUs. Thus, the data layout transformation should be automatic. In this experiment, we apply transform directives to convection and viscosity phases illustrated in Fig.4.10, which occupy 60% of total execution time of UPACS_turbo, as the first step towards the automatic data layout transformation. The convection and viscosity phases have their cell face variables defined as array of structures like Fig.4.11. At first, both phases update cell face variables, after that, update the cell center variables by using cell face variables. At the time of the cell face variables updating, their own values are not used in both phases. Thus, the target arrays of structures are used temporarily. However, our translator doesn't support *expand* clause that enables to transform derived type array yet. So we rewrite the target array of structure to simple multi-dimensional

array manually. We apply *present_transform* directives and *transpose_create* clauses to manually transformed multi-dimensional arrays, which have CPU optimal data layout because not a few number of real-world applications are optimized for multi-core CPUs.

Fig.4.12 shows the elapsed time of the one time step of convection and viscosity phases on a K20X GPU including translator's overhead. The second bar from left is the baseline of the translator evaluation. Although the translator's merits are proven by transforming from original derived type with *expand* clause, it is the future work. As a result, the translator achieves 123.5% of the performance of baseline. Furthermore, the overhead of the translator is less than 1% compared with hand written version because *transpose_create* create doesn't call transposition kernels only allocate transformed data.

4.6.3 CCS-QCD

From Fiber Miniapp Suite [6] maintained by RIKEN AICS, we use CCS-QCD as an almost real-world application. It is a benchmark program of the sparse matrix solver for lattice QCD calculation. In particular, BiCGStab solver and Clover routine are the mainly important. In CCS-QCD, the 3-D space of the lattice field is divided with MPI process, but we also use 1 MPI process in this evaluation. The target application CCS-QCD provided from Fiber Miniapp Suite has been already ported with OpenACC and optimized. The main optimizations applied to CCS-QCD are data layout transformation and loop interchange. In this evaluation, we remove both optimizations manually and use it as baseline, CPU optimal, version.

All the target arrays are shown in Table 4.2. The data type of the all target arrays are complex(kind=8). All the transposition rules shown in Table 4.2 are same as original CCS-QCD and can be considered as (2,1,3) or (2,3,1). Besides, we applied the *collapse* extension like Fig.4.13. The *collapse* extension appears at line 3. In this example, the loop order (ic, jc, ith, iz, iy, ix) is inter changed to (ith, iz, iy, ix, ic, jc) order to adapt transposed arrays.

Fig.4.14 and Fig.4.15 show the performance [GFlops] and elapsed time [sec], respectively. The problem class we use is CLASS1, which has $8 \times 8 \times 8$ 3-D space of the overall lattice field with 32 time dimension. All experiments are executed on a K20X GPU with 1 MPI process. The left most bars of both graphs show the performance of baseline version, which is removed optimizations manually. The 2nd and 3rd bars are the performance of the data layout transformed version by hand and translator, respectively. The 3rd

Table 4.2: All target data layout and transformation rule

target array	transposition rule	transpose create/in /out/inout
ue_t(1:col,1:col,0:nth,0:nz1,0:ny1,0:nx1,1:ndim)	(3,4,5,6,1,2,7)	inout
uo_t(1:col,1:col,0:nth,0:nz1,0:ny1,0:nx1,1:ndim)	(3,4,5,6,1,2,7)	inout
fclinve_t(1:clsph,0:nth,1:nz,1:ny,1:nx,1:2)	(2,3,4,5,1,6)	out
fclinvo_t(1:clsph,0:nth,1:nz,1:ny,1:nx,1:2)	(2,3,4,5,1,6)	out
ucle_t(1:col,1:col,1:nth,1:nz,1:ny,1:nx, 1:ndim*(ndim-1)/2)	(3,4,5,6,1,2,7)	createt
ucle_o(1:col,1:col,1:nth,1:nz,1:ny,1:nx, 1:ndim*(ndim-1)/2)	(3,4,5,6,1,2,7)	createt
be_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	in
xe_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	in
rte_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
pe_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
te_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
qe_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
re_t (1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
myo_t(1:col,1:spin,0:nth,0:nz1,0:ny1,0:nx1)	(3,4,5,6,1,2)	create
wce_t(1:col,1:col,0:nth,0:nz1,0:ny1,0:nx1,1:ndim)	(3,4,5,6,1,2,7)	create
wco_t(1:col,1:col,0:nth,0:nz1,0:ny1,0:nx1,1:ndim)	(3,4,5,6,1,2,7)	create
ve_t(1:col,1:col,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
vo_t(1:col,1:col,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
f1cl_t(1:clsp/2,1:clsp/2,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
f2cl_t(1:clsp/2,1:clsp/2,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
f1clinv_t(1:clsp/2,1:clsp/2,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
f2clinv_t(1:clsp/2,1:clsp/2,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create
f1cle_t(1:clsp/2,1:clsp/2,1:nth,1:nz,1:ny,1:nx)	(3,4,5,6,1,2)	create

bar's performance achieves 104.6% of the baseline version and 90.2% of the 2nd bar's. Also, the 4th and 5th bars are the performance of the data layout transformation + loop interchanged version by hand, which is original version, and translator, respectively. The 4th bar's performance is 120.7% and 92.3% compared with baseline version and hand written version. According to the experimental results, we confirm that it is insufficient to apply only data layout transformation, and the effectiveness of the loop interchanging.

4.7 Chapter summary

To improve the performance portability of OpenACC, we particularly focused on data layout and we proposed a directive extension to OpenACC that allows the users to flexibility specify optimal layouts. In this paper, we propose additional directive extensions to apply the set of directives to real world applications. We implement a translator for the set of directives and

evaluate it with real-world applications UPACS and CCS-QCD. The experimental results show that the translator overhead is less than 1% in the case of UPACS, whom target arrays don't need to keep coherency between original data layout and transformed data layout, compared with the manually optimized version. In the CCS-QCD case, our translator achieves 92.3% performance of the hand written version and 120.7% of the unoptimized version by adapting data layout transformation and the loop interchanging. These results show the effectiveness of the additional directive extensions such as *transpose_create/in/out/inout* and *loop collapse* extension.

However, our translator still needs more improvement. As we mentioned above, our translator doesn't support *expand* clause so we couldn't transform derived type target arrays of UPACS_turbo directly. In addition, our translator cannot hide the transformation kernel cost with Host-Device transfer. In order to achieve it, we plan to extend *acc data* directive but it is a future work. Furthermore, our translator doesn't support auto-tuning of data layout transformation. In this paper, we already know the optimal data layout for the target applications and we select it manually. It is the most difficult for OpenACC users to select optimal data layout for target device. To support auto-tuning is also significant important future work.

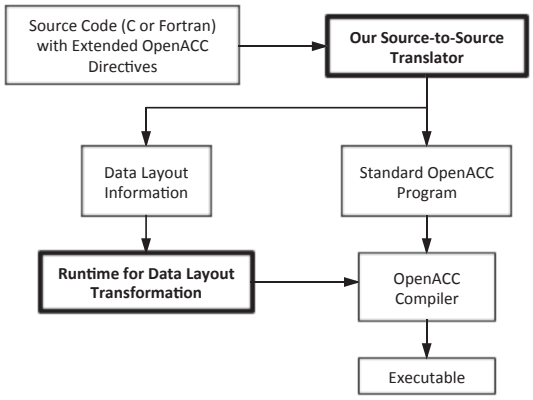

Figure 4.4: The overview of our translator. Our translator consists of source-to-source translator and runtime libraries.

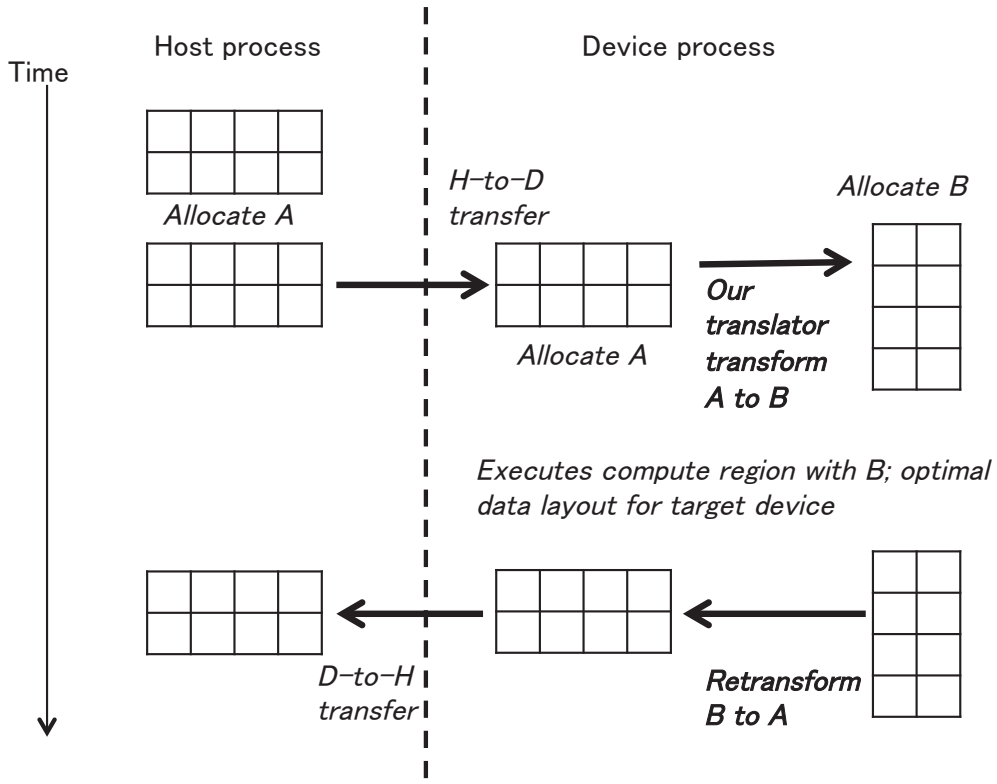

Figure 4.5: Data layout transformation process. This example transforms the target array A to B that has optimal data layout for device process.

```
!$acc data copyin (A) copyout (B)
!$acc transform transpose_in(A(l1:u1,l2:u2,l3:u3)::(2,1,3))
!$acc& transpose_out(B(l1:u1,l2:u2,l3:u3)::(2,1,3))
!$acc kernels present (A, B)
!$acc loop gang
DO k = l3, u3
   !$acc loop vector
   DO j = l2, u2
      !$acc loop seq
      DO i = l1, u1
         B(i,j,k) = A(i,j,k)
      END DO
   END DO
END DO
!$acc end kernels
!$acc end transform
!$acc end data
```

Figure 4.6: An example of acc transform directive.

```
!$acc data copyin (A), &
!$acc& copyout (B)
allocate( B_tp(l2:u2,l1:u1,l3:u3) )
allocate( A_tp(l2:u2,l1:u1,l3:u3) )
RUT_B_tp(3) = 3
UBT_B_tp(3) = u3
LBT_B_tp(3) = l3
RUT_B_tp(2) = 1
UBT_B_tp(2) = u2
LBT_B_tp(2) = l2
RUT_B_tp(1) = 2
UBT_B_tp(1) = u1
LBT_B_tp(1) = l1
RUT_A_tp(3) = 3
    ...
LBT_A_tp(1) = l1
CALL transpose_double_3(B_tp,B,LBT_B_tp,UBT_B_tp,RUT_B_tp,0)
CALL transpose_double_3(A_tp,A,LBT_A_tp,UBT_A_tp,RUT_A_tp,1)
!$acc kernels present (A_tp, B_tp)
!$acc loop gang
DO k = l3, u3
   !$acc loop vector
   DO j = l2, u2
      !$acc loop seq
      DO i = l1, u1
         B_tp(j,i,k) = A_tp(j,i,k)
      END DO
   END DO
END DO
!$acc end kernels
CALL retranspose_double_3(A,A_tp,LBT_A_tp,UBT_A_tp,RUT_A_tp,0)
CALL retranspose_double_3(B,B_tp,LBT_B_tp,UBT_B_tp,RUT_B_tp,1)
deallocate( A_tp )
deallocate( B_tp )
!$acc end data
```


Figure 4.7: The output of our translator using 4.6 as an input.

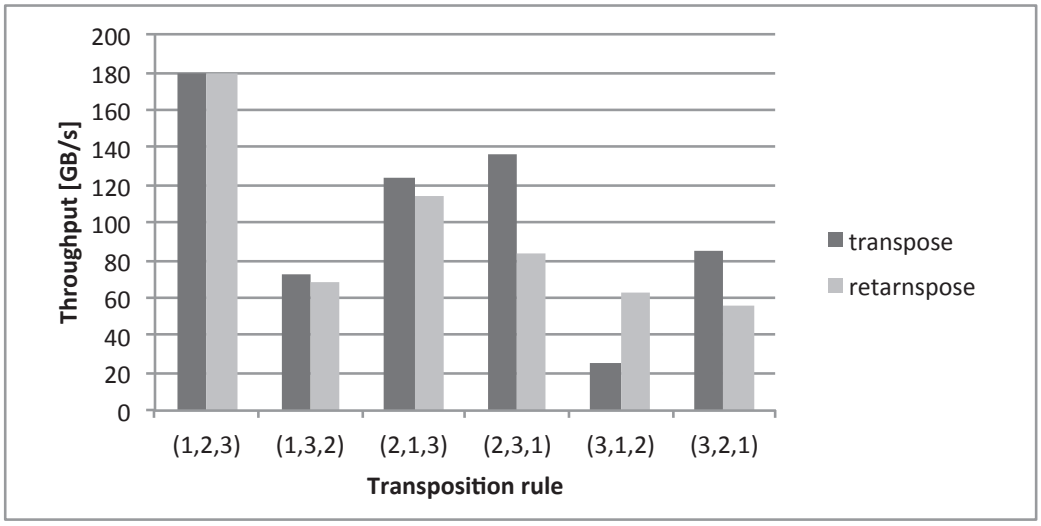

Figure 4.8: Runtime transposition kernel bandwidth (without PCIe transfer).

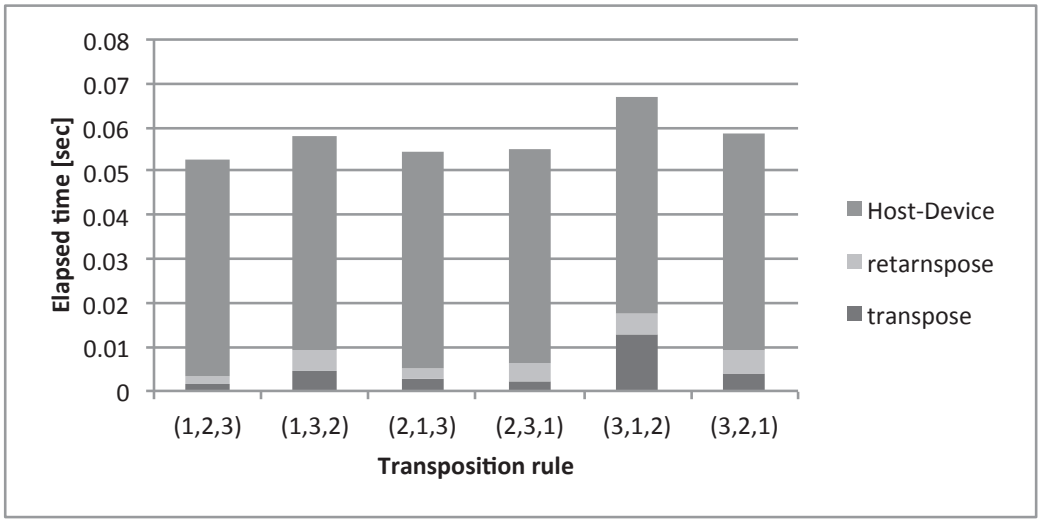

Figure 4.9: The elapsed time of the PCIe transfer time + transposition kernel time.

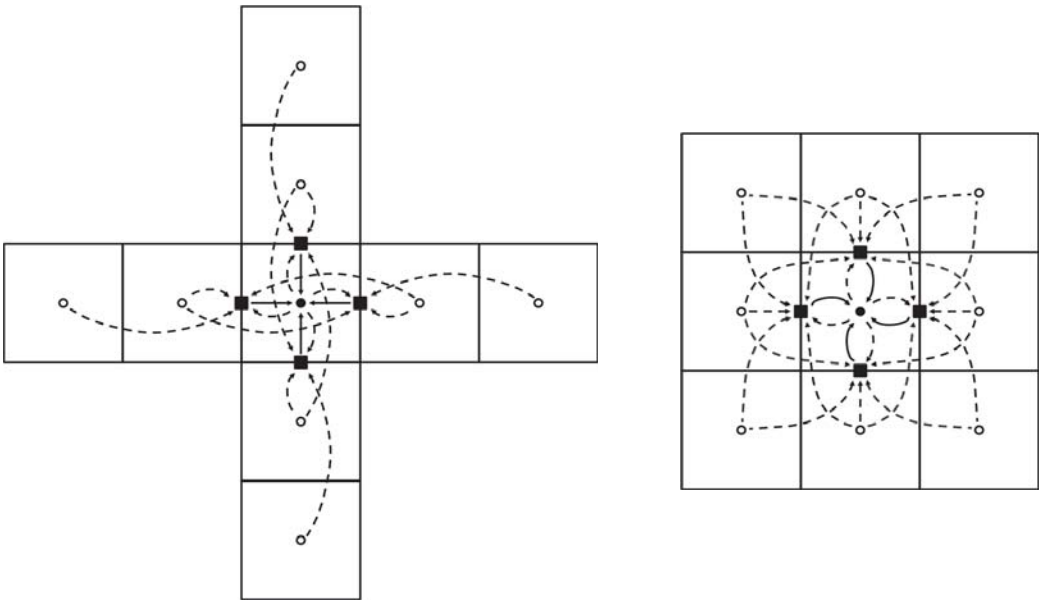

Figure 4.10: Convection (left) and viscosity (right) computation phases, illustrated in 2-D for simplicity. Each filled circle or rectangle represents a grid point or cell face updated by the stencil. Arrows are used to express read-write data dependencies from source to destination.

```
1   type cellFaceType
2      real(4)                :: area, nt
3      real(4), dimension(3) :: nv
4      real(4), dimension(5) :: flux
5      real(4), dimension(5) :: q_l,q_r
6      real(4)                :: shockFix
7   end type cellFaceType
8    ...
9   subroutine rhs_convect(blk)
10     type(blockDataType), intent(inout) :: blk
11     type(cellFaceType),dimension(:,:,:),pointer:: cface
12     allocate(cface(-1:blk%in+1, -1:blk%jn+1, -1:blk%kn+1))
13     ...
```

Figure 4.11: The target array of structures used in UPACS_turbo convection phase. Each structure defined at cell face illustrated as a filled circle in Fig.4.10.

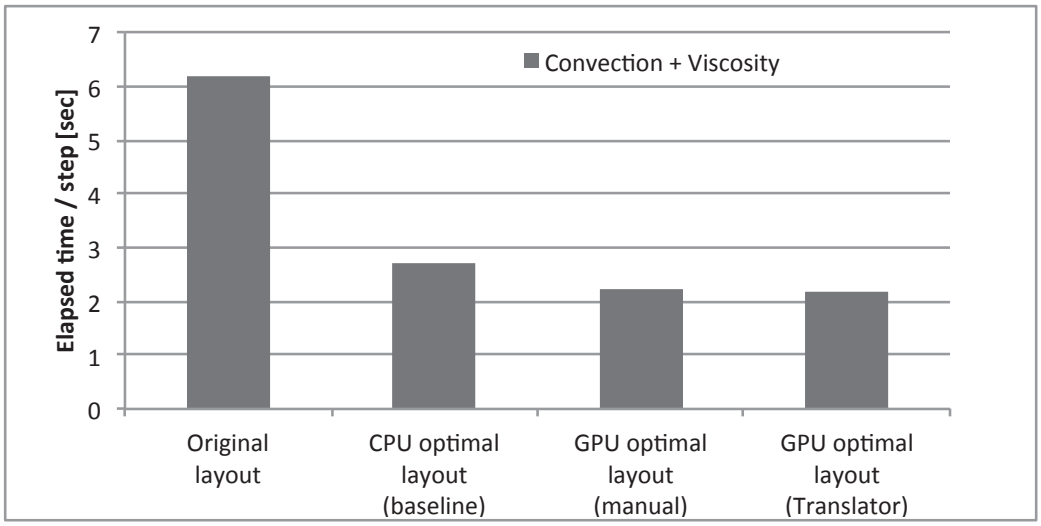

Figure 4.12: Elapsed time of the convection and viscosity phases of UPACS

```
1  !$acc kernels pcopy (wce_t, wco_t), &
2  !$acc & pcopyin (ue_t, uo_t) async (0)
3  !$acc loop collapse(6)::(3,4,5,6,1,2)
4  !$acc & independent &
5  !$acc & gang vector (128)
6     do ix=ixlow,NX
7     do iy=iylow,NY
8     do iz=izlow,NZ
9     do ith=0,NTH
10    do jc = 1,COL
11    do ic = 1,COL
12          ...
```

Figure 4.13: Applying *loop collapse* extension to a loop nest of CCS-QCD.

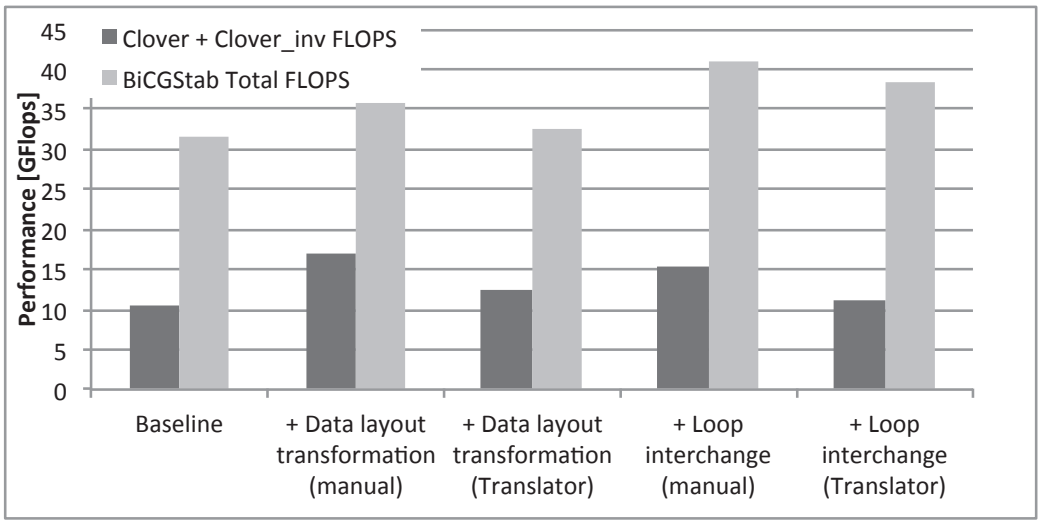

Figure 4.14: The performance of CCS-QCD

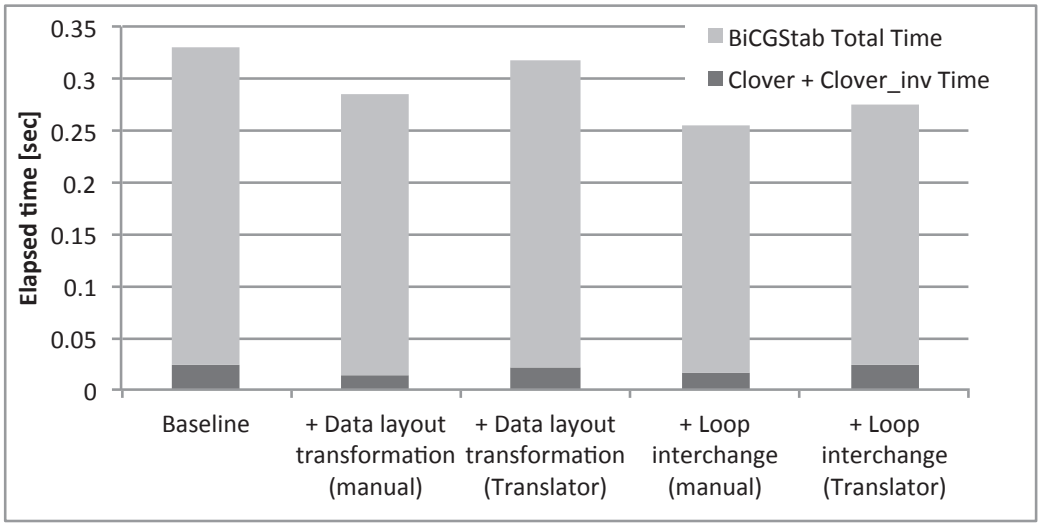

Figure 4.15: Elapsed time of CCS-QCD

Chapter 5

Framework Design for Many-core Processors

5.1 Introduction

The boundary element method (BEM) has several scientific applications. This method requires fewer unknowns and has a lower meshing cost compared to other volume discretization methods because it requires only the surface of the target objects for analysis. However, the computational cost and memory footprint of BEM analysis are significantly high because a dense coefficient matrix is generated during the analysis. To overcome these problems, parallel computing and approximation techniques, such as hierarchical matrices ($\mathcal{H}$-matrices) [8, 9, 7], $\mathcal{H}^2$-matrices [3], and the fast multipole method (FMM) [45] are often used for BEM analysis. Although these techniques have huge programming costs, BEM-BB [29], an open-source software framework for parallel BEM analysis, is useful to for reducing these costs. The framework employs $\mathcal{H}$-matrices to approximate the dense coefficient matrix, and it is parallelized using the MPI and OpenMP models. The BEM-BB framework allows for faster BEM analysis on parallel computers by simply preparing programs to calculate the integrals of boundary elements, settings of boundary conditions, and analysis output. In addition, the parallelization and the approximation programs are encapsulated in the framework. Thus, users can concentrate on developing the most important aspects of BEM analysis, namely, a user-defined function for calculating the i-th row and the j-th column of the coefficient matrix. Furthermore, the user-defined function

may vary depending on the targeted physical phenomena.

However, this framework does not consider single instruction multiple data (SIMD) vectorization, which is important for achieving high-performance computing on existing processors. For example, the most recent Intel processors, such as Skylake EP/EX and Xeon Phi Knights Landing (KNL), support AVX-512, that is, a 512-bit SIMD instruction set. SIMD vectorization cannot be separated from user-defined functions, unlike in MPI and OpenMP parallelization, because SIMD vectorization is instruction-level parallelization and because user-defined functions can vary. However, SIMD vectorization is difficult for application programmers because it requires knowledge of the compiler and the target processor architecture.

In this paper, we present a framework design based on BEM-BB for SIMD vectorization. A design to encapsulate SIMD-related aspects is proposed. In addition, we evaluate the performance of the proposed framework by solving two problems, namely, static electric field analysis with a perfect conductor and static electric field analysis with a dielectric, which contain different user-defined functions, on Intel Broadwell processor (BDW) and Intel Xeon Phi Knights Landing (KNL). We compare the performance of the proposed framework with the original framework and that of hand-tuned user functions. The results show that the proposed framework offers performance improvements of 2.22x and 4.34x compared to the original framework for the BDW processor and the KNL processor, respectively. Furthermore, the experimental results demonstrate that the performance of the framework is comparable to that achieved using the hand-tuned programs.

The remainder of this paper is organized as follows. In Section 5.2, we provide an overview of the BEM-BB framework. The proposed framework is described in Section 5.3. Numerical experiments involving electric field analysis are described in Section 5.4, and a few conclusions and suggestions for future work are presented in Section 5.6.

5.2 BEM-BB framework

In this section, the BEM-BB framework, which is the baseline implementation in this study, is introduced. The BEM-BB software framework is used for parallel BEM analysis. It is implemented in the Fortran90 programming environment and parallelized using the OpenMP + MPI hybrid programming model. To reduce the computational cost of parallel programming, the

framework supports model data input, assembly of the coefficient matrix, and solution of linear systems, steps that are generally required in BEM analysis. When employing this framework, users are required to generate user-defined functions that calculate each element of the coefficient matrix. In other words, users are required to implement a program to calculate the integrals of boundary elements, which depend on the governing target of BEM analysis. The target integral equation of the BEM-BB framework is described as follows. For $f \in H'$, $u \in H$ and a kernel function of a convolution operator $g : \mathbb{R}^d \times \Omega \to \mathbb{R}$,

$$\int_\Omega g(x,y)u(y)\mathrm{d}y = f \tag{5.1}$$

where $\Omega \subset \mathbb{R}^d$ denotes a $(d-1)$-dimensional domain, H the Hilbert space of functions on a Ω, and H' dual space of H. To numerically calculate Eq.(5.1), we divide the domain, Ω, into the elements $\Omega_h = \{\omega_j : j \in J\}$, where J is an index set. In weighted residual methods, such as the Ritz-Galerkin method and the collocation method, the function u is approximated from a n-dimensional subspace $H^h \subset H$. Given a basis $(\varphi_i)_{i\in\supset\!\!\!\!-}$ of H^h for an index set $\supset\!\!\!\!- := \{1,\ldots,N\}$, the approximant $u^h \in H^h$-u can be expressed using a coefficient vector $\phi = (\phi_i)_{i\in\supset\!\!\!\!-}$ that satisfies $u^h = \sum_{i\in\supset\!\!\!\!-} \phi_i\varphi_i$. Note that the supports of the basis $\Omega^h_{\varphi_i} := \mathrm{supp}\ \varphi$ are assembled from the sets ω_j. Equation (5.1) is then reduced to the following system of linear equations.

$$A\phi = b \tag{5.2}$$

$$A_{ij} = \int_\Omega \varphi_i(x) \int_\Omega g(x,y)\varphi(y)\mathrm{d}y\mathrm{d}x \tag{5.3}$$

$$b_i = \int_\Omega \varphi_i(x) f \mathrm{d}x \tag{5.4}$$

Here, $i, j \in \supset\!\!\!\!-$. The user-defined function required to calculate the elements of the i-th row and the j-th column of the coefficient matrix is expressed as Eq.(5.3).

There are two versions of the implementation: one based on dense matrix computations and the other based on $\mathcal{H}$-matrix computations. Although the $\mathcal{H}$-matrix version depends on the distributed parallel $\mathcal{H}$-matrix library $\mathcal{H}$ACApK [14], the problems of vectorization are similar. As shown in Fig. 5.1, the proposed framework consists of three components: model data

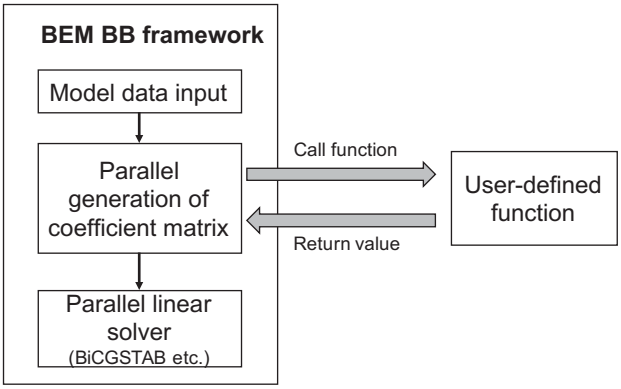

Figure 5.1: The design of BEM-BB framework

input, coefficient matrix generation, and linear solver. In this study, the objective is to interface coefficient matrix generation with user-defined function. Therefore, we focus on the coefficient matrix generation component.

Fig. 5.2 shows the coefficient matrix generation part. The target coefficient matrix is distributed to multiple thread and each thread sequentially calculates the i-th row and the j-th column element by using user-defined function. The coefficient matrices generated using the dense matrix version and the $\mathcal{H}$-matrix version are a dense matrix and an $\mathcal{H}$-matrix, respectively. A $\mathcal{H}$-matrix is also called a hierarchical matrix. $\mathcal{H}$-matrices are among the techniques used to approximate dense matrices. An $\mathcal{H}$-matrix is a set of low-rank approximated sub-matrices and small dense sub-matrices as shown in Fig. 5.2. $\mathcal{H}$ACApK generates the coefficient $\mathcal{H}$-matrix by exploiting the user-defined function according to the Adaptive Cross Approximation (ACA) algorithm [16]. The ACA algorithm is an approximation technique used to generate a low-rank approximated matrix of a dense matrix without generating the target dense matrix.

The interface of the user-defined function is shown in Fig. 5.3. In both versions, the function is called from each thread concurrently. To vectorize the user-defined function, the caller of the function, too, is important. Figures 5.4 and 5.5 show the callers of the user-defined functions of the dense matrix version and the $\mathcal{H}$-matrix version, respectively. Both programs call the user-defined function in loop structures. These loops are the target of SIMD vectorization. In the following sections, we treat the implementation

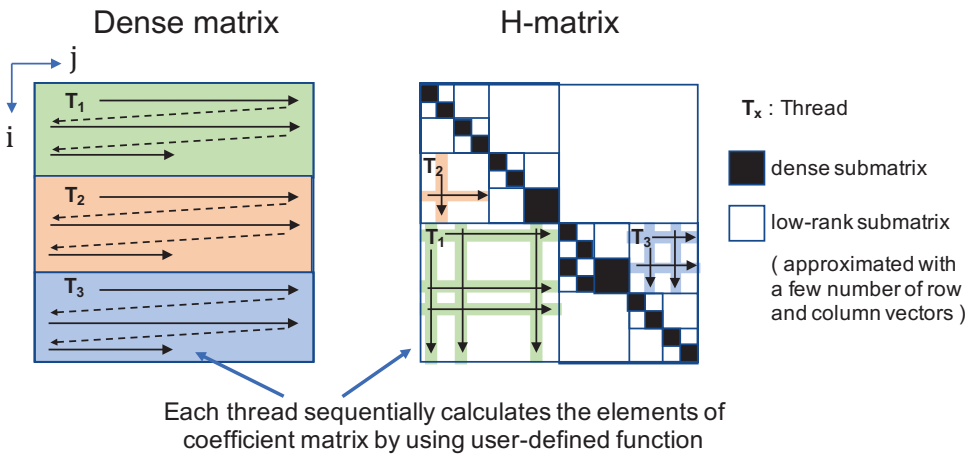

Figure 5.2: Parallel generation of coefficient dense matrix and $\mathcal{H}$-matrix.

shown in Fig. 5.4 as the baseline.

5.3 Framework Design for SIMD Vectorization with OpenMP SIMD Directives

In general, three methods are used to perform SIMD vectorization: (1) relying on compiler auto-vectorization, (2) using compiler directives, and (3) using intrinsic functions. However, vectorization using intrinsic functions is cumbersome job, and the required intrinsic functions depend completely on the user-defined function. In this study, we employ compiler auto-vectorization and the directive method. To use SIMD instructions efficiently, there are two constraints on the SIMD target vectors.

- There should be no data dependency among the elements of the target vector.
- Vector elements should be stored contiguously.

In addition, to generate efficient code by using compiler vectorizations, the code should be obviously vectorizable from the compiler's view point. Any new framework design should consider the above points. Furthermore, the

```
1  real(8) function ppohBEM_matrix_element_ij(i,j,nond,nofc,nond_on_fc,np,
       intpara_fc,nint_para_fc,dble_para_fc,ndble_para_fc,face2node)
2     !$omp declare simd
3     type :: coordinate
4        real(8) :: x,y,z
5     end type coordinate
6     integer ,intent(in) :: i,j,nond,nofc,nond_on_fc,nint_para_fc,
          ndble_para_fc
7     type(coordinate),intent(in) :: np(*)
8     integer, intent(in) :: face2node(3,*),int_para_fc(nint_para_fc,*)
9     real(8), intent(in) :: dble_para_fc(ndble_para_fc,*)
10
11    ! User defined calculations for the i-th row and the j-th column
          element
12
13 end function ppohBEM_matrix_element_ij
```

Figure 5.3: An interface of a user-defined function to calculate the i-th row and the j-th column element of the coefficient matrix. The function arguments after i and j are used as input variable of the calculation.

design should be user-friendly. Efficiently vectorized SIMD code should be generated if users are unaware of compiler requirements.

5.3.1 New interface definition for compiler vectorization

According to the two compiler requirements, the main problem associated with vectorization pertains to data access. Even though the computations associated with a user-defined function can be executed independently, if a compiler detects possibilities of data dependency, it conservatively generates instructions that are not fully vectorized. Therefore, we propose to handle data access and computation separately in the proposed framework design. We introduce two new interfaces `set_args` (Fig. 5.6) and `vectorize_func` (Fig. 5.7) for data access and computation, respectively. Figure 5.8 shows the function caller based on Fig. 5.4. The variables `SIMDLENGTH`, which appear in Figs. 5.7 and 5.8 and are defined by users, represent the SIMD length of the target processor. For example, the recommended `SIMDLENGTH` for KNL, which has a 512-bit (= sizeof(double) $\times 8$) wide SIMD unit, is 8. From the compiler's viewpoint, the `!$omp simd` loop (Fig. 5.8 line 14) has no data dependency because the arguments and the return values of `vector_func`

```
1 do i=lhp, ltp
2   !$omp simd
3   do j=j_st, j_en
4     a(j,i) = ppohBEM_matrix_element_ij( i, j, nond, nofc, &
5                                         nond_on_fc, np, intpara_fc, &
6                                         nint_para_fc, dble_para_fc, &
7                                         ndble_para_fc, face2node )
8   enddo
9 enddo
```

Figure 5.4: User-defined function caller for dense matrix. Here, a(j,i) is a coefficient dense matrix. The ranges of i and j are assigned to each thread adequately.

have no alias and are accessed independently for each iteration of the loop. In addition, the arguments and return values are stored contiguously. At this point, if the SIMD interface of the `vectorize_func` corresponds to the SIMD length, the loop (Fig. 5.8 lines 13-17) is vectorized similarly to a vector function.

To safely vectorize `vectorize_func`, we constrain the function such that it cannot contain globally accessible variables, allocatable arrays, or save variables. In addition, the SIMD interfaces of all functions or subroutines called from `vectorize_func` should correspond to the SIMD length. This parallelization method is similar to the Single Program Multiple Data (SPMD) programming model because each SIMD element executes a single program simultaneously.

To reduce the data access cost, we introduce a pair of interfaces `set_args_i` and `set_args_j`. In BEM analysis, the required data such as coordinate of the i-th element and the j-th element usually depends only on the variables `i` and `j`, respectively. Therefore, the subroutines `set_args_i` and `set_args_j` are used to set arguments depending only on `i` and `j`, respectively. The pair of interfaces work effectively in the $\mathcal{H}$-matrix version. As shown in Fig. 5.5, `i` and `j` are constants in the lines 4-9 loop and lines 13-18 loop, respectively.

5.3.2 Using the framework

The new interfaces are easy to vectorize for compilers, but they are not user-friendly. Specifically, the numbers of arguments of the `set_args` subroutine and the `vectorize_func` function depend on the target application, which

```
if( column vector calculation )
  i = ip + nstrtl-1
  !$omp simd private(j)
  do ii=1,s_m
    if(colmsk(ii)==0) then
      j = ii + nstrtt-1
      colvec(ii)=HACApK_entry_ij(i,j,st_bemv)
    endif
  enddo
else if( row vector calculation )
  j = ip + nstrtt-1
  !$omp simd private(i)
  do ii=1,t_m
    if(rowmsk(ii)==0) then
      i = ii + nstrtl-1
      rowvec(ii)=HACApK_entry_ij(i,j,st_bemv)
    endif
  enddo
endif
```

Figure 5.5: User-defined function caller for sub-matrix of $\mathcal{H}$-matrix. Here, HACApK_entry_ij is a wrapper function of ppohBEM_matrix_element_ij. The structure st_bemv contains the variables required as arguments of the user-defined function.

means users are required to modify the framework program in order to add variable declarations and correspond to the interface. In addition, users must vectorize the user-defined functions by using `!$omp declare simd` pragma. Furthermore, if users insert a wrong directive, the compiler generates a correct but unvectorized slow executable, which is often more cumbersome compared to a bug.

To minimize these difficulties, we require users to prepare the followings.

- Implement include files.
- Implement the `set_args, set_args_i, set_args_j` and the `vectorize _func` without the SIMD directives in the file "user_func.f90".
- Correctly implement the dummy function `ppohBEM_matrix_element_ij _dummy` (Fig. 5.9) without modifying the dummy function itself.
- Provide `SIMDLENGTH` of the target processor by using the -D compiler flag.

```
1  subroutine set_args(i,j,nond,nofc,nond_on_fc,np,intpara_fc,nint_para_fc
       ,dble_para_fc,ndble_para_fc,face2node,darg1,darg2,...,dargN,iarg1,
       iarg2,...,iargM)
2      real(8), intent(out) :: darg1,darg2,...,dargN
3      integer, intent(out) :: iarg1,iarg2,...,iargM
4
5      !  User defined data access for calculating an element of the i-th
           row and the j-th column from arrays to scalar args
6
7  end subroutine set_args
```

Figure 5.6: New interface for data access. The former arguments are the same as ppohBEM_matrix_element_ij. The latter arguments are the scalar variables used in vectorize_func. The number of arguments depends on the target application.

```
1  real(8) function vectorize_func(darg1,darg2,...,dargN,iarg1,iarg2,...,
       iargM)
2      !$omp declare simd simdlen(SIMDLENGTH)
3      real(8), intent(in) :: darg1,darg2,...,dargN
4      integer, intent(in) :: iarg1,iarg2,...,iargM
5
6      ! User defined calculations for an element of the i-th row and j-th
           column
7
8  end function vectorize_func
```

Figure 5.7: New calculation interface. This function should be called after the set_args subroutine and vectorized. All arguments of this function should have intent(in) attribute.

The include files that appear in the dummy function are used in the subroutine call interface. First, users of the framework must implement the include files as a fill-in-the-blank puzzle to correct the dummy function. In other words, the return value of the dummy function should be equal to `ppohBEM_matrix_element_ij`. At this point, users need not consider SIMD vectorization. Notably, users cannot modify the dummy function itself. If users do not need the `set_args` function, they must create an empty "call_set_args.inc" file. Second, the users must implement the user-defined functions in "user_func.f90." Notably, users need not consider SIMD vectorization as well. Finally, users must define the variable `SIMDLENGTH` by using a compiler option. During

```
real(8),dimension(SIMDLENGTH) :: ans
real(8),dimension(SIMDLENGTH) :: darg1,darg2,...,dargN
integer,dimension(SIMDLENGTH) :: iarg1,iarg2,...,iargM
   ...
do i=lhp, ltp
   do jj=j_st, j_en, SIMDLENGTH
      ii = 1
      do j=jj,min(jj+SIMDLENGTH-1,j_en)
         call set_args(i,j,...,darg1(ii),darg2(ii),...,dargN(ii) &
                       ,iarg1(ii),iarg2(ii),...,iargM(ii))
         ii = ii+1
      end do
      !$omp simd
      do ii = 1, SIMDLENGTH
         ans(ii) = vectorize_func(darg1(ii),darg2(ii),...,dargN(ii) &
                       ,iarg1(ii),iarg2(ii),...,iargM(ii))
      end do
      ii = 1
      do j=jj,min(jj+SIMDLENGTH-1,j_en)
         a(j,i) = ans(ii)
         ii = ii+1
      end do
   enddo
enddo
```

Figure 5.8: User-defined function using new interface caller for dense matrix.

compiling, the compile script automatically inserts SIMD directives into the user-defined functions implemented in user_func.f90 and automatically transforms the include files to adjust the framework, as shown in Fig. 5.10. Based on the results of the auto-transformation, we succeeded in separating almost all aspects related to SIMD vectorization from the user-defined function. Therefore, users are required to set only the `SIMDLENGTH` of the target processor.

5.4 Numerical Evaluations

5.4.1 Test Model and Processors

In this section, we evaluated the proposed framework by performing BEM analysis of two electrostatic field problems. We assumed a perfectly conductive sphere and a dielectric sphere. The electric potentials of the perfect conductor and the dielectric are given by the following functionals $\mathcal{P}$ and $\mathcal{D}$, respectively:

```
real(8) function ppohBEM_matrix_element_ij_dummy(i,j,nond,nofc,
    nond_on_fc,np,intpara_fc,nint_para_fc,dble_para_fc,ndble_para_fc,
    face2node)
   implicit none
   type :: coordinate
      real(8) :: x,y,z
   end type coordinate
   integer ,intent(in) :: i,j,nond,nofc,nond_on_fc,nint_para_fc,
       ndble_para_fc
   type(coordinate),intent(in) :: np(*)
   integer, intent(in) :: face2node(3,*),int_para_fc(nint_para_fc,*)
   real(8), intent(in) :: dble_para_fc(ndble_para_fc,*)
   integer :: ii,jj,j_st,j_en,lhp,ltp
   real(8) :: ans
#include "declaration.inc"
#include "call_set_args_i.inc"
#include "call_set_args_j.inc"
#include "call_set_args.inc"
#include "vectorize_func.inc"
   ppohBEM_matrix_element_ij_dummy = ans

end function ppohBEM_matrix_element_ij_dummy
```

Figure 5.9: Dummy function of user-defined function. Although the function is not used in the framework, users are required to implement this function correctly.

$$\mathcal{P}[u](x) := \int_\Omega \frac{1}{4\pi||x-y||} u(y)\mathrm{d}y, x \in \Omega \tag{5.5}$$

$$\mathcal{D}[u](x) := \int_\Omega \frac{\langle x-y, n(y)\rangle}{4\pi||x-y||^3} u(y)\mathrm{d}y, x \in \Omega \tag{5.6}$$

where Ω is the domain surface. Equation(5.5) and (5.6) correspond to Eq.(5.1) and the details of them are described in [7]. The spheres were set at a distance of 0.25 m from the ground with zero electric potential. The radius of the spheres was 0.25 m, and the electric potential of the spheres was 1 V.

For the numerical evaluations, we used the BDW and the KNL processors, which have a 256-bit SIMD unit and a 512-bit SIMD unit, respectively. The processor specifications are summarized in Table 5.1. For both processors, Intel Fortran compiler ver. 18.0.1 was used. The compiler options for BDW were `-align array64byte -xAVX2 -qopenmp -O3 -fpp -ipo -lm -qopt-report=5`

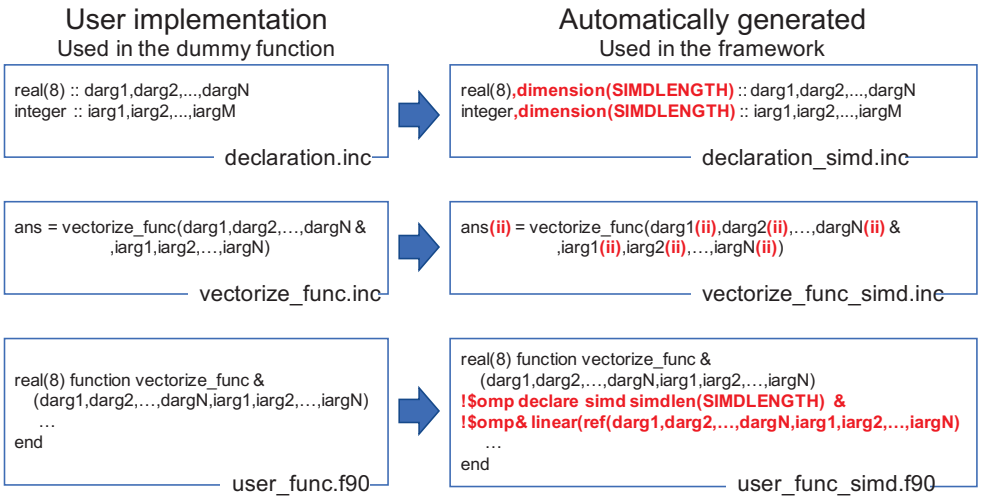

Figure 5.10: The users program automatically transformed at the compile time.

Table 5.1: Processor Specifications

	Processor Name	Number of cores	Peak performance	Length of SIMD unit
BDW	Intel Xeon E5-2695 v4	18	605 GFlops	256 bit
KNL	Intel Xeon Phi 7250	68	3,046 GFlops	512 bit

`-DSIMDLENGTH=4`, and those for KNL were `-align array64byte -xMIC-AVX512 -qopenmp -O3 -fpp -ipo -lm -qopt-report=5 -DSIMDLENGTH=8`.

5.4.2 Hand Tuning Using OpenMP SIMD Directives

To test the compiler vectorizations, we refactored and evaluated two user-defined functions. Vectorization with compiler directives often requires users to converse with the compiler. We tried to vectorize the user-defined functions by preparing the following series of implementations.

H1: Original implementation without compiler directives.

H2: `!$omp simd` directives are inserted above the SIMD target loops of H1.

H3: `!$omp declare simd` directives are inserted in the function shown in Fig. 5.3 and all user-defined functions called from the function of H2 shown in Fig.5.3.

H4: A `simdlen(SIMDLENGTH)` clause is attached to each `!$omp simd` and `!$omp declare simd` directive of H3.

H5: Replace the user-defined functions of H4 with the `set_args` and `vectorize_func` interfaces.

H6: The interfaces `set_args_i` and `set_args_j` are used as alternatives to `set_args` of H5.

H7: `linear` clauses are attached to a `!$omp declare simd` directive of `vectorize_func` of H6.

H8: `uniform` clauses are used as constant variables instead of `linear` clauses of H7.

Implementations H1-H4 are based on the original framework. The differences among these implementations are only in terms of the OpenMP directives. Therefore, users familiar with SIMD can implement H1-H4 with relative ease. Implementations H5-H8 are based on the proposed framework. Specifically, implementation H7 corresponds to the automatically generated program. Note that implementation H8 is more optimized than implementation H7. However, to automatically generate implementation H8, syntactic analysis is required. This will be realized in the future.

Figures 5.11-5.14 show the increase in speed compared to the speed of implementation H1, and Table 5.2 summarizes the elapsed times of implementations H1 and H7. The results discussed in this section are the averages of 10 measurements. As summarized in Table 5.2, although we recommend the BEM-BB H-matrix version, we evaluated the dense matrix version, the performance of which depends to a greater extent on the user-defined function. The main difference between the two functions from the viewpoint of SIMD vectorization is whether the function has a branch. Although the increase in speed in case of the dielectric problem shows a trend similar to that in case of the perfect conductor problem, it is slightly worse owing to the branch divergence caused by the dielectric function. The results obtained by solving the perfect conductor problem on a machine with the KNL processor (Fig.5.11) show that the proposed implementation (H7) achieved

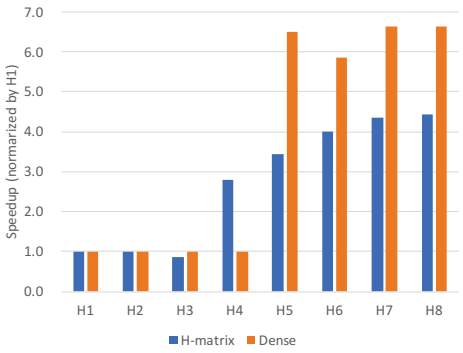

Figure 5.11: Solving perfect conductor problem using KNL processor

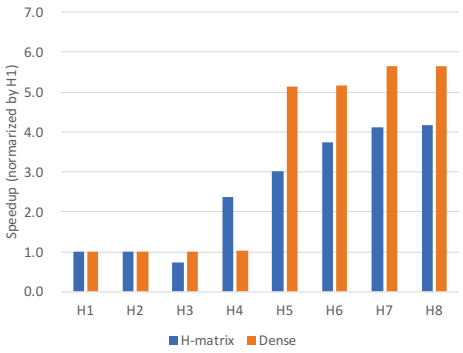

Figure 5.12: Solving dielectric problem using KNL proccesor

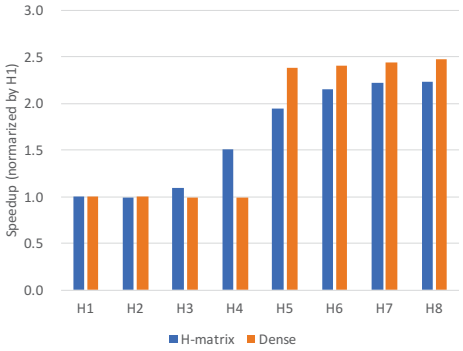

Figure 5.13: Solving perfect conductor problem using BDW processor

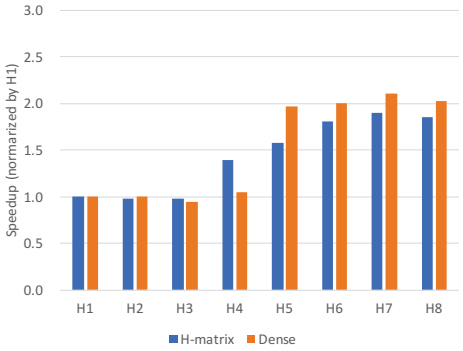

Figure 5.14: Solving dielectric problem using BDW processor

performance improvements of 4.34x and 6.62x compared to implementation H0 for the $\mathcal{H}$-matrix and the dense matrix versions, respectively. The theoretical speedup with SIMD vectorization equals `SIMDLENGTH`, and the results of the dense matrix version demonstrate that the framework improves SIMD vectorization performance considerably. In the results obtained on a machine with the BDW processor (Fig.5.13), implementation H7 achieved performance improvements of 2.22x and 2.44x compared to implementation H0 for the $\mathcal{H}$- matrix and the dense matrix versions, respectively

Table 5.2: The elapsed times of coefficient generation component of original implementation (H1) and implementation of proposed framework (H7)

	Perfect conductor				Dielectric			
	KNL		BDW		KNL		BDW	
	H-matrix	Dense	H-matrix	Dense	H-matrix	Dense	H-matrix	Dense
H1	10.00	215.0	10.51	233.2	13.07	249.5	13.53	265.5
H7	2.307	32.47	4.728	95.61	3.167	44.11	7.140	126.10

5.5 Related Work

The literature contains many studies about software frameworks for parallel PDE solvers of the finite element method, such as GeoFEM [25] and Free FEM++ [11]. Moreover, $\mathcal{H}$-matrices have been used in a few BEM applications [15, 17, 24], and parallelized in their application. Although many frameworks allow for MPI + OpenMP hybrid parallelization, few frameworks support SIMD vectorization, which highly depends on user-defined functions. The main contribution of this study is a SPMD-like SIMD vectorization method that handles data access and computation separately, and hides SIMD-related aspects in the framework. The method uses the characteristics of BEM analysis: the kernel function is relatively computationally intensive, and there exists no data dependency among the calculations of elements of coefficient matrix.

5.6 Chapter summary

We refined the open-source framework for parallel BEM analysis to enhance SIMD vectorizations, which is important for realizing high-performance computing. By using the refined framework design, we could successfully separate SIMD-related aspects from the user-defined function, which depends on target applications. We evaluated the proposed framework by solving two static electric field analysis problems containing different user-defined functions on a BDW processor and a KNL processor. The numerical results demonstrated the improved performance of the framework. Specifically, in solving the perfect conductor problem by using the KNL processor, we achieved performance improvements of 4.34x and 6.62x in the $\mathcal{H}$-matrix case and the dense matrix

cases, respectively.

The main contribution of this paper is separating the SIMD-related aspects from the user-defined function and hiding them to minimize the difficulties associated with SIMD. This SPMD-like SIMD vectorization technique can be used for other applications. In the proposed framework, the arguments of the `vectorize_func` must be scalar variable. This specification is not user-friendly but compiler-friendly. For example, to adjust the user-defined functions in the proposed framework, we separated the vector argument `coordinate(3)` to scalars `x`, `y`, and `z`. This type of transformation is a typical Array of Structure (AoS) to Structure of Array (SoA) transformation. To improve the not user-friendly specification, we will challenge to support the AoS to SoA transformation in future.

Chapter 6

Conclusion

To hide the difficulties of programming for many-core processors, such as load balancing and vectorization, we consider both compiler-directive-based and domain-specific approaches in this dissertation.

As directive-based approaches, we firstly evaluate the OpenACC. We port and optimize both kernel benchmarks and real-wold application code with OpenACC and CUDA to understand the performance, productivity, and portability of OpenACC. We find that data structure transformations, which require structural changes to the code, can effectively create efficient vectorized versions of applications. To enhance the efficient vectorization, we propose a set of extensions of OpenACC directives that abstract the data layout. We then develop a source-to-source translator that supports the proposed directives and evaluate it using two real-world applications, namely UPACS and CCS-QCD. The results show that the performance improved by 23% and 20% compared with the baseline for UPACS and CCS-QCD, respectively.

As domain-specific approaches, we enhance the $\mathcal{H}$ACApK open-source $\mathcal{H}$-matrix framework to work well on many-core processors. In particular, we propose many-core-focused load-balancing-aware parallel ACA algorithms for working with $\mathcal{H}$-matrices. We find that the proposed algorithms improve the performance of $\mathcal{H}$-matrix construction with ACA in all GPU cases, while they don't in all CPU cases. Because the algorithms are abstracted in the framework, framework users can easily select better one. We also find that the framework designs of $\mathcal{H}$ACApK and its wrapper framework BEM-BB prevent the efficient vectorization. To enhance the efficient vectorization, we propose a framework design for abstracting the vectorization process. We then adapt

this framework design to BEM-BB framework. We evaluate the adapted framework using two BEM problems on BDW and KNL. The results show that this approach can offer good vectorization performance while requiring little vectorization knowledge. Specifically, in perfect conductor analyses conducted using $\mathcal{H}$-matrices, the new framework improved performance by 2.22 and 4.34 times compared with the original BEM-BB framework on the BDW and KNL processors, respectively.

These abstractions also enable programs to be ported to different processors with different numbers of cores and vector lengths without sacrificing performance. Even though the optimal data layout, load-balancing algorithm and vector length may differ among processors, our proposals enable the most suitable approaches to be easily selected. We therefore believe that they are one possible answer to the multi- to many-core paradigm shift.

Bibliography

[1] M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, and M. Fatica. Petaflop biofluidics simulations on a two million-core system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1 –12, nov. 2011.

[2] James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski. Openmp for accelerators. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors, *OpenMP in the Petascale Era*, pages 108–121, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Steffen Börm and Joana Bendoraityte. Distributed h^2-matrices for non-local operators. *Computing and Visualization in Science*, 11(4):237–249, Sep 2008.

[4] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[5] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[6] Fiber Miniapp Suite. http://fiber-miniapp.github.io/.

[7] Steffen Börm ; Lars Grasedyck ; Wolfgang Hackbusch. Hierarchical matrices. Technical report, Max Planck Institute for Mathematics in the Sciences, 2003.

[8] W. Hackbusch. A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices. *Computing*, 62(2):89–108, Apr 1999.

[9] W. Hackbusch and B. N. Khoromskij. A sparse h -matrix arithmetic. part ii: Application to multi-dimensional problems. *Computing*, 64(1):21–47, January 2000.

[10] Tianyi David Han and Tarek S Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[11] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[12] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. An openacc extension for data layout transformation. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 12–18, Piscataway, NJ, USA, 2014. IEEE Press.

[13] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:136–143, 2013.

[14] Akihiro Ida, Takeshi Iwashita, Takeshi Mifune, and Yasuhito Takahashi. Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of Information Processing*, 22(4):642–650, 2014.

[15] Takeshi Iwashita, Akihiro Ida, Takeshi Mifune, and Yasuhito Takahashi. Software framework for parallel bem analyses with h-matrices using mpi and openmp. *Procedia Computer Science*, 108:2200 – 2209, 2017.

[16] S. Kurz, O. Rain, and S. Rjasanow. The adaptive cross-approximation technique for the 3d boundary-element method. *IEEE Transactions on Magnetics*, 38(2):421–424, Mar 2002.

[17] S. Kurz, O. Rain, and S. Rjasanow. The adaptive cross-approximation technique for the 3d boundary-element method. *IEEE Transactions on Magnetics*, 38(2):421–424, Mar 2002.

[18] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[19] The Green 500 List. http://www.green500.org/.

[20] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, 2011.

[21] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5dd blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

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

[23] Oakforest-PACS. https://www.cc.u-tokyo.ac.jp/supercomputer/ofp/service/.

[24] Makiko Ohtani, Kazuro Hirahara, Yasuto Takahashi, Takane Hori, Mamoru Hyodo, Hiroshi Nakashima, and Takeshi Iwashita. Fast computation of quasi-dynamic earthquake cycle simulation with hierarchical matrices. *Procedia Computer Science*, 4:1456–1465, 2011.

[25] Hiroshi Okuda, Kengo Nakajima, Mikio Iizuka, Li Chen, and Hisashi Nakamura. Parallel finite element analysis platform for the earth simulator: Geofem. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya, editors, *Computational Science — ICCS 2003*, pages 773–780, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[26] OpenACC-standard.org. The openacc application programming interface. 2011.

[27] OpenACC-standard.org. The openacc application programming interface, http://www.openacc.org/sites/default/files/openacc.1.0_0.pdf, 2011.

[28] Pascal Architecture Whitepaper. http://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[29] ppOpen-HPC. Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT), http://ppopenhpc.cc.u-tokyo.ac.jp/ppopenhpc/.

[30] Greg Ruetsch and Massimiliano Fatica. Cuda fortran for scientists and engineers. *Nvidia Corporation, Santa Clara, CA*, 2011.

[31] Karl Rupp. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.

[32] Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Berlin Heidelberg, 2003.

[33] Takashi Shimokawabe, Takayuki Aoki, Junichi Ishida, Kohei Kawano, and Chiashi Muroi. 145 tflops performance on 3990 gpus of tsubame 2.0 supercomputer for an operational weather prediction. *Procedia Computer Science*, 4(0):1535 – 1544, 2011.

[34] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016.

[35] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–72, 2010.

[36] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11, May 2012.

[37] Ryoji Takaki, Kazuomi Yamamoto, Takashi Yamane, Shunji Enomoto, and Junichi Mukai. The development of the upacs cfd environment. In Alex Veidenbaum, Kazuki Joe, Hideharu Amano, and Hideo Aiso, editors, *High Performance Computing*, volume 2858 of *Lecture Notes in Computer Science*, pages 307–319. Springer Berlin / Heidelberg, 2003.

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

[39] The OpenMP API specifications for parallel computing. `https://www.openmp.org/specifications/`.

[40] TSUBAME3. `http://www.gsic.titech.ac.jp/tsubame3`.

[41] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter Mey. Openacc — first experiences with real-world applications. In Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012.

[42] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[43] Takeshi Yamada, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A case study of user-defined code transformations for data layout optimizations. In *2015 Third International Symposium on Computing and Networking (CANDAR)*, pages 535–541. IEEE, 2015.

[44] Hiroyuki Yamazaki, Shunji Enomoto, and Kazuomi Yamamoto. A common cfd platform upacs. In *High Performance Computing*, pages 182–190. Springer, 2000.

[45] Rio Yokota, L.A. Barba, Tetsu Narumi, and Kenji Yasuoka. Petascale turbulence simulation using a highly parallel fast multipole method on gpus. *Computer Physics Communications*, 184(3):445 – 455, 2013.