Performance Portability Strategies for Computational Fluid Dynamics (CFD) Applications on HPC Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Pei-Hung Lin

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Professor Pen-Chung Yew, Advisor
Professor Paul R. Woodward, Co-Advisor

June, 2013

© Pei-Hung Lin 2013
ALL RIGHTS RESERVED

Acknowledgements

First and foremost, I would like to thank my advisors, professor Paul R. Woodward and professor Pen-Chung Yew for their guidance in my doctoral program. Professor Woodward continuously inspired and encouraged me to pursue my research career. Professor Yew with his solid background in architecture and compiler guided me and taught me in academic research. They selflessly advised me conducting researches and shared their experiences in research, in work, and also in their lives. I also would like to thank my thesis committee members – professor Antonia Zhai and professor Eric Van Wyk for their insight and efforts in reviewing my dissertation.

Furthermore, I would like to thank my colleagues and mentors in my graduate study. Dr. Dan Quinlan, and Dr. Chunhua Liao provided me the internship opportunities to work at Lawrence Livermore National Laboratory and will continue to support me in my post-doctoral career. Dr. Jagan Jayaraj (now at Sandia National Laboratory) and Michael Knox were my long-term colleagues in both study and work and we exchanged lots ideas in achieving our research goal. Dr. Ben Bergen and Dr. William Dai at Los Alamos National Laboratory provided me strong support and great assistant during my internship at LANL. I would also express my gratitude to other professors, researchers, and colleagues I have interacted with: James Greensky (Intel), Dr. Shih-Lien Lu (Intel) , Dr. Tong Chen (IBM), Professor Falk Herwig (University of Victoria), Dr. David Porter (University of Minnesota), Sara Anderson (Cray), Mark Straka (NCSA), Dr. Guojin He (Mathworks), Dr. Yangchun Luo (AMD), Dr. Venkatesan Packirisamy (nVidia), Professor Dave Yuan (University of Minnesota), Professor Wei-Chung Hsu (NCTU, Taiwan), Karl Stoffels, and David Sanchez for their support and encouragement.

Last, but not least, I want to dedicate all this work to my parents, my brother, and my lovely wife. I would not be able to complete my PhD study without their unconditional love and support.

Abstract

Achieving high computational performance on large-scale high performance computing (HPC) system demands optimizations to exploit hardware characteristics. Various optimizations and research strategies are implemented to improve performance with emphasis on single or multiple hardware characteristics. Among these approaches, the domain-specific approach involving domain expertise shows its high potential in achieving high performance and maintaining performance portability.

Deep memory hierarchies, single instruction multiple data (SIMD) engines, and multiple processing cores in the latest CPUs pose many challenges to programmers seeking significant fractions of peak performance. Programming for high performance computation using modern CPUs has to address thread-level parallelization on multiple cores, data-level parallelization on SIMD engines, and optimizing memory utilization for the multi-level memories. Using multiple computational nodes with multiple CPUs in each node to scale up the computation without sacrificing performance increases programming burden significantly. As a result, performance portability has become a major challenge to programmers. It is well known that manually tuned programs can assist the compiler to deliver the best performance. However, generating these optimized codes requires deep understanding in application design, hardware architecture, compiler optimizations, and knowledge in the specific domain. Such manually tuning process has to be done for each new hardware design. To address this issue, this dissertation proposes strategies that exploit the advantages of domain-specific optimizations to achieve performance portability.

This dissertation shows the combination of the proposed strategies can effectively exploit both the SIMD engine and on-chip memory. High fraction of peak performance can be achieved after such optimizations. The design of the pre-compilation framework makes it possible to automate these optimizations. Adopting the latest compiler techniques to assist domain-specific optimizations has high potential to implement sophisticated and legal transformations. This dissertation provides a preliminary study using

polyhedral transformations to implement the proposed optimization strategies. Several obstacles need to be removed to make this technique applicable to large-scale scientific applications. With the research presented in this dissertation and suggested tasks in the future work, the ultimate goal to deliver performance portability with automation is feasible for CFD applications.

Contents

Acknowledgements i

Abstract iii

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Dissertation Objective and Summary 3

1.2 Related Work 4

1.2.1 Domain-specific optimization 4

1.2.2 Source-to-source framework 5

1.2.3 User-directed optimization 5

1.2.4 SIMD optimization 6

1.2.5 Implementation on HPC 7

1.3 Dissertation Contributions 8

1.4 Dissertation Outline 9

2 HPC System Design 10

2.1 CPU design in HPC 11

2.1.1 Homogeneous multi-core CPU 12

2.1.2 IBM Cell processor 14

2.1.3 General-purpose graphics processing unit (GPGPU) 15
2.1.4 Many-core CPUs 16
2.2 Barriers to high performance computing 17

3 Computational Fluid Dynamics 22
3.1 CFD applications 23
3.1.1 Rayleigh-Taylor Instability 23
3.1.2 PPM advection 24
3.1.3 Stellar convection 25
3.1.4 PPM single-fluid 25
3.1.5 Two-fluid PPM gas dynamics with strong shocks 26
3.1.6 Inertial confinement fusion (ICF) 26

4 Optimization for SIMD Engine 28
4.1 Challenges in vectorization 30
4.2 Implementation of vectorization framework 33
4.3 Performance result 40
4.4 Contribution 43

5 Optimization for On-Chip Memory 45
5.1 Memory organization 46
5.2 Subroutine inlining 49
5.3 Loop transformations and computational pipelining 50
5.4 Array size contraction 55
5.5 Performance result 56

6 Optimization for Limited Memory Bandwidth 63
6.1 Computational intensity 64
6.2 Briquette-pencil pipeline 66
6.3 Briquette-block pipeline (3-dimensional pipeline) 68
6.4 Performance result 71

6.4.1 PPM advection . . . 72
6.4.2 PPM-star simulation . . . 73
6.4.3 Inertial confinement fusion . . . 74
6.5 Discussion . . . 74

7 Transformation Framework **77**
7.1 Source-to-Source transformation . . . 77
7.2 Source-to-Source implementation . . . 79
7.2.1 ANTLR-based tool . . . 79
7.2.2 ROSE compiler . . . 81
7.3 User-directive interface . . . 82

8 Discussion and Performance Study **84**
8.1 Viewing modern processors from a Cell programmer's perspective . . . 84
8.2 Performance study . . . 89
8.2.1 Multi-core CPUs . . . 90
8.2.2 GPGPUs . . . 93
8.3 Characterizing the multifluid gas dynamics application code . . . 98

9 Study With Latest Compiler Technique **105**
9.1 Polyhedral framework . . . 106
9.1.1 Methodology . . . 106
9.1.2 Polyhedral implementation . . . 108
9.2 Polyhedral transformation . . . 108
9.3 Comparison and Discussion . . . 111
9.3.1 Experiment setting . . . 111
9.3.2 Performance evaluation . . . 113
9.3.3 Customized experimental setting . . . 119
9.3.4 Conclusion . . . 123

10 Conclusion and Future Work **124**

10.1 Future Work 127

10.1.1 Optimization development 127

10.1.2 Extending the framework to cover more CFD applications 127

10.1.3 Performance portability for future HPC systems 128

References **129**

Appendix A. Glossary and Acronyms **140**

A.1 Glossary 140

A.2 Acronyms 141

Appendix B. PPM example code **143**

List of Tables

2.1 List of multicore processors 13
6.1 3-D pipelining 75
7.1 List of directives used in optimization 83
A.1 Acronyms 141

List of Figures

2.1 Projected HPC performance development (source from Top500.org) . . . 11
2.2 Multi-core CPU statistic in Top 500 (Nov. 2012) 12
2.3 IBM Cell processor diagram 14
2.4 Nvidia Fermi SM design 16
2.5 Intel MIC design 17
2.6 Ratio between peak floating-point performance and sustainable memory bandwidth. [1] 20
3.1 Visualization from RT instability simulation 23
3.2 Visualization from advection simulation 24
3.3 Visualization from stellar convection simulation 25
3.4 Visualization from ICF simulation 27
4.1 Illustration of SIMD width 29
4.2 Overview of vectorization framework 34
4.3 Example for Strip-mining 37
4.4 Example for code generation 38
4.5 Code generation for conditional vector merge function 38
4.6 Vectorization comparison for IBM Blue Gene/Q processor 43
5.1 Illustration of data reorganization 48
5.2 Illustration of 5-point stencil example 52
5.3 The pipeline illustration 56
5.4 Performance comparison with no compiler optimization on Intel Nehalem 58
5.5 Performance comparison of on-chip memory optimization 59

5.6 Performance comparison of vectorization 60
5.7 Speedups achieved by the on-chip memory optimization plus vectorization over the performance of the original code. 61
6.1 Performance impact from computational intensity 65
6.2 Stencil computation 67
6.3 Briquette-pencil pipeline 67
6.4 Parallelization briquette-pencil pipeline 68
6.5 3-D pipeline 70
7.1 domain-specific transformation framework 78
8.1 PPM advection performance on CPUs 91
8.2 PPM single-fluid performance on CPUs 93
8.3 PPM advection performance on GPUs 94
8.4 illustration of the GPU implementation 96
8.5 PPM single-fluid performance on GPU 97
8.6 Characterization of the multifluid PPM gas dynamics code and its performance on a multicore CPU. Speeds are quoted in Gflop/s/core, measured as the code is executed in parallel by all 6 cores of the Westmere CPU. . 101
8.7 Characterization of a portion of the PPM gas dynamics code and its performance on a many-core GPU. 104
9.1 Performance results with polyhedral optimization (Intel Sandy Bridge w/ 256-bit SIMD, 2.0GHz) 114
9.2 Performance results with polyhedral optimization (Intel Nehalem w/ 128-bit SIMD, 2.93 GHz) 114
9.3 Performance comparison with polyhedral optimization (Sandy Bridge w/ 256-bit SIMD, 2.0 GHz) 122
9.4 Performance comparison with polyhedral optimization (Nehalem w/ 128-bit SIMD,2.93 GHz) 122

Chapter 1

Introduction

The main objective in high performance computing research is to achieve high computational efficiency from large-scale computing systems. HPC systems passed the petaflop milestone in the last decade, and is moving toward the exascale computing era. Inspection of the Top 500 list of large-scale computing systems shows the achievement in providing high computational power from latest large-scale systems. However, it also shows the disparities in system design, especially in the CPU architecture. Tuning applications to achieve high performance on multiple platforms is a significant burden on programmers. Achieving performance portability becomes immensely challenging.

Modern HPC systems pursue higher computational power by increasing the number of computational nodes and providing more powerful CPUs. Having a large number of computational nodes in an HPC system poses a significant challenge in scalability, communication and synchronization from the perspective of application design. It also raises the resilience concern from the perspective of system design. However, achieving high performance on such diverse computing processor designs is the most challenging task in performance portability.

The demand for powerful computing system has spawned numerous CPU architectural innovations in the past decades. They include multiple cores per CPU, multiple simultaneous threads per core, and, especially with GPUs, highly complex memory hierarchies. Those innovations focus on achieving high computational performance and

low power consumption. Those innovations in computer architecture design have posed great challenges in application portability. Language extensions and new programming models have been developed to accommodate new features in hardware systems. However, to apply these new programming features and keep the same application running on multiple platforms, programmers must make significant effort to pursue performance portability. Novel programming models have been proposed with new programming languages and runtime library support. The purpose is to overcome the disparities in modern CPUs and simplify the programming effort. These new models have a high potential to achieve function portability, but still lack capabilities to exploit hardware characteristics for high computational efficiency. Through this approach, an application optimized for one platform might deliver only suboptimal performance on other platforms. The design of programming models is often purposely decoupled from the hardware design.

The SIMD engines in modern CPU and GPU cores and on-chip memory are keys to obtaining high performance for scientific application codes. These common elements of all present computing devices make performance portability possible. However, achieving high performance requires optimizations to exploit these hardware features. An optimization strategy that is integrated with domain expertise can reveal more optimization opportunities in scientific applications than a strategy relying only on an analytical model. Suboptimal performance is delivered by general compiler optimizations, because they do not exploit domain knowledge. There is a necessity to develop domain-specific optimizations to achieve high performance for scientific application codes. Integrating specialized optimizations into a compiler framework automates the optimization process and improves programming productivity. Joint efforts between compiler development and domain expertise can better deliver performance portability.

1.1 Dissertation Objective and Summary

The objective of this dissertation is to research on strategies that can achieve performance portability for CFD applications running on HPC systems. A code transformation framework is proposed and implemented to automate the domain-specific optimizations.

The first section of this dissertation introduces the processor design in latest HPC systems and the CFD applications used in this study. With the diverse hardware designs, a list of challenges to achieve high performance and portability is identified based on the latest CPUs. Our research team has applied the proposed optimizations to the selected CFD applications. Large-scale simulations using these application codes were performed and studied on multiple HPC systems. These applications exemplify the potential of performance portability on scientific applications that use structured grids.

In the second section, three optimizations are proposed, developed and studied for the CFD applications using structured grids. Prior research in general optimizations for scientific applications focused on analytical models to identify optimizations and perform program transformations on a compiler's intermediate representation. The proposed strategies integrate domain expertise and compiler expertise to discover the maximal optimization opportunities. Optimizations for SIMD engine, on-chip memory, and limited memory bandwidth are found profitable for the CFD applications chosen in this dissertation.

The third part of this dissertation presents a code transformation framework. Prior research in code transformations focused on language translation, auto-parallelization, and optimization validation. We have devised and implemented a framework to perform domain-specific optimizations and tackle the language barrier. Thanks to the growing research in source-to-source compilers, the need is foreseen to integrate the proposed framework into a state-of-the-art source-to-source compiler.

We have applied the optimization strategies, through manual and automated transformation, to a subset of CFD applications. The SIMD engine and on-chip memory are crucial elements in CPU to achieve high performance. Our performance study shows

the proposed optimizations effectively exploit both SIMD engine and on-chip memory. These strategies, with the assistance from a code transformation framework, deliver a high fraction of peak performance on multicore CPUs from different vendors. The performance measurements are comparable to, or higher than, other CFD applications implemented on GPUs.

In the final section, a study on the latest compiler techniques is presented. This study discusses the potential of using latest analytical compiler to implement the domain-specific optimizations presented in this dissertation. The technique chosen in this study has strength in data dependence analysis and loop transformations. To fully adopt the latest compiler techniques for research in performance portability, a list of tasks to be completed in order to overcome existing challenges is presented. Possible future work for the research in performance portability is identified to conclude this dissertation.

1.2 Related Work

1.2.1 Domain-specific optimization

Domain-specific optimizations can be implemented through compilers and programming languages. The Broadway compiler[2] relies on an annotation language to implement library-level optimizations. The annotations convey domain knowledge to a domain-independent and configurable compiler framework. The optimization framework presented in this dissertation is different in its application-level optimizations and is compiler independent. Spiral [3, 4] is a code generation framework for digital signal processing (DSP) algorithms. Its feedback-driven optimizer drives the framework to generate alternative implementations, and a searching and learning feature explores alternative implementations to find the best code transformations. Liszt [5] is a domain-specific language for constructing mesh-based PDE solvers. The unstructured mesh target is different from the structured grids in this dissertation.

1.2.2 Source-to-source framework

Source-to-source transformations have been implemented in many research compiler platforms. The Cetus compiler [12-13] is one example that focuses on auto-parallelization for applications written in the C language. Mint [6], based on the ROSE [7] compiler framework, provides a source-to-source compiler to generate CUDA programs for 3-D stencil methods. G-ADAPT [8], another sourced-to-source translator, addresses the influence of program inputs on GPU program optimizations. It relies on statistical learning techniques to generate optimized codes in CUDA. Although these efforts target different architectures, they also exploit domain knowledge to apply code transformations.

1.2.3 User-directed optimization

User-directed optimization and interactive compilation are adopted by several compiler research projects. Extendable pattern-oriented optimization directives (EPOD)[9] provides a pattern-oriented optimization framework built on top of the Open64 compiler[10]. There are two interfaces in EPOD to direct the compilation. Programmers use pattern-oriented directives to perform algorithm-specific optimizations. Compiler developers can use low-level scripts, called EPOD scripts, to define customized optimization schemes. POET[11] is a scripting language designed for parameterizable source-to-source transformations. A ROSE compiler based framework exploits POET for programmable control on compiler optimizations [12]. The compiler automatically generates a POET script for the default optimizations. Programmers can modify the script to change the order of transformations or to integrate additional optimizations. CHiLL[13] is a framework built upon the Omega Library for polyhedral transformations. Transformation scripts are provided in CHiLL to perform primitive optimizations. The scripts assist programmers to search and select optimal compiler optimizations from a rich set of code transformations. An auto-tuning framework[14] is built in CHiLL to support the optimization of applications and computational libraries. All these existing projects exploit scripting languages to provide user input to the optimization process. The optimization scripts have the capability to identify the code regions for optimizations and the order of

the program transformations. However, the common drawback among them is the complexity in the optimization script. There could be a steep learning curve for developers to master the scripting language for the optimal optimization. The number of lines in the scripts to perform a combination of optimizations can be as big as the source code of the applications itself. In contrast to our proposed research, these existing projects focus on libraries or small codes.

1.2.4 SIMD optimization

Despite over 40 years of research, vectorization is still an active research area in compiler optimization and HPC research. With the evolution in processor design, vectorization migrates its implementation from the Cray-style long vector instructions into the SIMD-style short vector instructions. One major research focus is in automatic vectorization for both long vector and short vector implementations [15, 16, 17, 18, 19, 20]. Other research looks into compiler design to improve SIMDization [21]. Research contributions from the IBM group [22, 21, 23] and the VAST compiler [24] address the memory alignment restrictions of SIMD engines. Other research focused on the evaluation for the vectorizing compilers. In the 80's, Callahan, Donagarra and Levine designed the TSVC benchmark written in Fortran language to evaluate vectorizing compilers [15]. The benchmark is over 20 years old but still serves as a benchmark for modern processors with SIMD engine design. Maleki and his group evaluated some state-of-the-art vectorizing compilers (for short-vector style SIMD engines) and found only a subset of of loops inside synthetic benchmarks are vectorized [25]. Today's best vectorizing compilers still fail to vectorize many patterns. Hand analyses and manual transformations are still necessary for exploiting SIMD engines.

Many previous studies [26, 23, 27, 28] have used a source-to-source or preprocessing approaches to vectorization. We only elaborate on a few representative studies here. SWARP [27] is a retargetable preprocessor for multimedia application. It uses loop distribution, unrolling and pattern matching for exploiting SIMD instructions. A sophisticated rewrite system that accepts annotated C programs is used to retarget code to different machines. Krzikalla et al. [28] created a clang [29]-based source-to-source

translator, called Scout, to vectorize C loops. Their tool uses source pragmas, loop simplification, and unroll-and-jam for vectorization. Configuration files using C++ syntax are used to support multiple architectures. SAC [30] is another source-to-source compiler aimed at generating SIMD instructions for multiple platforms. It defines a meta instruction set to represent different instruction sets. Python has also been used to translate a pseudo instruction set into the intrinsics of a target processor.

1.2.5 Implementation on HPC

After Compute Unified Device Architecture (CUDA) was introduced for GPGPU programming development, a large body of work has been done to support code translation into CUDA. The majority of these projects focused on the usage of CUDA parallelism and language translation [31, 32, 33]. Relatively few of them address optimizations for specific GPGPU hardware features. Also, most of the studies used small kernels, not real applications. As the exascale era is expected to arrive in the upcoming decade, researchers are aiming to find the most feasible approach for hardware, compiler, software, and programming model that can provide strong scalability for applications. Accelerators [34, 35], like GPGPUs and many-core processors, are potential target machines.

GPU parallel computing has become a popular research topic in the past few years and many studies have shown that GPUs can deliver substantial speedups compared to CPUs. Lee and his group from Intel [36] run a series of throughput kernels and compare the performance delivered by both CPU and GPU. They found an average speedup of 2.5 is achievable by the GPUs. To achieve high performance on a GPU, applications have to adopt a new language and a programming model. Garland [37] presents experiences with parallel computing on GPUs and addresses how to exploit parallelism on the GPU for different research domains. Lee et al. [32], and Han et al. [38] develop translation frameworks to ease the programming burden of using GPU's native programming languages. Researchers have explored multiple approaches to improve GPU performance[39, 40, 41, 42, 43]. Sundaram and Raghunathan [40] use domain-specific parallel programming templates to enable high performance on GPUs. Memory optimization [39], data prefetching [41], pipelined execution [42], and loop optimizations

[43], are also discussed in different perspectives for GPUs. Most performance results reported for GPUs are throughput kernels with relatively small code size and with characteristics that suit a GPU programming model. Nevertheless, some research groups successfully move full scientific applications to GPU platforma. Quantum Chromodynamics (QCD) research from NCSA [44, 45] reports about 100 Gflops/s on a single Nvidia GTX280 GPU. Shimokawabe et al. [46] program the high-resolution weather prediction model ASUCA to a cluster equipped with 4000 Nvidia Fermi GPUs and a 15 Tflops/s performance is achieved for a $6956 \times 6052 \times 48$ mesh. A $1.3\times$ speedup on an earlier model of GPUs, the Nvidia GTX8800, is reported for the Weather Research and Forecast (WRF) by Michalakes and Vachharajani [47].

1.3 Dissertation Contributions

This dissertation has made several contributions to achieve performance portability for CFD applications, with emphases on optimization designs and implementation of code transformation framework.

1. This dissertation proposes three optimizations that help CFD applications to achieve high performance. These optimizations include vectorization for SIMD engines, optimization and memory footprint reduction for on-chip memory, and increase of computational intensity.

2. This thesis presents a code transformation framework to perform the domain-specific optimizations. With assistance from this framework, programmers can be relieved from the burden of tedious and error-prone optimizations involved.

3. A series of implementations and performance studies are presented in this dissertation. The results show the performance portability for CFD applications with structured grids is feasible.

4. A study on state-of-the-art compiler techniques is presented in this dissertation. Strengths and challenges in latest optimization techniques are discussed.

5. This dissertation proposes a co-design of domain expertise and compiler expertise for the development of future HPC applications.

1.4 Dissertation Outline

The organization of the rest of this dissertation is outlined as follows:

- Hardware platforms and software applications
  - Chapter 2 presents the features of current CPU designs on HPC systems.
  - Chapter 3 briefly introduces the CFD applications used in this dissertation.
- Optimization design
  - Chapter 4 presents the vectorization strategies.
  - Chapter 5 introduces the optimizations for on-chip memory.
  - Chapter 6 shows the strategy of increasing computational intensity to tackle limited memory bandwidth.
- Code transformation framework and performance study
  - Chapter 7 demonstrates the design and implementation of a code transformation framework, as well as the ideal infrastructure for future development.
  - Chapter 8 discusses the implementations and performance studies on the latest HPC systems.
- Discussion and conclusion
  - Chapter 9 presents a study using existing compiler techniques to implement the proposed optimizations.
  - Chapter 10 concludes this dissertation and lists the future work.

Chapter 2

HPC System Design

The demand of large computing power to perform scientific computation and simulation drives the innovations in HPC system design. Reviewing the HPC history, Cray 2 achieved the Gigascale (10^9) milestone in 1985, the Intel ASCI Red System passed Terascale (10^{12}) in 1997, and IBM Roadrunner System delivered Petascale (10^{15}) computing in 2008. Following the trend in performance increase, the next milestone of Exascale (10^{18}) computing is likely to be in the upcoming decade (Fig. 2.1). The design of HPC systems scales up in size as well as the density of computational power to achieve higher peak performance. With the increase in theoretical computational power, the power consumption of HPC system has to be maintained within an manageable limit. The CPU device design has to provide both high computational power and high efficiency in power consumption. The HPC community now finds itself in the midst of a revolution in CPU device design. There are numerous CPU architectural innovations, commonly seen in HPC systems, including multiple cores per CPU, multiple simultaneous threads per core, and highly complex memory hierarchies. This chapter summarizes these innovative hardware designs and the CPU devices in latest HPC systems. The optimization challenges for these hardware designs will also be discussed.

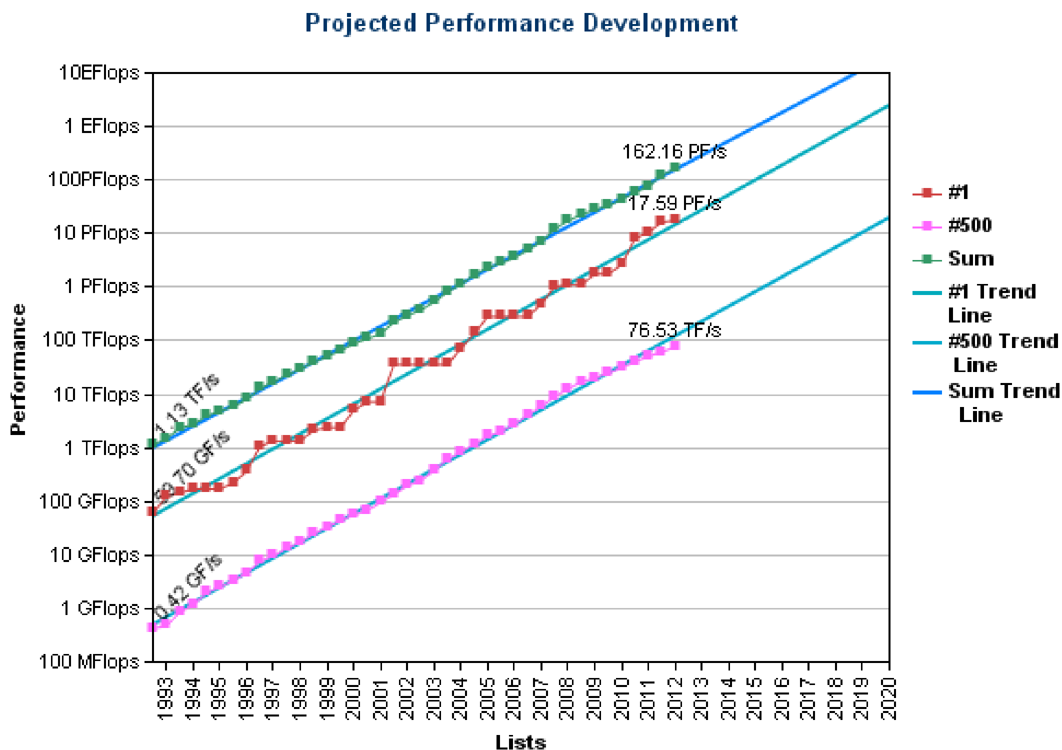

Figure 2.1: Projected HPC performance development (source from Top500.org)

2.1 CPU design in HPC

The design of micro architecture continues to evolve to approach high theoretical peak performance. Inspecting the latest HPC systems on Top 500 list, the processors used in HPC systems can be categorized in different approaches. Majority of system designs choose multi-core processors. An increasing number of systems start to adopt innovate designs using many-core processors, co-processors, or accelerators. Based on the design of multi-cores, there are heterogeneous and homogeneous multi-core processors. The multi-core and many-core processors differ in the number of processing units on a single die. Coprocessors or accelerators appear in HPC systems to supplement the computing capability of the primary, or host, processors. In this dissertation, four processor designs,

used in different HPC systems, are chosen for the study. A brief description of these designs is presented in this section.

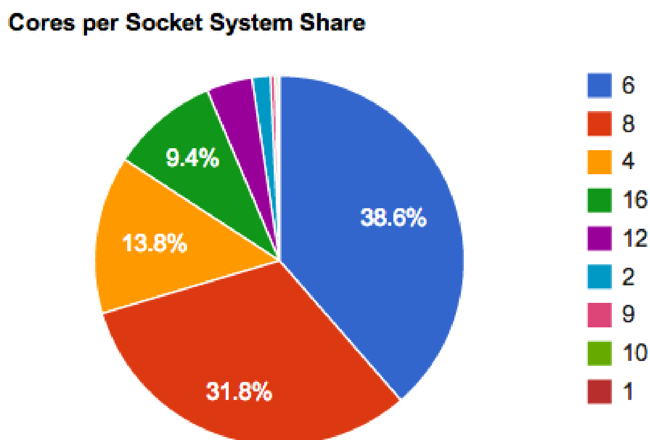

Figure 2.2: Multi-core CPU statistic in Top 500 (Nov. 2012)

2.1.1 Homogeneous multi-core CPU

For years, processor design continued to increase clock rates and instruction-level parallelism to achieve high single-thread performance. With the limitation in power dissipation, multicore processor design dominates CPU manufacturing industry and continues to follow Moore's law. Inspecting the current Top 500 list, processors with 6 or 8 processing units (cores) dominate the HPC systems (shown in Fig. 2.2). The majority of the multicore processors use homogeneous designs with multiple identical cores in a single socket. In addition, multicore CPUs mixed with simultaneous multi-threading (SMT), on-chip memory hierarchies, and single-instruction-multiple-data (SIMD) engines provide the capabilities to deliver high performance for HPC systems.

Multicore processors used in this dissertation contain 4 to 8 cores in a single socket. Each core in either of these CPUs has a 4-way or 8-way SIMD engine that operates on 128-bit or 256-bit operands. With the capability to perform a fused multiply-add

operation in one clock cycle, these CPUs can execute 8 to 16 single precision floating point operations in a single clock cycle. SMT is supported in all chosen multicore processors. The design of SMT allows multiple threads to share hardware resources and to execute concurrently. Multiple levels of cache memories are equipped in these processors. Private L1 caches (data and instruction) and private L2 caches are common components in multicore processors. Most of the designs have a much larger last-level cache, and some have shared and partitioned L2 cache memory. All processors have different combinations of cache policies, cache associativities, memory latencies, and memory bandwidth. The SIMD engine and multi-level cache memory are common components in these latest multicore processors. Table 2.1 lists in detail all the multicore processors used in this dissertation.

processor	cores	clock rate	cache hierarchy	SMT	SIMD
Intel Nehalem	4	2.93 GHz	64 KB L1 (D+I) 256 KB L2 8 MB L3	SMT 2	128 bit
Intel Westmere	6	2.93 GHz	64 KB L1 (D+I) 256 KB L2 12 MB L3	SMT 2	128 bit
Intel Sandy Bridge	4,8	2.0 GHz	64 KB L1 (D+I) 256 KB L2 20 MB L3	SMT 2	256 bit
Intel MIC	60,61	1.1 GHz	64 KB L1 (D+I) 512 KB L2	SMT 4	512 bit
AMD Interlargos	8	2.3 GHz	64 KB L1 (D+I) 256 KB L2 32 MB L3	SMT 2	256 bit
IBM Power 7	8	3.8 GHz	64 KB L1 (D+I) 256 KB L2 32 MB L3	SMT 4	128 bit
IBM Blue Gene/Q	16	1.6 GHz	32 KB L1 (D+I) 2 MB L2	SMT 4	256 bit

Table 2.1: List of multicore processors

2.1.2 IBM Cell processor

The IBM Cell processor [48] is a heterogeneous multi-core processor that includes one Power Processor Unit (PPU) and eight synergistic processing units (SPU) connected by a high-bandwidth Element Interconnect Bus (EIB). Fig. 2.3 shows the design of of the Cell processor and the interconnections among all processing units.

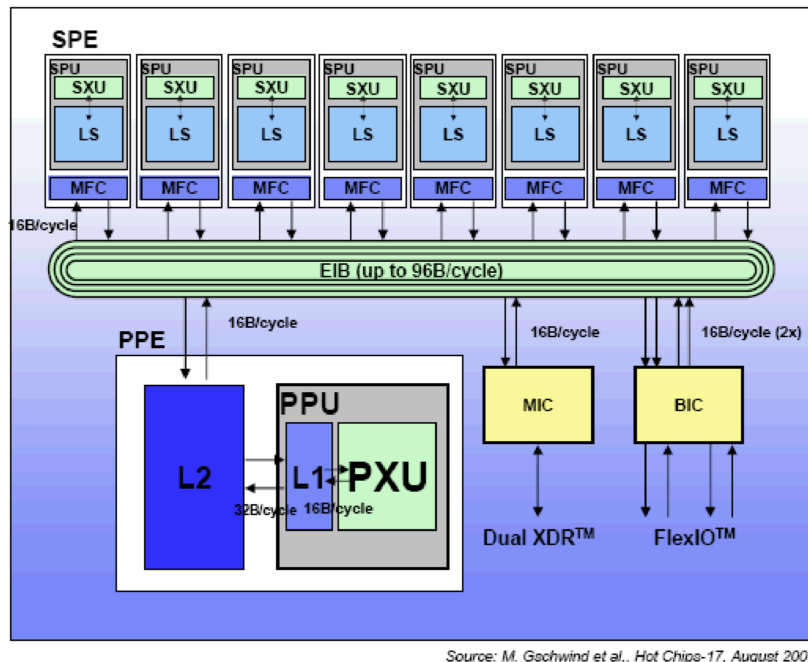

Figure 2.3: IBM Cell processor diagram

Each SPU performs computation in the SIMD fashion on 128-bit data packed with 4 single-precision elements or 2 double-precision elements. At 3.2 GHz clock speed, the 4-way SIMD engine in a SPU delivers 25.6 Gflop/s single-precision peak performance. There are 128 128-bit registers in the register file and a 256 KB local memory in each SPU. The SPU's 256 KB local memory supports 16-byte accesses of memory instructions and 128-byte accesses for instruction fetch and direct memory access (DMA) transfer. The DMA engine equipped in the Cell processor transfers data between local memory and the system memory. The 128-byte padded and aligned DMA transfer can be initiated from both SPU and PPU. All the data and instructions are hosted in the 256 KB

memory space and require software control to transfer them to and from the system memory.

The PPU is a 64-bit Power Architecture processor with two-level cache memories. The PPU is responsible for running operating system and coordinating the eight computational threads spawned on the eight SPUs. The heterogeneous design of the Cell processor naturally divides the program execution into two parts. The PPU is in charge of the logistical operations and the eight SPUs are assigned to perform the computation-intensive work load. The eight SPUs provide the main computational power in the Cell processor [49]. The Cell processor is the core computational unit in the IBM Roadrunner system. Utilizing quad-core AMD processors to support cross-network communication, the Roadrunner system was the first HPC system to achieve the 1 petaflop performance milestone.

2.1.3 General-purpose graphics processing unit (GPGPU)

The massive number of threads and high theoretical peak performance in GPGPU design has increased the usage of GPGPUs in HPC systems. With the sophisticated thread-switching mechanism in GPGPUs, the massive number of GPU threads can effectively hide the memory latency. In addition, high memory bandwidth supported in GPGPUs reduces the performance impact caused by the "memory wall". Similar to the SIMD support in CPUs, GPGPUs implement the single-instruction-multiple-threads (SIMT) parallelism. A group of GPU threads execute the same instruction with multiple data segments simultaneously. SIMT has flexibility to execute discrete data. However, a performance penalty is applied for fetching non-coalesced data to the GPGPU cores.

The Nvidia Fermi GPU is chosen as the GPGPU example in this dissertation. Each GPU contains 14 streaming multiprocessor (SMs). Each SM has a variety of on-chip memory stores each with different uses and limitations. There is an 8 KB cache working set for constant memory, 64 KB configurable on-chip memory (L1 cache memory and shared memory) in addition to a large number of registers. A read-only texture cache is preserved for graphic processing and a L2 cache memory can assist the memory operations for GPGPU. Fig. 2.4 shows the high-level overview of the Fermi SM design.

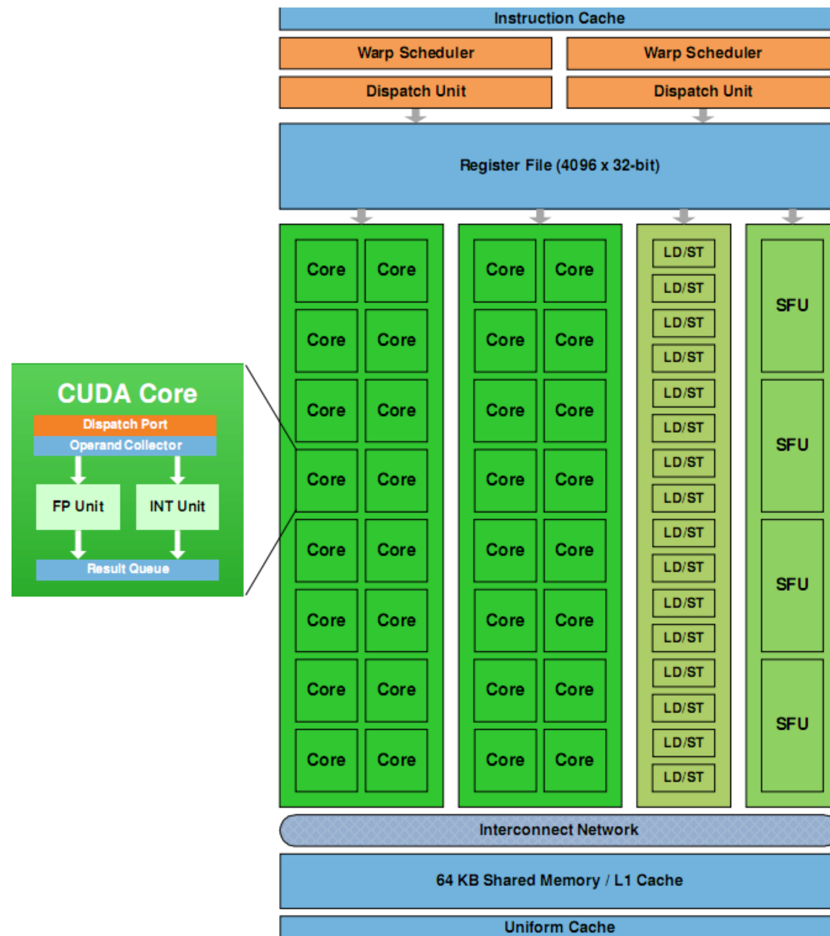

Figure 2.4: Nvidia Fermi SM design

To program Nvidia GPUs, programmers have to rely on the Compute Unified Device Architecture (CUDA) software development kit. Programmers also manually specify a host program, which performs the setup, I/O and logistic functions, and computational kernels, which consist of computation-intensive codes off-loaded to the GPUs.

2.1.4 Many-core CPUs

Intel Many Integrated Core Architecture (MIC) is a multiprocessor derived from the earlier Intel Larabee many-core architecture. The processor design incorporates more than 30 in-order cores at a low clock speed. Each core supports SMT and allows up

to 4 threads. There is a 512-bit SIMD engine equipped in each individual core and it can perform execution on 16 single-precision floating point data concurrently. Similar to other Intel x86 microprocessors, there is a 64 KB L1 cache (data and instruction) private to each core. Each core is assigned a 512 KB L2 cache and all L2 caches are connected by a ring interconnection to make them shareable among all processing units. Fig. 2.5 illustrates the design of MIC architecture. The design of MIC is considered to be a mixture between a GPGPU and a CPU. Similar to GPGPU, MIC is developed as a coprocessor and built on a PCI-E board. It requires software control to offload the data to the memory through the PCI-E channel. Similar to multicore CPUs, the core design is derived from an earlier generation of x86 design. The programming model used in multicore CPUs can be applied to MIC with a few modifications. With a larger number of processing units and a wider SIMD capability, MIC delivers its theoretical peak performance over 1 TFlop/s and is considered comparable with the GPGPUs in the same generation.

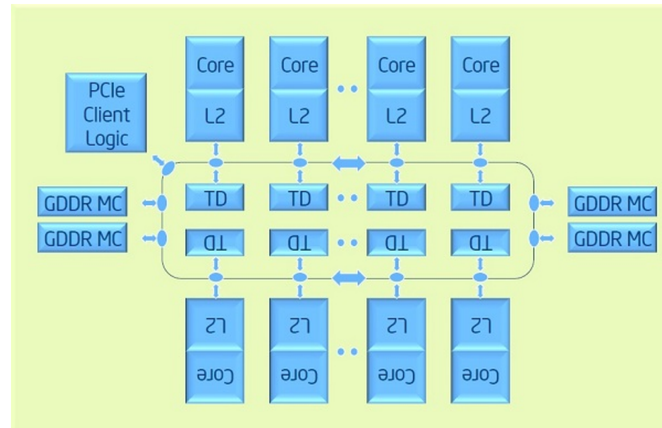

Figure 2.5: Intel MIC design

2.2 Barriers to high performance computing

Research in high performance computing falls into two major domains: hardware design and software implementation. The hardware design mainly focuses on delivering high

computational power. It also involves studies in logistical hardware components, such as the memory hierarchy, interconnection network, and I/O system. The software implementations has to exploit the available hardware resources and perform computation in an optimal way. The spectrum of software implementations covers a wide range of research interests. These include numerical algorithm design, performance tuning and optimization, exploiting parallelism, and program rewriting using new languages and programming models.

Innovations in computer architectural design have successfully delivered petascale performance in the past decade. The demand for upcoming exascale computing drives CPU vendors to develop computer architectures with an even higher peak performance. Besides the computational power, the demand for high power efficiency is added to the design roadmap for the future computer architectures. In each new generation of architecture design, hardware delivers higher theoretical peak performance to fulfill the demand for high computational power. Software applications can gain performance improvements from the higher number of processing units and the higher clock rate in the new hardware design. However, the achievable percentage of peak performance did not follow that trend. To achieve high percentage of peak performance will require significant efforts in programming optimization. These optimizations have to exploit the hardware characteristics to perform computation with a high efficiency. The followings are factors in hardware design that post challenges in high performance computing.

1. Multi-/Many-core design:

   Due to the physical limitations, increasing CPU clock speed to achieve higher performance is no longer feasible in the CPU design. Instead, placing multiple processing units on a single die provides a feasible way to gain theoretical performance improvement. This also makes the CPU design continuously follow Moore's Law. Besides the increasing number of processing units, the requirement of high power efficiency leads to more specialized hardware designs. Processing units with in-order execution, reduced computational power, and specialized memory hierarchy have started to appear in the latest HPC systems to lower power

consumption. It has been known for years that parallel programming has to be adopted to exploit these multiple processing units on a single die. The majority of the programmers in the HPC community has crossed this first hurdle and overcame the challenge. However, with a massive number of cores placed on a single die, we can foresee the challenge to feed data to many processing units with the limited memory bandwidth. The cache memory hierarchy, along with the cache coherency design to accommodate the multi-/many-core processors, adds another level of programming complexity. Extra burdens are added to programmers as a result. Achieving high computational performance on the latest CPUs requires significant efforts from programmers.

2. Multi-level cache hierarchy:

   Accessing off-chip memory takes a significant amount of CPU cycles and a large energy consumption. The cache memory provides a temporary space with fast access to allow the computing units to process the data stored in it. With more data stored and accommodated in the on-chip memory space, higher performance and energy efficiency can be expected. The latest cache memory designs include the multi-level memory hierarchies to provide much larger storage space with slightly higher access latency. This design provides a potential to relieve the pressure in the precious first-level cache memory. However, the cache associativity, cache replacement and writing policies, and the implicit cache memory management all require additional efforts from programmers to achieve efficient cache memory utilizations. Software applications with optimal temporal and spatial localities are expected to have efficient cache memory usage. Efficient cache utilization has become one of the key challenges in high-performance computing.

3. Memory bandwidth:

   The growing disparity in speed between CPU and off-chip memory leads to the well-known memory wall. The traditional vector machine had good balance between computational performance and available memory bandwidth. The ratio between peak floating-point performance and sustainable memory bandwidth was

1 Flop/word. The latest CPUs have ratios between 10 Flop/word and 200 Flop/-word. These ratios from a subset of the latest CPUs are shown in Fig. 2.6. Systems delivering low sustained memory bandwidth, with high ratios, tend to be memory bandwidth-limited. Modern CPU designs add cache memory to overcome the imbalance between computational power and available memory bandwidth. The cache memory makes it possible to have traditional vector-style computation in modern CPUs. However, applications with low cached data reuse or long data reuse distance gain limited benefit from the cache memory. Memory bandwidth therefore is still a significant factor in high performance computing.

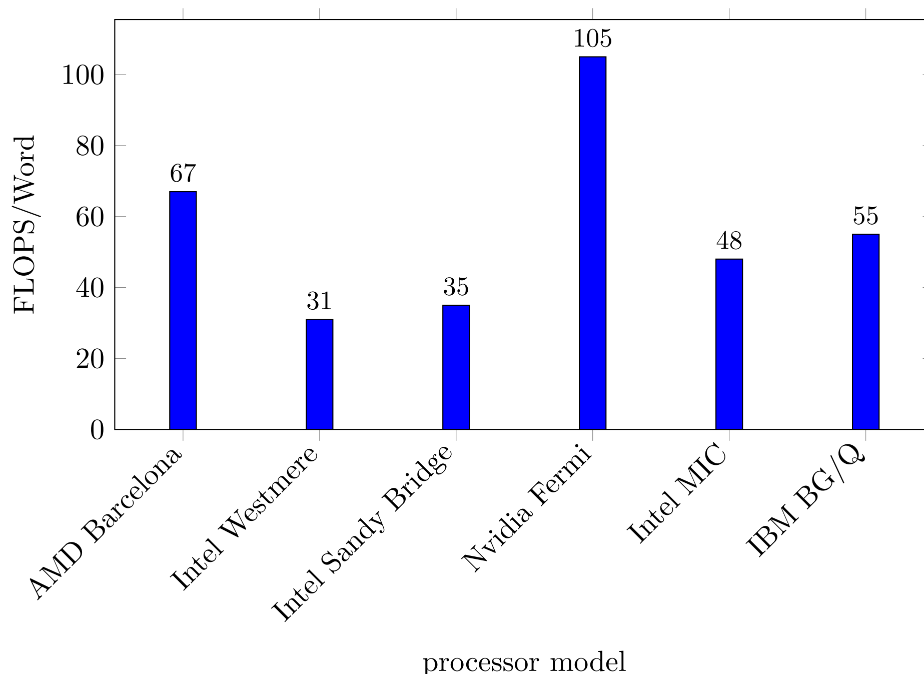

Figure 2.6: Ratio between peak floating-point performance and sustainable memory bandwidth. [1]

These factors are commonly seen in the latest HPC systems. The most extreme examples are found in the systems exploiting GPUs or Intels upcoming MIC architecture. With over 50 cores integrated in the add-on card, which is attached to its host system

through the PCI-E channel, the memory system on the card provides memory bandwidth that is improved by only a modest factor compared to that of the host memory. Feeding data into a huge group of processing cores easily saturates the limited memory bandwidth. The simultaneous multithreading supported in certain CPUs exacerbates the memory starvation problem by sharing the precious on-chip resources. The intent for these new hardware designs is to achieve high performance. However, software applications can easily fail that expectation unless they can exploit these hardware features efficiently. A significant amount of optimization and performance tuning are necessary to tackle these performance challenges. We are foreseeing the need for the software implementation to share the responsibility in delivering high performance computation.

Chapter 3

Computational Fluid Dynamics

Science and engineering research in fluid dynamics involves both theory and experiments. The development of accurate numerical algorithms and the revolution in computer technology have introduced a new approach in fluid dynamics research. Computational fluid dynamics (CFD) was introduced in the twentieth century as a "third approach" in the philosophical study and developments of fluid dynamics. It provides a synergistic approach to complement the other two approaches of pure theory and pure experiment. But it will never replace the need for theory and experiment. Computational fluid dynamics assists in the interpretation and understanding of the theoretical and experimental results. Therefore, research in fluid dynamics relies on a proper balance among these three fundamental approaches.

CFD involves the simulation of fluid flow to obtain basic scientific understanding (such as the understanding of stellar evolution), to predict fluid behavior (as in weather forecasting), or to design mechanical devices (such as airplanes). Almost all CFD calculations are carried out on grids that span a physical domain, usually in 3 spatial dimensions. Although these grids can be irregular, at a local level they are usually quite regular, at least in their logical structure. This feature of CFD simulations produces a good fit to computation with the SIMD engines that lie at the heart of all modern computing devices.

In this dissertation, the experimental study chooses a subset of CFD applications.

These applications represent various implementations and simulations in several scientific domains. Research presented in this dissertation is applied to more CFD applications with much larger scale computation. The following sections briefly introduce the CFD applications related to this dissertation.

3.1 CFD applications

3.1.1 Rayleigh-Taylor Instability

The first one is Raleigh-Taylor Instability, or RT instability (after Lord Rayleigh and G. I. Taylor). It is an instability of an interface between two fluids of different densities that occurs when one of the fluids is accelerated into the other.[50]. Our research team implemented the RT insatiability simulation using a hydrodynamics code based on the piecewise-parabolic method (PPM). This research performed large-scale simulations using the the IBM Roadrunner machine at Los Alamos National Laboratory [51]. Fig. 3.1 presents the visualization output generated from the simulation on the Roadrunner machine.

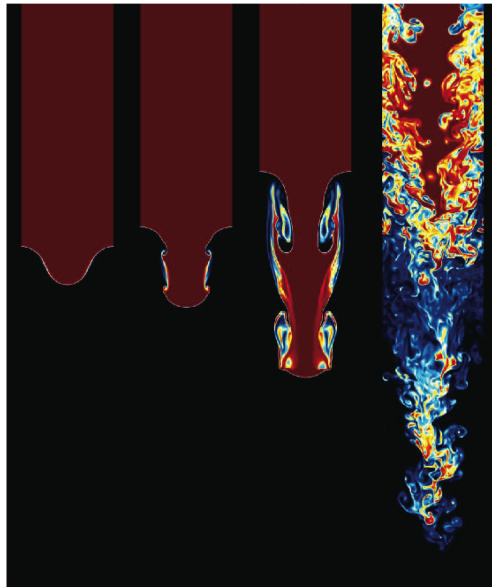

Figure 3.1: Visualization from RT instability simulation

3.1.2 PPM advection

PPM advection computes the motion of a distribution of a concentration (a scalar variable that is conserved along fluid streamlines) through a uniform Cartesian mesh according to a prescribed velocity field that varies in space but not in time. The velocity value at any point is computed from a simple formula. This application involves a portion of our CFD codes that performs interpolation on a grid, and from that it computes flux values and applies a simple conservation law. The interpolation algorithm is complex (details described in [52]), and it involves a wide difference stencil. In this respect, this algorithm is similar to many others that are in general use in the CFD community. The wide difference stencil comes from the algorithm's high-order accuracy as well as from constraints that are applied to the interpolation polynomials to enforce monotonicity of the interpolation polynomial when appropriate (see [52]). We compute the velocity field from a formula rather than reading it from memory in order to make this small code module end up with about the same overall computational intensity as our full multi-fluid hydrodynamics application. The PPM advection is representative of the CFD applications implemented within our research team. The simulation is widely implemented and studied on multiple computing devices. Fig. 3.2 presents the visualization output generated from the advection simulation.

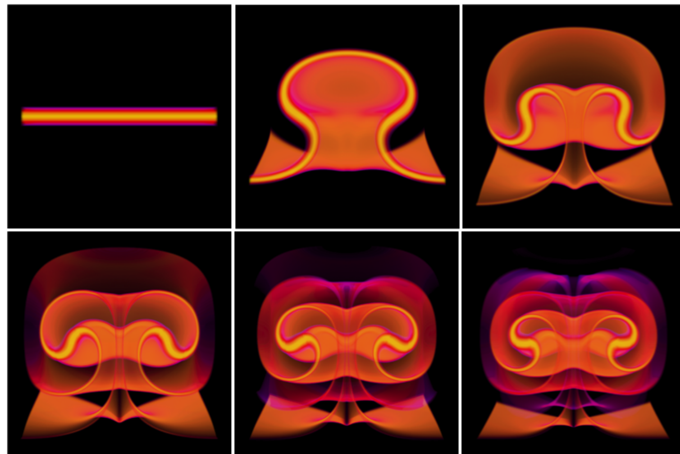

Figure 3.2: Visualization from advection simulation

3.1.3 Stellar convection

A PPM gas dynamics code is used to simulate the helium shell flash convection in the deep interior of an asymptotic giant branch (AGB) star [53, 54]. This simulation code includes the full PPM gas dynamics scheme as set out in [52] and the PPB multi-fluid fractional volume advection scheme described in [55, 56, 51]. This is a fully functional 3-D code that has been run on over 98,000 CPU cores on the Kraken machine in National Institute for Computational Sciences. A visualization of the helium shell flash is shown in Fig. 3.3.

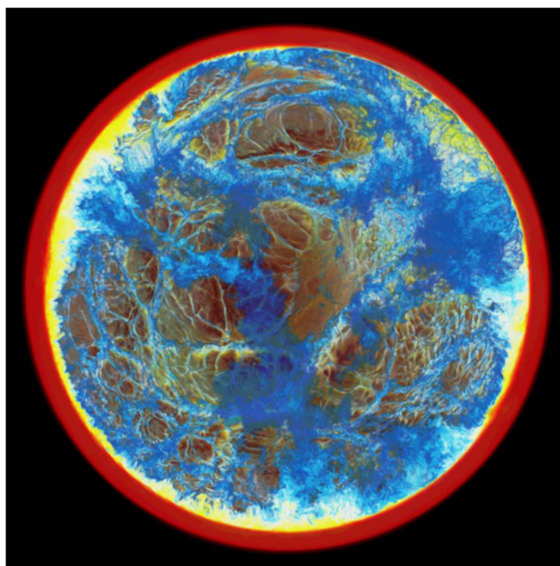

Figure 3.3: Visualization from stellar convection simulation

3.1.4 PPM single-fluid

Single-fluid, low Mach number PPM gas dynamics is used to simulate in great detail for homogeneous, compressible turbulence as well as convection flows in stars (see article and references in [57]), and it scales extremely well up to 98,312 CPU cores. Our research in recent years has been concentrated on more difficult multi-fluid problems, which build PPB advection of a multi-fluid volume fraction together with single-fluid PPM [56, 51]. Our most recent work adds in the necessary algorithmic features that handle very strong

shocks, and this complicates the message passing considerably, although at little cost in overall code performance. The version of PPM gas dynamics used in this dissertation is similar to, but not the same as the sPPM benchmark code [58] that was used several years ago to specify performance requirements for large DOE computer acquisitions and that won the Gordon Bell award in the performance category in 1999 [59].

3.1.5 Two-fluid PPM gas dynamics with strong shocks

The single-variable advection code discussed above brings out many but not all of the important implementation issues and performance considerations of processors. Within the context of explicit gas dynamics applications, we feel that the PPM code with a PPB moment-conserving treatment of multi-fluid volume fraction advection shares many features with the algorithms found in full application codes within the CFD community. These PPM and PPB algorithms (see [52, 56, 51]) involve a great deal of computation and also logic. PPM performs more computation per datum than PPB, but both are highly computationally intensive. The PPM ([52]) explicitly reflects the tight coupling of the 5 conservation laws of gas dynamics by working with Riemann invariants and Riemann solvers. This coupling forces the algorithm to perform computations that cannot be decomposed into tiny kernels characterized by many flops performed on a few inputs with a few outputs. This tight coupling of multiple variables is a fundamental fact about fluid dynamics that is reflected in the algorithm. This nature of that application reflects the character of the entire multi-fluid PPM+PPB grid cell update calculation, where we read in 35 numbers per grid cell and write back 15 numbers. The computation thus demands that if enough data workspace is not available on the same silicon chip as the processor, a significant amount of bandwidth to and from a more distant memory must be provided.

3.1.6 Inertial confinement fusion (ICF)

A multi-fluid PPM code is set up to solve an inertial confinement fusion (ICF) test problem described in [35] that is derived from that of Youngs [36]. The simulation

studies a process where nuclear fusion reactions are initiated by heating and compressing a fuel target. This fully functional CFD code has scaled on the Blue Waters machine at National Center for Supercomputing Applications to over 730,000 threads running on over 24,000 nodes. This same simulation achieves a sustained 1.5 Pflop/s performance. Fig. 3.4 shows the visualization result from this simulation.

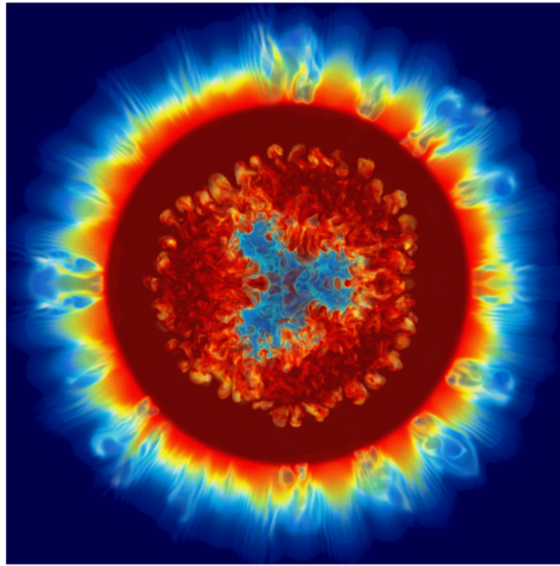

Figure 3.4: Visualization from ICF simulation

Chapter 4

Optimization for SIMD Engine

Single instruction multiple data instructions were used for computation in the vector supercomputer such as CDC-star100 [60] in the 70's. Each instruction performs computation on a vector of data, or a packed superword. Modern microprocessors incorporate multimedia extensions to achieve higher performance on multimedia workloads. Those extensions from different vendors can be characterized as SIMD units that execute on packed fixed-length short vectors. The popular SIMD examples are Streaming SIMD Extensions (SSE) for Intel X86 processors, Advanced Vector Extensions (AVX) for Intel X86 processors, Vector Multimedia eXtension (VMX/AltiVec) for IBM Power processors, and the SPU extensions for the IBM Cell processor. The early generations of SIMD instruction set support only integer computation for multimedia applications. With the capability to exploit parallelism by simultaneously executing an instruction on multiple data, the SIMD engine is commonly adopted in general-propose microprocessors. SIMD implementations are also adopted in general computations and are not only restricted to multimedia anymore.

SIMD achieves significant performance improvement in high performance computing and has become a crucial element in microprocessor design. Processor design starts to incorporate greater SIMD width for higher data-level parallelism. The latest Intel multi-core Sandy Bridge CPU has a 256-bit SIMD unit for an 8-way single precision SIMD. The Intel MIC architecture uses a 512-bit SIMD engine and can perform 16 single precision

floating-point operations simultaneously. The popularity of general-purpose graphics processing units (GPGPU) also leads to a broader use of SIMD implementations. Fig. 4.1 illustrates the SIMD widths for different SIMD instruction sets.

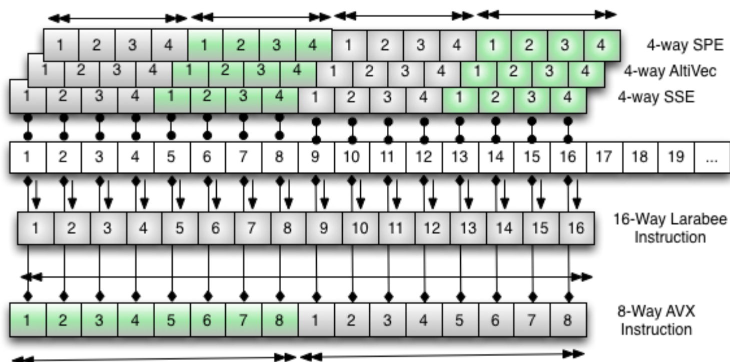

Figure 4.1: Illustration of SIMD width

There are three main approaches to generate SIMD instructions: writing SIMD instructions in assembly code, using SIMD intrinsic functions provided by vendor's compiler, and relying on vectorizing compiler to perform vectorization. Most programmers rely on compiler optimization to perform vectorization for the SIMD engine. One study has shown that three state-of-the-art compilers can vectorize only 45-71% of the loops in the synthetic benchmarks [25]. Additional manual analyses and transformations are required to augment the automatic vectorizing capability. Programming in assembly language has the full control of hardware features. The difficulty in assembly programming and required knowledge in low level details significantly reduce the programming productivity. SIMD intrinsic functions provide one-to-one mappings to the low level hardware instructions. Vectorization with intrinsic functions enables programmers to execute SIMD instructions from high-level languages. With the SIMD intrinsic functions, programmers can bypass some compiler optimizations and analyses to assure SIMD engine execution. This approach also eliminates the complexity in low-level details such as register management and memory address computation.

Implementations of vectorization in current compiler packages have differences in

their analytical strategies. Therefore, pursuing vectorization through compiler optimization is limited by the vectorizing capability of the compiler. In this chapter, a source-to-source approach using the SIMD intrinsic functions is used. This transformation is implemented to perform vectorization for CFD applications studied in this dissertation. The transformation is implemented in a pre-compilation tool and can support the latest SIMD instruction sets, including SSE, AVX, Altivec (IBM), and the most recent 512-bit wide SIMD instructions for Intel MIC. We discuss the challenges in current compiler vectorization and present the details of our vectorization implementation in the following sections.

4.1 Challenges in vectorization

Vectorization exploits data-level parallelism for high performance computation. In order to vectorize, a loop has to be the innermost loop in a single basic block with no jumps and branches. The loop must have countable iteration space with unit-stride memory accesses. No backward loop-carried dependences, special functions, and subroutine calls are allowed in a vectorizable loop. Programmers need to provide very rigid loop structures for compilers to perform vectorization. Vectorizing compilers can fail in vectorization for several reasons and require additional transformations and optimizations to overcome these vectorization obstacles. The following section discusses the vectorization challenges and some solutions for them.

Non-unit stride memory access: Vector instructions process data stored in vector registers or in consecutive memory location. Data stored in discrete locations requires extra operations for vectorization and leads to low computational efficiency. One well-known example is the array of structures and is commonly seen in object-oriented programs (example shown in Code 4.1). Vectorizing compilers could vectorize this structure but would require instructions to gather data from discrete memory locations to a vector register. This leads to inefficient vectorization and overhead cost in computation. To allow efficient loads and stores for vectorization, the program has to be transformed to

use structures of arrays (example shown in Code 4.2.) Compilers have limited capability to perform this transformation when the data structure is shared across procedures. Performing such transformations manually involves a significant amount of work in programming. Code 4.3 shows a second example with non-unit stride accesses. Loop interchange can be applied to provide the optimal vectorizable loop structure, as shown in Code 4.4. Outer-loop vectorization [61] serves as an alternative solution for this case, but the implementation is not generally available in compilers.

Code 4.1: Array of structures

```
typedef struct
{
  float rho;
  float p;
} data;
data AOS[nsize];
```

Code 4.2: Structure of arrays

```
typedef struct
{
  float rho[nsize];
  float p[nsize];
} SOA;
```

Code 4.3: Longer stride access

```
do i = 1, 16
do j = 1, 16
 a(i,j) = b(i,j) + c(i,j)
enddo
enddo
```

Code 4.4: unit-stride access

```
do j = 1, 16
do i = 1, 16
 a(i,j) = b(i,j) + c(i,j)
enddo
enddo
```

Unaligned array or operand: The restriction to aligned arrays or operands is due to the vector load and store operations. Code 4.5 and code 4.6 show the difference between aligned array accesses and unaligned array accesses. Transformations such as loop peeling and array padding are adopted to circumvent this limitation. The latest microprocessors support unaligned load and store instructions but with a performance penalty. Several compiler techniques were developed to enhance the vectorization ability through memory reorganization[21, 22]. However, the availability of these optimizations depend on the vectorizing compilers.

Code 4.5: Aligned memory access

```
do i = 1, 16
  a(i) = b(i) + c(i)
enddo
```

Code 4.6: Non-aligned memory access

```
do i = 1, 16
  a(i) = b(i+2) + c(i+3)
enddo
```

Data dependence: Vectorization performs data-level parallelism and has a similar data dependence limitation as in loop-level parallelism. All lexicographical backward dependences have to be eliminated for optimal vectorization. Research in compiler optimizations has contributed a significant amount of effort in resolving this issue. However, the capabilities to eliminate data-dependences vary in vectorizing compilers. Vectorizing compilers also support directives for the user to advise that no loop-carried dependences exist in the loop structure (e.g. ivdep pragma in GCC and Intel compiler). This type of compiler advisory is generally considered in performing vectorization.

Control flow: Control flow in a loop structure prohibits vectorization in many vectorizing compilers. The conditional vector merge functions, known as CVMGM functions in Cray vector machines, are written using if statement in current Fortran and C languages. These vectorizable functions are treated as control flow and cause vectorization failures. Vectorizing compilers need to implement special analysis and code generation phases to vectorize code with conditional vector merge functions. Some compilers are capable to vectorize the conditional vector merge functions if it is implemented with conditional operator (example shown in Code 4.7). However, the conditional operator is only supported in languages like C and C++.

Code 4.7: Conditional vector merge function using conditional operator

```
e = (a > b) ? c : d;
```

Others: Vectorizing compilers follow an analytical approach to evaluate the profitability of vectorization. The vectorization decision depends on the loop structure and statements contained inside the loop. Complex mathematical operations, such as division and trigonometric functions, might require much larger loop iterations to cover the long computational pipeline. It is commonly seen that these mathematical operations

hamper compiler vectorization. Loop fission can be applied to split a loop into pieces and hoist the complex mathematical functions from the original loop structure. The transformed code with multiple smaller loops can be vectorized but contains suboptimal loop structures because of worse memory locality and lower data reuse.

With decades of research effort, state-of-the-art vectorizing compilers still have limitations performing vectorization. The reasons could be the engineering difficulties in implementation, the lack of required analytical methodologies, and the conservative optimization strategy. Vectorization is one of the many factors within compiler optimizations. Rigid vector loop structures can be broken by the phase-ordering problem in compiler optimization [62]. It is a known challenge for analytical compilers to balance vectorization with other optimizations. To prioritize the importance of vectorization in compiler optimization, SIMD intrinsic functions allow programmers to generate vector instructions from high-level languages. In this dissertation, a vectorization framework is introduced to ease the programming efforts using SIMD intrinsic functions. The details in implementation is presented in Sec. 4.2.

4.2 Implementation of vectorization framework

The vectorization framework is implemented on top of the source-to-source transformation tools (discussed in Chapter 7). Fig. 4.2 illustrates the high-level design of this vectorization framework. As we walk through the following steps, the transformation details will be presented.

1. **Loop normalization:** A transformation is first applied to normalize the loop structure written in C/C++ languages. Normalized loops have unit increment and comply with regulations to have the loop variable starts at 0. Loop normalization simplifies the analysis process to verify the vectorization legality.

2. **Identifying vectorizable loop:** The key to vectorization is to identify the vectorizable loops. Compilers use analytical approaches and performance models to evaluate the profitability of vectorization. Loops could be transformed or divided

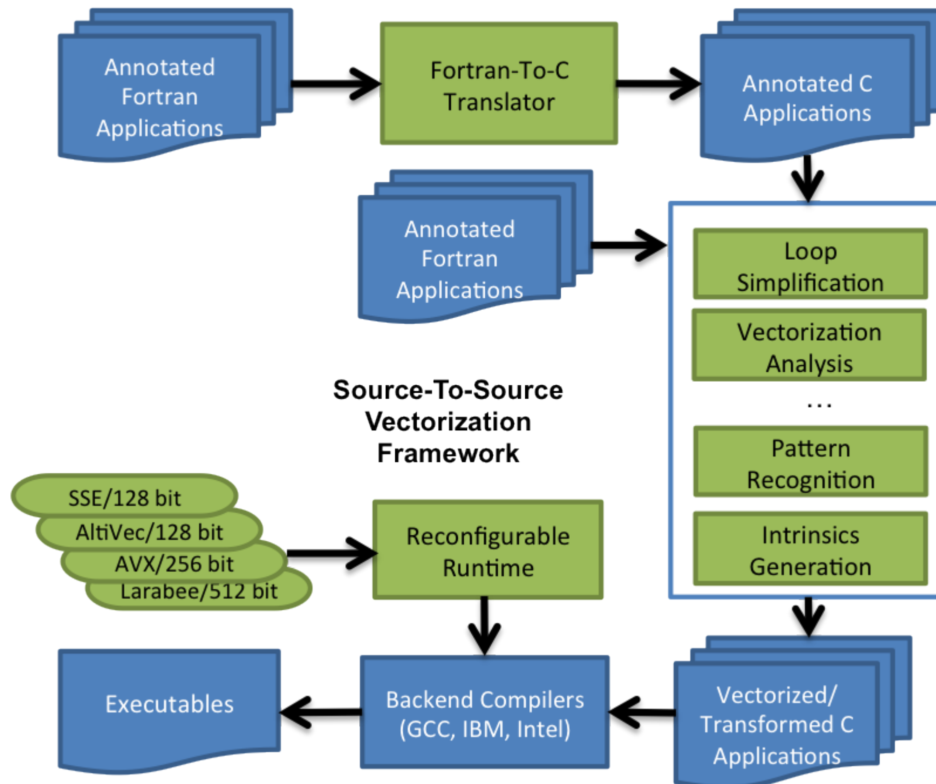

Figure 4.2: Overview of vectorization framework

into pieces to generate partial vectorized loops. But we have found that this might not deliver the expected performance due to the issues discussed in Sec. 4.1. Our vectorization framework will focus on scientific fluid dynamics applications and help application developers to boost the computational performance. The implementation adopts a different approach, namely that the framework will grant highest priority to user directives to identify the vectorizable structure. The tool takes the "*!DEC$ VECTOR ALWAYS*" directive(shared with the Intel Fortran compiler) for the Fortran code and "*#pragma SIMD*" directive for the C/C++ code . The tool will perform transformation to the loops with this directive. A later implementation (built on top of ROSE compiler) adopts more automatic approaches to identify vecotizable loops. The compiler scans all the innermost loops and identifies them as vectorization candidates. An analysis is applied to

the vectorization candidates to verify the legality of vectorization. All vectorizable loops are selected and submitted to the later transformations.

3. **Vector operand declaration:** Once all vectorizable loops are identified, the vectorization framework needs to declare all the variables and arrays inside the vector loops to be the packed SIMD data type. In the code example in Code 4.8, *uxavl* and *sintwopicyl* are the floating-point arrays that will be declared as packed SIMD data types. The floating-point array *sintwopixl* will be declared as a scalar array, because it is only referenced as a scalar in the vector loop. Integer variables *j* and *i16m00* serve as indices and will be declared as scalar integers. The parameter *nssq* will be replaced by its actual value in the output of vectorization. All constant values in vector loops will be translated and promoted to packed constant variables. Intel SSE uses *_mm_set1_ps* to assign a constant value to all elements in packed data. The packed floating-point data type for Intel SSE is *__m128*, and IBM AltiVec uses *vector float*. The packed data type for 4-way SIMD packs 4 words inside it and usually is called a quadword. The *uxavl* array has to declare its array size to be a quarter of the original array size in the quadword type, but the overall memory space for *uxavl* will be the same. After this step, the tool generates the declaration section shown in Code 4.9.

4. **Memory alignment:** Memory has to be aligned for SIMD execution. The vectorization framework will generate declaration specifiers to force the data to be aligned on required boundaries(e.g. 16-byte for SSE). The *__declspec(align(16))* in Code 4.9 is the declaration specifier for Intel SSE. To generate a vector loop for the loop in Code 4.6, a compiler or translation tool must make a special effort. The compiler team from IBM developed shifting policies to shift all operands to aligned addresses for SIMD execution [22]. In our vectorization framework, we create a "packed register" that will store 4(or more for a wider SIMD width) contiguous words and use it as an operand for SIMD execution. For the Intel SSE extension, our translator will simply use *_mm_loadu_ps* and *_mm_storeu_ps* that will perform an unaligned load and store. In the example in Code 4.6, b[i+2] ∼

Code 4.8: Input example for SIMD declaration

```
      parameter (nssq=nyy*nzz)
      dimension uxavl(nssq,nx+nghostcells)
      dimension sintwopixl(nx+nghostcells)
      dimension sintwopicyl(nssq)
      dimension cylm(nssq), ym(nssq)

!DEC$ VECTOR ALWAYS
      do j = 1,nssq
      cylm(j) = ym(j)
      sintwopicyl(j) = sin(twopi*cylm(j))
      enddo
      sintwopixl(i16m00) = sin (twopi * xl(i16m00))
!DEC$ VECTOR ALWAYS
      do j = 1,nssq
      uxavl(j,i16m00) = sintwopixl(i16m00) * sintwopicyl(j)
      enddo
```

Code 4.9: Output example for SIMD declaration

```
__declspec(align(16)) __m128 uxavl_vec[(128)/4];
__declspec(align(16)) __m128 sintwopicyl_vec[(16)/4];
float *uxavl;
float *sintwopicyl;
float sintwopixl[8];
int j, i16m00;
uxavl = (float *) uxavl_vec;
sintwopicyl = (float *) sintwopicyl_vec;
```

b[i+5] will be stored in one virtual packed register; and c[i+3] ∼ c[i+6] will be stored in another virtual packed register for the first SIMD loop iteration. However, unaligned expressions of the sort in Fig. 4.6 rarely appear in code that has been transformed by our other optimizations (discussed in Chapter 5). The extra effort needed to take care of memory alignment in these rare cases has a negative impact on performance. Therefore, we honor aligned expressions and assure vectorization for them. The vectorization framework provides a feature to handle unaligned cases, but recommends that they be used very rarely. Arrays b and c in the example might reference out of their bounds at the very last vector loop iteration, causing extra instructions to drag performance down. Programmers are asked to re-verify their vector loop structures to avoid this.

5. **Loop transformation:** The source-to-source transformation framework generates the abstract syntax(AST) tree from the input source code. Prior to the code generation, the tool can perform transformations on the AST. Strip-mining is the loop transformation applied in vectorization. Fig. 4.3 shows one example of loop strip-mining.

```
float a[16], d[16];
for(int i=0; i < 16; ++i)
        d[i] = a[i];
```

```
float a[16], d[16];
for(int i=0;i<16;i+=4){
for(int j=i;j<min(i+4,16);++j){
        d[j] = a[j];
}}
```

Figure 4.3: Example for Strip-mining

6. **Code generation:** The next step is to translate the loop body into SIMD expressions. The transformation simply replaces the original numerical operators with the intrinsic function calls. The tool also recognizes fused multiply-add/sub expressions and grants a higher precedence for them. One example is shown in

Fig. 4.4. The conditional vector merge functions, (CVMGMs), become if statements inside vector loops. The vectorizer will convert it into an intrinsic function for comparison followed by another intrinsic function for selection. Fig. 4.5 shows an example of the conditional vector merge function in Intel SSE format.

```
float a[16], d[16];
for(int i=0; i < 16; ++i)
        a[i] = sinf(d[i]);
        d[i] = a[i] * a[i] + d[i];
```

```
float a[16], d[16];
__SIMD *a_SIMD, *d_SIMD;
a_SIMD = ((__SIMD *)a);
d_SIMD = ((__SIMD *)d);
for(int i=0,j=i;i<16;i+=4,j++){
  a_SIMD[j]=_SIMD_sin_ps(d_SIMD[j]);
  d_SIMD[j]=_SIMD_mad_psd(a_SIMD[j],a_SIMD[j],d_SIMD[j]);
}
```

Figure 4.4: Example for code generation

```
      if (a(j) .lt. b(j)) b(j) = a(j)
```

```
#define _mm_sel_ps(a,b,c) _mm_or_ps(_mm_andnot_ps(c,a),_mm_and_ps(c,b))

__declspec(align(16)) __m128 tpatt;

tpatt=_mm_cmplt_ps(a_vec[(j-(1))],b_vec[(j-(1))]);
b_vec[(j-(1))] = _mm_sel_ps(b_vec[(j-(1))],a_vec[(j-(1))],tpatt);
```

Figure 4.5: Code generation for conditional vector merge function

7. **Vector/scalar conversion:** Because part of the code has to be retained in the non-vector format, packed data promotion or scalar extraction are necessary in code generation. However, packed data requires an extra index to identify the element inside it to be manipulated. This will increase the complexity in the translation and might influence the application performance. Our solution for

this is to create extra scalar pointers. The vectorizer will declare a scalar pointer pointing to each packed data variable. Whenever the packed data is used outside the vector loop, this scalar pointer will point to the corresponding memory space. The *uxavl* code 4.9 is the pointer that points to the quadword array *uxavl_vec*.

8. **Scalar variable in vector loop:** The vectorization framework requires liveness analysis to perform code generation for the scalar variables. If a scalar variable is live-in of a vector loop, the code generation promotes that scalar variable to a SIMD operand before entering the vector loop. The SIMD operand is used for computation within the vector loop. If a scalar variable is live-out of a vector loop, the code generation extracts the scalar value from SIMD operand after exiting the vector loop. When a scalar variable is computed and consumed in a reduction loop (also known as a summation loop), it requires different approach or intrinsic functions(e.g. *_mm512_reduce_add_ps* for Intel MIC architecture) to perform the vectorization.

9. **Auxiliary translation:** In this vectorization framework, we design a new directive called *$PPM! DMA*. In the application for the Cell processor, data has to be copied to on-chip memory through explicit DMA commands. We designed this directive to generate DMA commands in the translation. Through the explicit DMA commands, asynchronous data transfers can be overlapped with computation. However, for general-purpose processors, the DMA command becomes a regular data copy between memory and the subroutine stack. The DMA directive is retained in the vectorization framework, because the Cell processor is one of our target architectures. Another reason is the DMA directive identifies the candidate for data prefetching. Specialized intrinsic functions, e.g. *_mm_prefetch()* used in X86 instruction set, can be generated to perform user-directed software prefetching. There are several special code generations for different purposes. Some processors support specialized load and store instructions to load or store data into SIMD registers. For example, *VEC_LD()* function loads memory data from the given address and performs a data type conversion on the IBM Blue

Gene/Q processor. The vectorization framework allows developers to create new directives to identify these special occasions.

4.3 Performance result

Vectorization is a common optimization adopted in many commercial and research compilers. The importance of this optimization draws more attention from researchers in both HPC and general computation domains. The vectorization framework presented in this dissertation is motivated by research experiences in our research team. The goal is to prioritize vectorization over other optimizations and also provide an alternative approach to pursue vectorization in scientific codes. The research and engineering effort in this vectorization framework will also contribute to general software applications. Therefore, this section presents a brief performance study using our vectorization framework.

The first experiment uses Intel X5690 CPU (3.47 GHz) as the hardware platform. Since GCC is the most popular open-source compiler and is pervasively installed in most HPC systems. Therefore GCC (in version 4.4.1) is chosen as the compiler for this study. The vectorization framework is built in ROSE compiler. Details of the ROSE compiler infrastructure is presented in Chapter 7. The benchmark chosen in this performance study is the Test Suite for Vectorizing Compilers (TSVC) developed by Callahan, Dongarra and Levine[15]. We chose 32 loops from 135 loops in TSVC benchmark. These loops represent most commonly-seen loop structures in scientific codes. We also chose loops involving control flows (CVMGM function) inside the loop body to evaluate the capability for our vectorization framework.

Among the 32 chosen loops, 21 loops have lower execution time compared to the optimized results with highest optimization and vectorization from GCC. After inspecting the 11 loops with lower performance, we found that loop fission was performed in these loops. The fission divides loops into pieces and allows vectorization to be performed in the smaller loop regions. However, the fission leads to worse temporal data locality and results in a lower performance. Code 4.10 is one of the loops with lower performance

after vectorization. In the CFD applications used in this dissertation, we apply data reorganization to expose vectorization opportunity. Loops with similar structure in Code 4.10 could still be vectorized in the computation.

Code 4.10: Example code with lower performance achieved

```
for (int nl = 0; nl < 2*ntimes; nl++) {
        for (int i = 0; i < LEN-1; i++)
                a[i] = b[i] * c[i  ] * d[i];
        for (int i = 0; i < LEN-1; i++)
                b[i] = a[i] * a[i+1] * d[i];
}
```

The second expriment uses IBM Blue Gene/Q multi-core CPU (16 1.6 GHz cores) and IBM XL Fortran compiler in version 14.1 as the platform. The study uses two CFD applications, PPM advection and inertial confinement fusion code (details in Chapter 3), and compares the vectorization results using vendor's compiler and the vectorization framework presented in this dissertation. Both applications are optimized with the domain-specific optimizations (discussed in Chapter 5). Coarse-grained parallelization can be scaled up to 64 OpenMP threads in a single CPU with 4 threads running concurrently on a single core. The optimized codes have optimal vector loop structure with vector length of 16. Compiler options for vectorization and SIMD/vector directives are inserted as vectorization guidance in the compilation process. Optimized codes after aggressive compiler optimizations deliver best performances of 253.27 Mflop/s per core and 454.88 Mflop/s per core for PPM advection and ICF code respectively (shown in Fig. 4.6). Vectorization report generated from vendor's compiler showed only a small percentage of loops are considered vectorizable. The following reasons are given in the report:

- Vectorization is not profitable because of low loop counts. A loop with a number of iterations larger than 16 is considered profitable. Some loops in the codes contain only 4 iterations and are eliminated from the vectorization.

- Memory references with non-unit-stride loads/stores or non-vectorizable strides. All memory accesses in a vector loop have unit stride after the domain-specific optimizations. The optimizations also create a group of barrel-shift indices for array subscripts in most multi-dimensional arrays. This effectively reduces the memory footprint but complicates the compiler analysis.

- Loops contain control-flows. The compiler treats all CVMGM functions, discussed earlier, as control flows in the vector loops and eliminates them from vectorization.

- Statements contain function calls, i.e. $abs()$, $sin()$, and $cos()$. Vendor's compiler lacks the mechanism to generate SIMD instructions for special functions.

- Loops contain unsupported loop structures. All situations listed above are considered unsupported loop structures. A single precision constant number inside a loop is also considered unsupported structure. Compiler lacks the capability to promote a single precision constant value into a double precision SIMD operand.

- Partitions exceed the partition size limit (Only seen in ICF code). ICF code has a higher complexity in computation and a larger code size. The temporary space required for inter procedure analysis (IPA) exceeds its allocated limit. Optimizations, including vectorization, will be dropped for this reason.

Vectorization framework presented in this dissertation generates SIMD intrinsic function calls to force SIMD computation. Both PPM advection and ICF code are considered vectorizable after the domain-specific optimizations. The vectorization framework can vectorize loops containing iteration number as low as 4. The barrel-shift indices are kept in the array subscripts and all array accesses are consider vectorizable. CVMGMs and special functions are mapped to SIMD intrinsic functions through the transformation. Scalar variables and constant numbers are type-converted and promoted to SIMD operands for vectorization. ICF code with a large code size can still be vectorized through this framework. However, the issue of limited partition size still occurs when we submit the vectorized code to vendor's compilers. Fig. 4.6 shows significant performance improvement compared to the vectorization result using vendor's

compilers. The best performance of 658.96 Mflop/s per core is achieved.

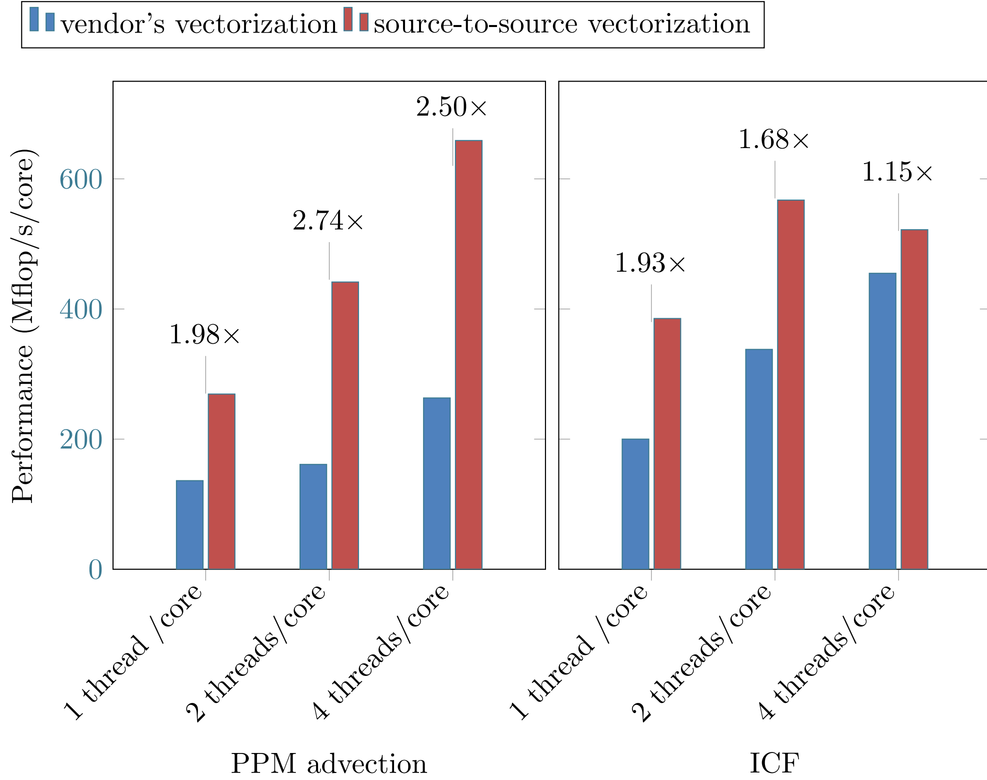

Figure 4.6: Vectorization comparison for IBM Blue Gene/Q processor

4.4 Contribution

Vectorization can deliver crucial performance improvement and is considered as an important optimization for HPC research. To perform vectorization on the HPC systems, users have to provide a rigid code structure and compilers have to vectorize the code accordingly. In this dissertation, a vectorization framework is developed to perform vectorization for the scientific applications. The main difference of this framework compared to other techniques implemented in vectorizing compiler is it generates SIMD intrinsic functions in high-level languages(C or Fortran) instead of low-level assembly instructions. We have found the following advantages of this framework in the research

for performance portability.

1. This framework relives programmers' burden in writing vector program using SIMD intrinsic functions.

2. The SIMD intrinsic functions provide one-to-one mapping to the low-level hardware vector instructions. Writing program with SIMD intrinsic functions guarantees to generate SIMD instructions in compiler's code generation process. When compiler fails vectorization in vectorizable code sections or loops, a significant performance degradation is observed. This framework provides an approach to compensate for the vectorization capability in vectorizing compilers. On the other hand, vectorizing compilers will perform vectorization when the support in SIMD intrinsic functions is insufficient.

3. Current SIMD intrinsic functions do not follow a unified standard and have variations among vendors' compilers. This vectorization framework provides a compiler-independent and hardware-independent vectorization platform. The support for multiple hardware platforms and compilers allows generation of vector code with SIMD intrinsic functions for different HPC systems.

4. This framework provides vectorization capability for experimental or early-access hardware platforms. When a newly designed hardware becomes available, the software stack development(e.g. compiler) might not be in a mature stage for full optimization capability. Several hand-tunings and manual optimizations are required to pursue high performance on the experimental platform. We have applied this framework to perform vectorization in several research projects using the early-access hardware platforms, including the IBM Roadrunner system and the Intel Many Integrated Core Architecture.

5. This framework successfully delivers performance improvements for several scientific applications and analytical benchmarks. Performing vectorization for performance portability can be achieved through this source-to-source vectorization framework.

Chapter 5

Optimization for On-Chip Memory

The increasing gap between the processor speed and the memory (DRAM) speed causes the well-known ”memory wall” [63]. Fetching data from remote memory is expected to have long memory latency. To reduce the performance impact caused by latency, the processor design adopts caching to include faster cache memory with shorter latency close to the processing unit. These fast on-chip memory units serve as a buffer to relay data from remote memory locations to the processing unit. Caching can effectively benefit programs with good temporal and spacial localities and reduce remote memory traffics with long latencies. To reduce the number of cache misses, a multi-level cache memory hierarchy is designed to host a much larger data set on chip. Hardware implicitly manages all the data movement in the complex memory hierarchy. Other techniques were developed to increase the efficiency of on-chip memory and lower the average memory latency. More recent processor designs incorporate out-of-order execution and simultaneous multithreading (SMT) to overlap the long memory latency with computation. Prefetching, through hardware prefetchers and compiler optimizations, takes an heuristic approach to speculate the memory access patterns and pre-load the required data into on-chip memory prior to the data consumption. Such hardware designs provide potential for high performance computing. However, it also demands

programmers or compilers to generate optimal code that can fully exploit the hardware characteristics.

Compiler optimization takes different strategies to enhance temporal and spacial on-chip memory localities. There are several existing optimizations to achieve efficient usage of the on-chip memory space. Loop fusion is popularly adopted to shorten the memory reuse distance and enhance the temporal locality. Loop tiling and memory blocking execute a subset of data to enhance both temporal and spacial localities. Software prefetching strategies compensate the weakness of hardware prefetchers to prefetch data with irregular memory access patterns. There are other transformations to increase the reuse rate of cached data. Compilers have the strength of performing optimizations and transformations automatically. However, compilers might not perform the optimal optimizations due to a lack of domain knowledge. In this dissertation, a series of optimizations (performed in a fixed order) are identified to perform on-chip memory optimization for CFD applications with structured grids. The main purpose of these optimizations is to reduce the memory footprint, increase computational intensity, and enhance spacial and temporal localities. These optimizations are presented in the applied order in the following subsections.

5.1 Memory organization

CFD programmers have long learned to formulate their codes in terms of vector computation. This fits well with SIMD processing, except for two considerations: vector alignment and vector length. SIMD operands must be aligned in order to be processed. If the vectors in the program are not aligned, the compiler must generate extra instructions that produce this necessary alignment, and this comes at some cost. For long vectors, this cost creates only a longer pipeline delay but it can be amortized over the large number of words processed in the pipeline. For short vectors, the impact of vector misalignment on performance is much greater. Unfortunately, modern CPUs require short vectors for the best performance. For long vectors, the on-chip cache memories may be too small to hold many intermediate vectors needed for reuse of cached data.

Long vector computation therefore tends to be limited in performance by the relatively small bandwidth between the CPU and its main memory.

CFD codes are written in a variety of coding styles, most of which exploiting vectorization. In an application using a structured grid, the problem is decomposed into a group of smaller subdomains. A grid cell is the smallest unit in the structured grid and contains one to several physical-state variables. A grid structure in a three dimensional computation can be represented as a Cartesian grid, and all required data is represented by three-dimensional arrays. The computationally intensive portions of these codes consist of a large number of triply nested loops that run over the physical domain of the computation for a single MPI process. As an example, we give a Fortran loop nest taken from the CM1 community code[64] (shown in code 5.1).

Code 5.1: CFD code example

```
do k=1,nk
do j=1,nj
do i=i1,i2
if(rru(i,j,k).ge.0.)then
dum(i,j,k)=rru(i,j,k)*(2.*s(i-3,j,k)
&  -13.*s(i-2,j,k)+47.*s(i-1,j,k)
&  +27.*s(i,j,k)-3.*s(i+1,j,k) )*tem
else
dum(i,j,k)=rru(i,j,k)*(2.*s(i+2,j,k)
&  -13.*s(i+1,j,k)+47.*s(i,j,k)
&  +27.*s(i-1,j,k)-3.*s(i-2,j,k) )*tem
endif
enddo
enddo
enddo
```

Eleven flops per loop iteration are performed here, with two vector operands (*rru* and *s*) streamed from memory and one streamed back (*dum*), for a computational intensity

of 3.7 flops/word. The required memory bandwidth is unlikely to be available on any modern device to support this loop's computation at full speed, unless somehow all the data could fit into an on-chip cache.

To boost performance for programs written in such a style, a fundamental data element used in the proposed domain-specific optimizations is a new data structure called the briquette. A briquette is a small contiguous record of data corresponding to a cubical region of the problem domain. Several physical-state variables can be packed together and are stored one after another in a briquette. Fig. 5.1 provides the overview of the memory organization using briquettes. The physical domain of the computation for a single MPI process will be decomposed into multiple grid-pencils. Each grid-pencil is formed by a sequence of grid briquettes and all the briquettes are stored in adjacent memory space. Within a briquette, the memory layout can be further decomposed into grid planes and grid cells. The array for the physical domain in a MPI process is dimensioned as following:

```
dimension   DD(4,4,4,6,nbx,nby,nbz)
```

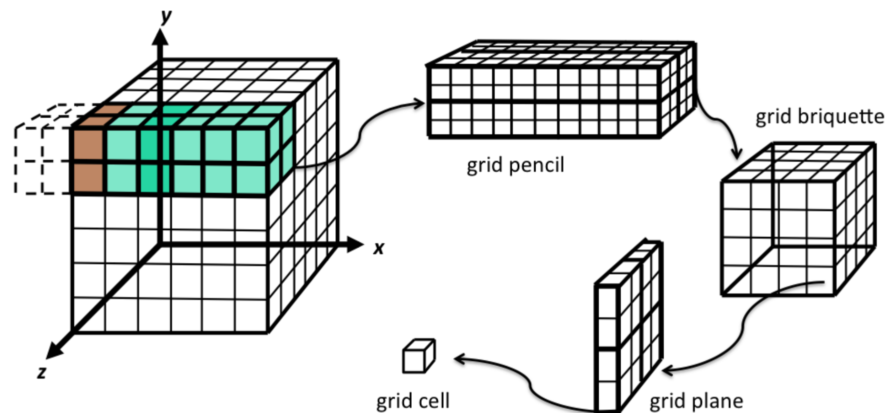

Figure 5.1: Illustration of data reorganization

The briquette structure provides the following benefits for CFD applications:

- Exploiting the SIMD engine: Memory layout in briquettes provides an optimal memory access pattern for SIMD execution. The size of a briquette is adjustable

for different hardware architectures. It is determined by the SIMD width and the cache memory capacity. We make the size of a briquette grid cell plane an integer multiple of the SIMD width. The data alignments for both the briquette grid cell plane and for each briquette are preserved by the carefully chosen size. Data packed in a briquette grid cell plane fulfills SIMD alignment and naturally serves as a SIMD operand for the computation. The spacial locality provided by the briquettes exploits SIMD engines with tremendous efficiency.

- Enhancing memory localities: Computation using briquettes implements a different format of cache blocking. Briquettes host blocked data stored in contiguous memory space. Accessing data in the blocked region requires a minimum number of cache line accesses. In a CFD application using a directionally split algorithm, we perform data transposition between two directional updates to force the data alignment for SIMD computation and to reduce the number of strided memory accesses. Transposition for a large physical domain requires tremendous strided accesses to main memory. With briquettes, the gather-scatter for transposition is executed within a briquette. This provides optimal temporal and spacial localities, as the briquette is cache resident during its transposition.

- Coalescing memory access: Processors perform memory loads and stores in the granularity of the cache line size. Briquettes pack contiguous records of data into multiple cache lines. All physical-state variables packed in the briquette record are stored in adjacent memory locations. Loading contiguous cache lines at a time decreases the translation lookaside buffer (TLB) pressure and avoids expensive TLB miss penalties. In addition, this minimizes accesses across new page boundaries and may benefit hardware prefetching.

5.2 Subroutine inlining

Large scale software applications are often divided into several smaller subroutines. Input to each subroutine comes from data stored in global memory space or data passed

through function parameters. Computation is performed within subroutines and results are returned before exiting. This approach allows the applications to be easily developed and reused. However, it increases the complexity in compiler optimization.

Inlining (or inline function expansion) replaces function calls with the function body. Inlining benefits several optimizations including register allocation, code scheduling, common subexpression elimination, constant propagation, and dead code elimination [65, 66]. However, aggressive inlining expands the code size and increases the instruction cache pressure. We perform inlining to eliminate function call overhead, reduce data reuse distance, and as a preparation for other transformations. Application developers insert directives to identify subroutines to be inlined and the source-to-source transformation tool executes the inlining process. Recursive function calls are rarely seen in the CFD applications used in this dissertation. Therefore, the transformation tool excludes them from the inlining process.

5.3 Loop transformations and computational pipelining

Transformations introduced in this section focus on cache usage optimization. The highlights for these optimizations are reducing data reuse distance, enhancing cache locality, and minimizing memory footprint. Before application of the transformations, a manual data structure reorganization packs data into briquettes. All relevant routines must be cognizant of this reorganization and inlined to form a single computation-intensive subroutine.

Given a 64^3 grid ($nx = 64$) and the number of ghost cells at each boundary equal to 4 ($nbdy = 4$), a domain of 72^3 grid cells is updated by a single thread running on a single CPU core. Each variable array used in a single-dimensional sweep needs data augmented by ghost cell values in only a single direction and thus consists of $64 \times 64 \times 72$ words, which is 1.125 MB. Data of this size cannot fit into the L1 and L2 caches. There will consequently be a tremendous amount of traffic to bring this data into the processor. A code snippet for this example is shown in Code 5.2. In a single dimensional sweep for the PPM example, the programmer splits a single sweep over a $64 \times 64 \times 72$ grid

into 16×16 sweeps over grid pencils of $4 \times 4 \times 72$ grid cells each, that is, 18 grid briquettes. Loop transformations and computational pipelining transform the code for such a briquette pencil update.

Code 5.2: Before memory optimization

```
      Dimension s(nssq,1-nbdy:nx+nbdy)
      Dimension unsmth(,1-nbdy:nx+nbdy)
      Dimension pl(ny,nz,1-nbdy:nx+nbdy)
      Dimension p(ny,nz,1-nbdy:nx+nbdy)
c     ...
      Do i=2-nbdy,nx+nbdy
      Do k=1,nz
!DEC$ VECTOR ALWAYS
      Do j=1,ny
        s(j,k,i)=1.
        if((p(j,k,i)-p(j,k,i-1)).lt.0.)
     &  s(j,k,i)=-1.
        difmon(j,k,i)=s(j,k,i)*difmon(j,k,i)
        if(difmon(j,k,i).gt.0.) then
        difmon(j,k,i)=
     &  max(s(j,k,i)*(pl(j,k,i)-p(j,k,i)),0.)
        endif
        pl(j,k,i)=pl(j,k,i)-
     &  unsmth(j,k,i)*s(j,k,i)*difmon(j,k,i)
      enddo
      enddo
      enddo
```

CFD applications can be constructed from many 3-D nested loops with the style similar to Code 5.2. The computation executes these loops in sequence and generates all required temporary results. When the temporary results are served as input data for

another 3-D nested loop, the long data reuse distance crossing 3-D loops and the computation tend to have cache utilization with low efficiency. The capacities of temporary arrays have to be big enough to host data for grids that span the whole physical domain. Loop tiling can be applied to enhance cache locality, but the tiled loops could involve large memory strides, which would increase TLB misses. For processors like IBM's Cell processors, main memory and on-chip memory have separate memory address spaces. Extra software control is required to extract tiled data from main memory and copy to the on-chip memory space.

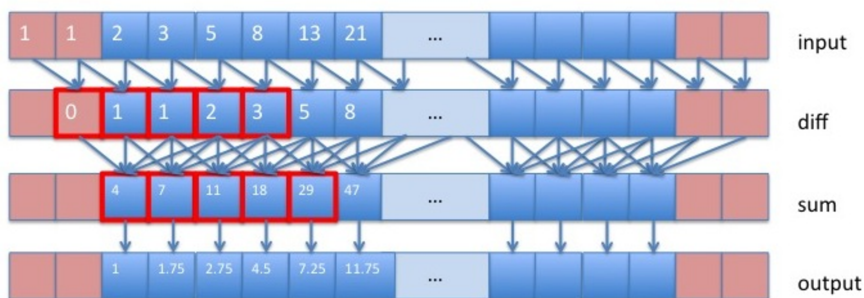

Figure 5.2: Illustration of 5-point stencil example

In a single dimensional update, the computation sweeps through the given arrays. Each grid cell's new value is computed by referencing neighboring grid cells in a fixed pattern. This highly regular pattern exposes the optimization opportunities. A simple code using 5-point stencil computation exemplifies our transformations and computational pipelining (shown in Fig. 5.2). Briquettes pack data to allow a group of concurrent single dimensional updates and exposes opportunities for SIMD engine usage. The following example excludes the briquette structure to simplify the explanation. The example computes the following formula:

$output_i = \{(x_{i-1} - x_{i-2}) + (x_i - x_{i-1}) + (x_{i+1} - x_i) + (x_{i+2} - x_{i+1})\} /4$

Without simplifying the equation, three loops are used to compute the differences, the sums, and the averages. The results are stored in array $diff$, sum, and $output$ respectively. Code 5.3 presents the example in its original form.

Step1: loops are right-justified to have the same upper loop bound (Code 5.4). This

Code 5.3: without transformations

```
      subroutine doSlow(input, output)
      real*4  input(1-nbdy:nsize+nbdy)
      real*4 output(1-nbdy:nsize+nbdy)
c
      real*4   diff(1-nbdy:nsize+nbdy)
      real*4   sum4(1-nbdy:nsize+nbdy)
c
      do i = 2-nbdy, nsize+nbdy
        diff(i) = input(i) - input(i-1)
      enddo
      do i = 1, nsize
        sum4(i) = diff(i-1) + diff(i) +
     &             diff(i+1) + diff(i+2)
      enddo
      do i = 1, nsize
        output(i) = sum4(i)/4
      enddo
      end
```

Code 5.4: loop bound adjustment

```
      subroutine doFast1(input, output)
      real*4  input(1-nbdy:nsize+nbdy)
      real*4 output(1-nbdy:nsize+nbdy)
c
      real*4   diff(1-nbdy:nsize+nbdy)
      real*4   sum4(1-nbdy:nsize+nbdy)
c
      do i = 2-nbdy, nsize+nbdy
        diff(i) = input(i) - input(i-1)
      enddo
      do i = 1+nbdy, nsize+nbdy
       sum4(i-nbdy) = diff(i-1-nbdy)
     &                + diff(i-nbdy)
     &                + diff(i+1-nbdy)
     &                + diff(i+2-nbdy)
      enddo
      do i = 1+nbdy, nsize+nbdy
        output(i-nbdy) = sum4(i-nbdy)/4
      enddo
      end
```

Code 5.5: loop fusion without peeling

```
      subroutine doFast2(input, output)
      real*4  input(1-nbdy:nsize+nbdy)
      real*4 output(1-nbdy:nsize+nbdy)
c
      real*4   diff(1-nbdy:nsize+nbdy)
      real*4   sum4(1-nbdy:nsize+nbdy)
c
      do 9900 i = 1-nbdy, nsize+nbdy
       if(i .lt. 2-nbdy) goto 9900
       diff(i) = input(i) - input(i-1)
       if(i .lt. 1+nbdy) goto 9900
      sum4(i-nbdy) = diff(i-1-nbdy)
     &                + diff(i-nbdy)
     &                + diff(i+1-nbdy)
     &                + diff(i+2-nbdy)
      output(i-nbdy) = sum4(i-nbdy)/4
9900  continue
      end
```

Code 5.6: computational pipelining

```
      subroutine doFast3(input, output)
      real*4  input(1-nbdy:nsize+nbdy)
      real*4 output(1-nbdy:nsize+nbdy)
c
      real*4  diff(5),sum4(5),ave4(5)
c
      i5m0 = 5
      i5m1 = 4
      i5m2 = 3
      i5m3 = 2
      i5m4 = 1
      do 9900 i = 1-nbdy, nsize+nbdy
      i5m5 = i5m4
      i5m4 = i5m3
      i5m3 = i5m2
      i5m2 = i5m1
      i5m1 = i5m0
      i5m0 = i5m5
       if(i .lt. 2-nbdy) goto 9900
       diff(i5m0) = input(i) - input(i-1)
       if(i .lt. 1+nbdy) goto 9900
      sum4(i5m2) = diff(i5m3) +
     &             diff(i5m2) +
     &             diff(i5m1) +
     &             diff(i5m0)
      ave4(i5m2) = sum4(i5m2)/4
      output(i-nbdy) = ave4(i5m2)
9900  continue
      end
```

adjustment in loop bound serves as a preparation for the loop fusion.

Step2: Loop fusion is then applied to reduce the data reuse distance. We avoid peeling the computation for the boundary condition (red cells in Fig. 5.2) by inserting if statements into the fused loop (Code 5.5).

Step3: The array size can be minimized by exploiting the characteristics of stencil computation. In the example, no more than five values from these arrays are required to compute one output value. We can reduce the size of the working set by pipelining the results of the computation at a finer granularity. The size of temporary arrays can be only five for this example. The least-recently used location is overwritten by the latest computed result to reuse the memory space. Arrays become circular buffers and a group of barrel-shifted indices ($i5m0, i5m1...$) are assigned to identify the location in the circular buffers.

Following the optimization steps, Code 5.7 is the optimized output for Code 5.2. The briquette structure is also applied in this code to expose opportunities for SIMD engine usage. The *icube* and i loops are the nested loops that iterate through the grid planes. In order to update a given grid plane in the single-fluid PPM example code, we need data from 4 neighboring grid planes on each side in the dimension along the grid pencil. Only 5 grid planes of each variable are required, so that the p array, for example, in Code 5.7 is just 320 bytes in size. With this memory reduction from the full grid pencil to just 5 grid planes, most of the arrays can stay in the L2 cache memory. These transformations greatly reduce the memory footprint and eliminate a great deal of traffic between the CPU core and the main memory. The original triply-nested loop becomes another multiply-nested loop. The outermost two loops iterate through the grid pencils. Parallelization can be applied at this level to allow each single thread to update an individual grid pencil. These loops contain an inner loop over the multiple grid planes of a grid briquette (see diagram in Fig. 5.3). We then pipeline this loop nest over the briquettes in a strip to eliminate redundant unpacking of data records and redundant computation. Also shown in Fig. 5.3, prefetching and asynchronous write-back of data records are used to overlap computation with communication to the main memory.

Code 5.7: After memory optimization

```
      Dimension s(nssq)
      Real*4    unsmth
      Dimension pl(nssq,0:1)
      Dimension p(nssq,0:4)
      do 9990   icube = 0,ncubes+1
...
      do 9900   i = ibegin,ibegin+3
...
!DEC$ VECTOR ALWAYS
      Do j=1,nssq
        difmon(j)=pl(j,i1o0)-p(j,i4m4)
        s(j)=1.
        if((p(j,i4m3)-p(j,i4m4)).lt.0.)
     & s(j)=-1.
        difmon(j)=s(j)*difmon(j)
        if(difmon(j).gt.0.) then
        difmon(j)=max(s(j)*(pl(j,i1o0)-
     & p(j,i4m3)),0.)
        endif
        pl(j,i1o0)=pl(j,i1o0)-
     & unsmth*s(j)*difmon(j)
      enddo
...
9900 continue
9990 continue
```

5.4 Array size contraction

Computational pipelining greatly reduces the temporary array size and makes all the reduced-size arrays circular buffers in a computation with high memory efficiency. The lifetimes for the temporary values vary in the computation and the array sizes could be further reduced. A liveness analysis is applied to determine the live range for each temporary variable. The size of a temporary array, or circular buffer, can be determined by its live range. For example, *pl* has a live range of 2 and *p* has a live range of 5 in Code 5.7. Multiple groups of barrel-shifted indices are required to manipulate all the

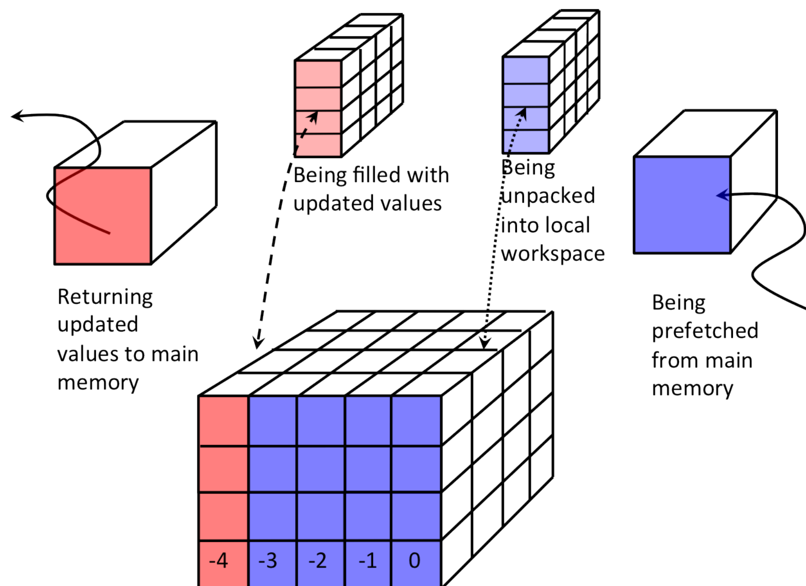

Figure 5.3: The pipeline illustration

different circular buffers. After the array size contraction, the memory footprint for a CFD application is minimized.

5.5 Performance result

To evaluate the performance result, we ran experiments on two different architectures with multiple compilers. The first platform is a cluster of dual quadcore Intel Xeon x5570 CPU (23.44 Gflops/s/core peak 32-bit performance) nodes. The compilers for the performance tests are the GCC compiler, version 4.6.1, the Intel compiler, version 12, and the Pathscale compiler, version 3.2. The second platform is an IBM workstation equipped with four 8-core Power7 (30.88 Gflops/s/core peak 32-bit performance) processors. On the IBM machine, we use the XL Fortran compiler, version 13.1, and the XL C compiler, version 11.1. We have chosen 3 test application codes that exhibit different types of computations, each of which is representative of a portion of the work that would be done in a full CFD application. These 3 test applications are: (1) PPM advection, (2) PPB advection, and (3) PPM single-fluid gas dynamics for low-Mach-number

flows.

To demonstrate the usefulness of our code transformations for CFD applications, we give performance results for each compiler working with 3 different code expressions. The first of these is a simplified and quite standard vectorizable Fortran expression, which is used as input to our code transformations. The computation is expressed in a sequence of triply nested, vectorizable loops extending over a regular 3-D Cartesian grid. The second Fortran code expression we use for performance testing is one that has undergone the on-chip memory optimization. A third and final code expression that we test for performance here is the result of our vectorization applied to this second, transformed and pipelined Fortran code. All the tests here are parallelized using 8 or 16 OpenMP threads, with either one or two threads assigned to a CPU core. The computational workload is equally shared among all the threads.

To demonstrate that the second expression is superior in managing and utilizing the cache memory than the first code expression, we perform a comparison between the two with both using no compiler optimizations. The second code expression performs at an average 2.4× speedup over the first code expression. This comparison result is shown in Fig. 5.4. We therefore believe the optimized code expression by itself produces a significant impact. Together with the most aggressive compiler optimizations, specified by the –O3 flag, this transformation achieves a significantly higher percentage of peak performance for CFD applications, and a 3.6× speedup over the original.

Fig. 5.5 shows the comparison between the results with and without on-chip memory optimizations. The best performance for the first expression comes from the PPM single-fluid code using the Pathscale Fortran compiler. However, the 1.4 Gflops/s/core delivered performance is only 5.9% of the peak performance. The large memory footprint of this computation leads to a tremendous amount of traffic between the processor and main memory. Execution becomes memory bandwidth limited using the first code expression. Compilers without access to domain-specific knowledge optimize the code to only a limited extent. Without efficiently reusing the cache memory, the achievable performance is limited by the available memory bandwidth. The pipelined, cache optimized code expression generated from our on-chip memory optimization, however,

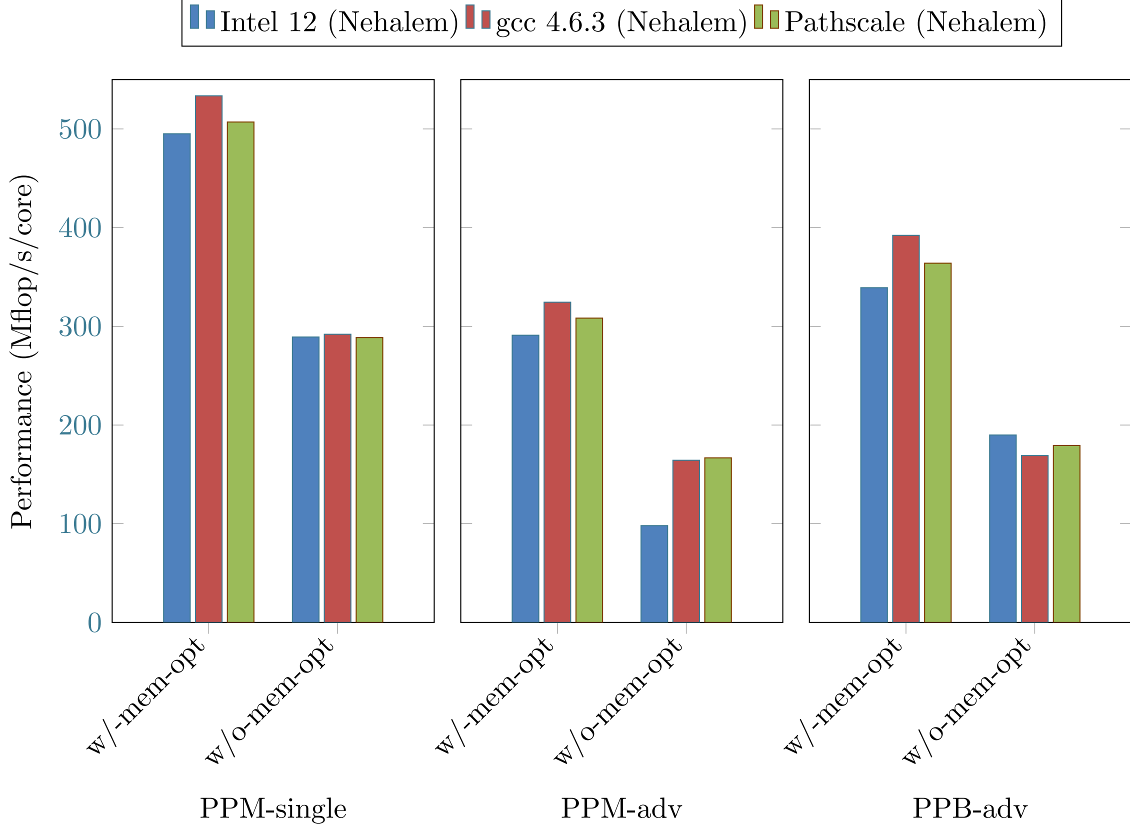

Figure 5.4: Performance comparison with no compiler optimization on Intel Nehalem

delivers much higher performance. The result in Fig. 5.5 shows an average of 3.6× speedup using this optimized expression. The memory footprint of the transformed computation is small enough to fit into the 256 KB L2 cache, so that fetching the briquette from main memory and writing the updated briquette back are the only demands placed on the very limited main memory bandwidth. If intrinsic function calls giving hints for the data prefetching are included in the transformed code, an even better performance is possible. We manually tuned the advection code using $mm_prefetch()$ and the Intel complier. This results in 3.3 Gflop/s/core performance, which gives an additional 5.5% performance increase.

In Fig. 5.5, we see that the Intel compiler achieves the best performance, at 14% of

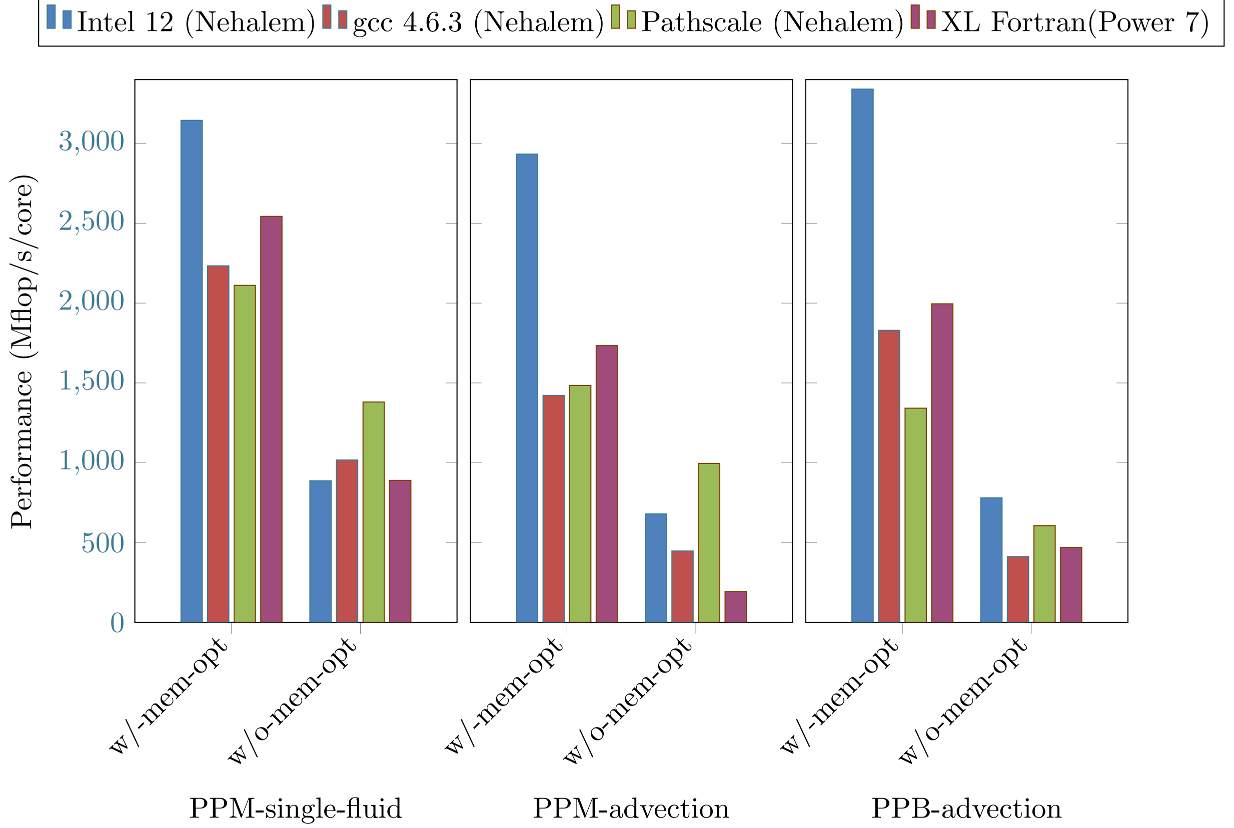

Figure 5.5: Performance comparison of on-chip memory optimization

peak, and we get 9.5% and 9% from GCC and Pathscale, respectively. Different compiler vendors rely on different benchmark software to validate optimization strategies. We have reviewed the optimization reports and the assembly codes generated by each of the compilers. We conclude that compiler vectorization leads to the 50% performance difference between them on our code examples. The conditional vector merge functions, known also as CVMGM in vector computing, which we discussed earlier, are allowed inside a vectorizable loop. However, expressing them with the if statement makes compilers sometimes treat them as jumps or branches. The Intel compiler seems to hoist many of these functions out of the loops. Loop distribution is then applied to create several smaller vectorizable loops. GCC and Pathscale avoid vectorizing these loops

entirely, because branches in loops violate their vectorization heuristics. These conditional vector merge functions are pervasive in our three examples. Partially vectorized execution therefore results in only a 50% overall performance improvement compared to non-verctorized execution for these loops.

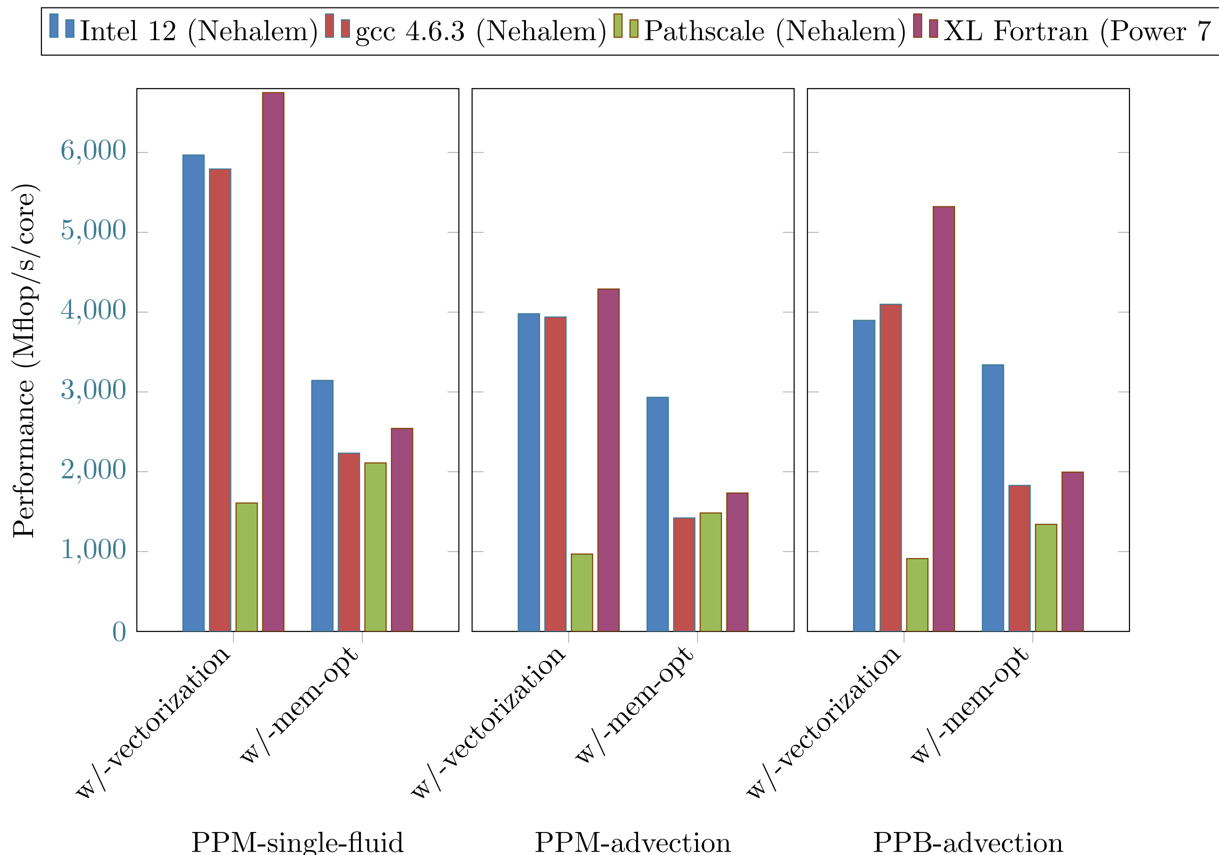

Figure 5.6: Performance comparison of vectorization

A third and final code expression that we test for performance here is the result of our vectorization applied to this second, transformed and pipelined Fortran code. A comparison between the second and the third code expressions is shown in Fig. 5.6. On the Intel hardware, we see that both the Intel compiler and the GCC compiler perform the best on the PPM single-fluid code. The achieved percentages of peak are 25% and 24%. The Pathscale compiler shows the worst performance in all three

examples for the transformed code. After inspecting the assembly code generated, we suspect the SIMD intrinsic functions are not fully supported by the Pathscale compiler. Instead of behaving as inlined functions, these intrinsic functions appear to be treated as regular functions. Thus the function call overhead drags down the overall performance. Excluding the Pathscale compiler, our translated SIMD expression has an average 2× speedup over the second code expression using the Intel processor. Experiments from the Power7 platform are also shown. With Power7's higher clock rate, the best performance is 6.75 Gflop/s/core, which is 21.8% of its peak performance.

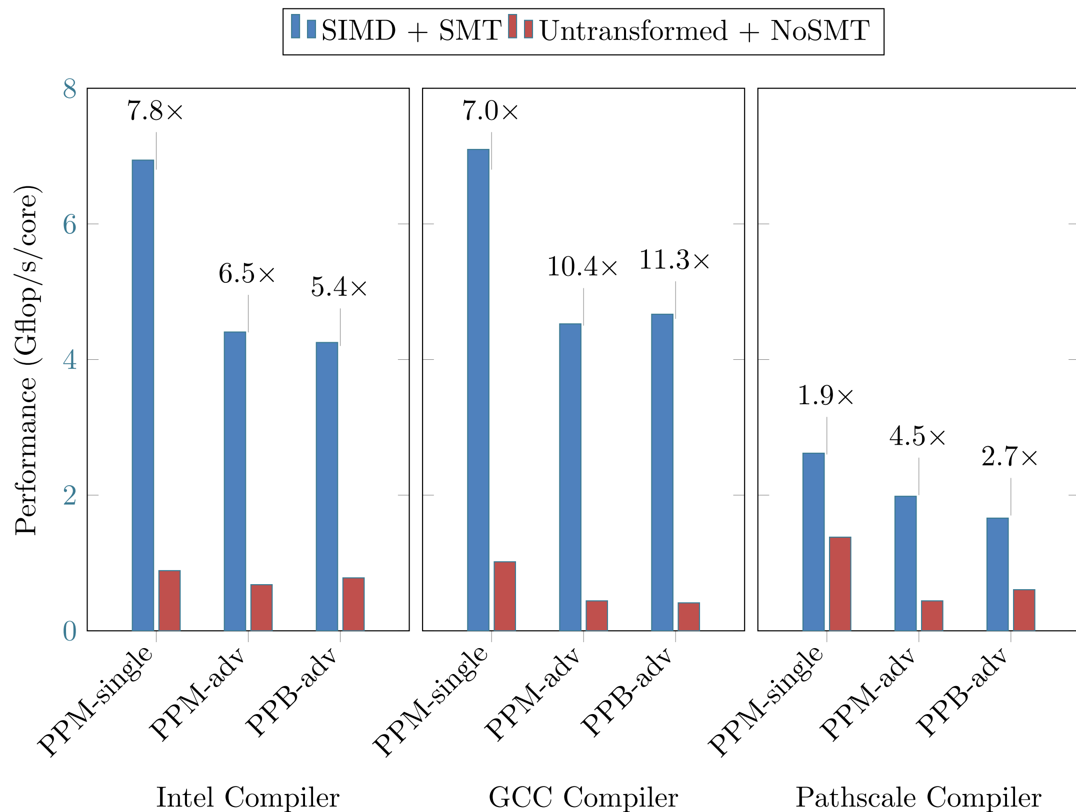

Figure 5.7: Speedups achieved by the on-chip memory optimization plus vectorization over the performance of the original code.

From these experiment results, we note that for the transformed code expressions that use small grid briquette data records, short, aligned vector operands, and fully

pipelined code executing almost entirely out of the on-chip cache we see by far the best performance. This transformed code brings the main memory data bandwidth requirement for the numerical algorithm to a very low level. The translated expression also utilizes the SIMD engines to the maximum extent. The Intel Nehalem processor used in our performance tests has a two-thread SMT design. Two threads in a single core share hardware resources and can be executing in any given pipeline stage concurrently. As shown in Fig. 5.7, the combination of our on-chip memory optimization and vectorization with the SMT achieve the best performance, 30% of peak, for the PPM single-fluid code, with $\sim$ 20% delivered on the other example codes. Using two threads per core does not improve performance on these codes in their original forms. Comparing the best-achieved performance with and without transformation, an average 9.6$\times$ speedup shown in Fig. 5.7 (or 7.3$\times$ including all the experiments) is obtained.

Chapter 6

Optimization for Limited Memory Bandwidth

Recent trends in HPC system design expose significant challenges to application code development. Multiple cores per CPU, multiple simultaneous threads per core, and highly complex memory hierarchies bring in higher computing power into a single processor. Compared to the fast improvement in computing power, the growth in memory bandwidth is only at a modest speed. A challenge we are facing in going to future HPC systems is to maintain a balance between sustained memory bandwidth and the peak computing performance. To tackle the memory bandwidth issue, we must exploit on-chip memory and increase the computational intensity, the number of flops we perform for each word that we bring onto or write off of the CPU chip. There are only two approaches to increase the computational intensity of a code. The first is to modify the numerical algorithm to one that exhibits a higher computational intensity. The second is to restructure the code so that it achieves a pipelined form that dramatically increases its reuse of data cached in the on-chip memory. Chapter 5 has discussed an approach to achieve higher computational intensity through on-chip memory optimization. However, it is applicable to only single core or single thread execution. With multiple computing threads competing for a limited memory bandwidth, a strategy involving multi-thread computation to increase computational intensity is a necessity for

the HPC research. This chapter emphasizes on the memory bandwidth optimization and presents one strategy that can significantly increase the computational intensity for the CFD applications.

6.1 Computational intensity

The ratio of floating point operations to the memory accesses (memory read from, or write to the distant memory), which we call computational intensity, plays a critical role to measure the computation efficiency. Looking back to the vector processor era, the balance between computational power and the memory bandwidth allows the Cray-1 to support half of the peak performance with a main memory bandwidth of only 1 flop/word. The Cray-1 exploited its 8 vector registers to accomplish this feat. The computational intensity of vector loops had to exceed 1 flop/word to get this performance on the Cray-1. CFD codes easily achieved this on a per-loop basis, without extensive pipelining and loop fusion. Using today's CPU designs, only processing the L1 cache-resident data can make it possible to achieve the same performance with the low computational intensities generally found in typical vectorizable loops of CFD codes. The disparity between computing power and memory bandwidth, as discussed earlier, exposes the importance of computational intensity. How to overcome the limited memory bandwidth bottleneck has become the primary challenge for the latest multi-core systems.

Fig. 6.1 shows a performance study of computational intensity using one of the CFD applications running on single nodes with dual quad-core Intel Nehalem CPUs with 2.93 GHz clock speed. The computational intensity of the application is about 44 flops/word. We add in more memory writes to its distant memory to lower the computational intensity. On the other hand, we add extra computations to the cache-resident data to increase its intensity. The curve in Fig. 6.1 shows that increasing the computational intensity, which moves the curve toward its right, will effectively increase the achieved performance. We believe that each CPU has its own curve like the one shown in Fig. 6.1, but with a different steepness at the cliff on its left and flattening at a different

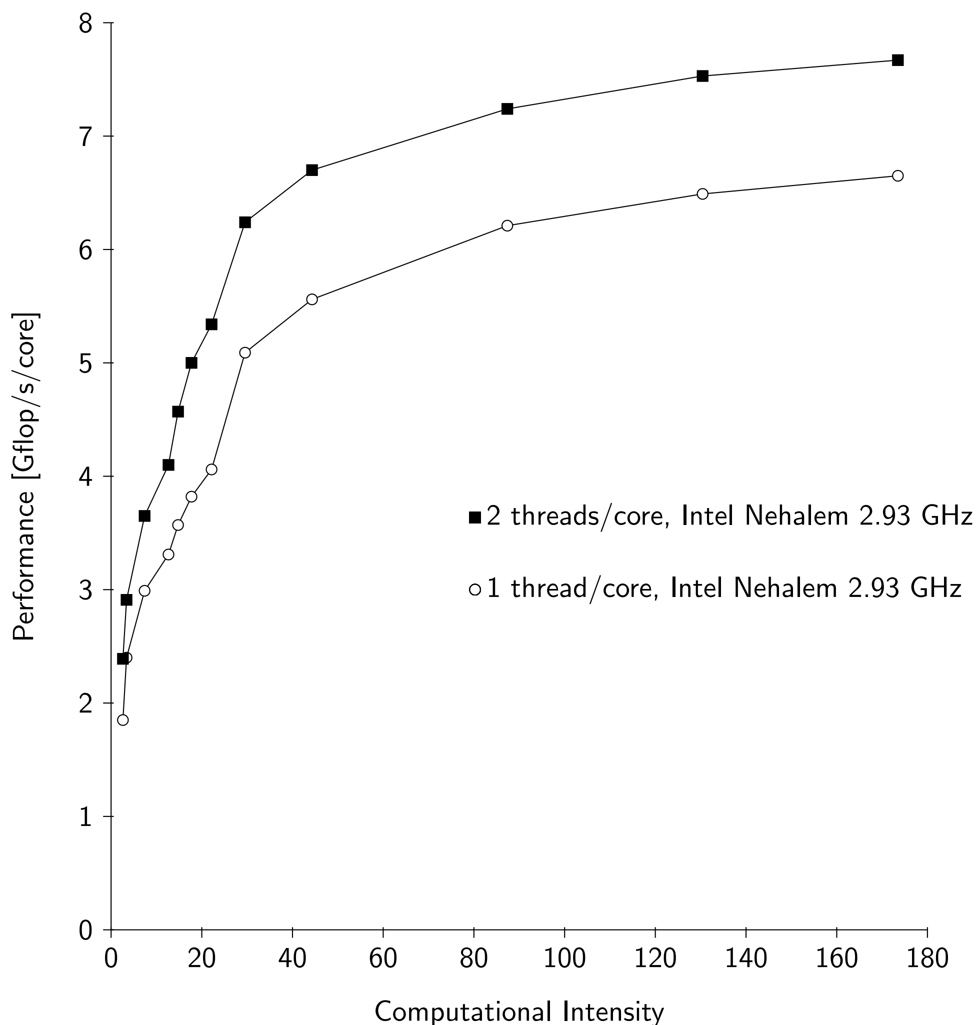

Figure 6.1: Performance impact from computational intensity

performance level. Increasing computational intensity is an important factor for high performance computing, but it is never easy or straightforward. Changing the numerical algorithm to expose higher computational intensity is one of the possible approaches. This may not be advantageous for every algorithm, it will not be adopted as our strategy in this dissertation. The second approach is to restructure the source code to tackle the challenges introduced by hardware features such as the limited size of the cache memory, a large SIMD width, high arithmetic pipeline latency, and/or the simultaneous

multithreading. All these challenges put tremendous pressure on the capacity of the limited-sized cache memory and tend to force the computational intensity toward its low end. Two computational pipelining strategies are presented in this dissertation. One focuses on cache memory optimization for single-core computation. The other one is developed on top of the first pipelining strategy to collaborate multi-thread/multi-core executions and push up the computational intensity.

6.2 Briquette-pencil pipeline

Stencil computation iteratively updates array elements according to some fixed patterns. This style of computation is commonly seen in CFD applications for solving partial differential equations and interpolations. Updating a single array element will reference one to many adjacent neighboring elements in different directions. Fig. 6.2(A,B) exemplifies the multi-dimensional stencil computation. Fig. 6.2(B) also illustrates the worst scenario that all the surrounding elements (26 surrounding elements in 3-dimensional update) are required to update the center element. This is expected to generate a huge computational overhead and lead to low performance. Grouping the array elements is the most highly recommended approach to lower the overhead from the neighboring computations. However, it enlarges the memory footprint and increases the cache memory pressure. To avoid high overhead, the CFD applications in this dissertation have always been formulated instead to directional sweeps. Three separate dimensional updates are performed for the 3-dimensional computation. The directional sweeps significantly reduce the size of required neighboring data (shown in Fig. 6.2(C)). The briquette data structure is further applied to the directional sweeps. The briquette size is adjusted based on the cache memory size, SIMD width, and also size of difference stencils required in the numerical algorithm. The computation shown in Fig. 6.2(D) seems to generate the most overhead for a one-directional sweep. However, we can further pipeline the computation and iteratively reuse the computed and cached data.

Fig. 6.3 briefly illustrates briquette-pencil pipelining. Eight contiguous briquettes form a single briquette pencil in this example. The ghost cells required for this pipelined

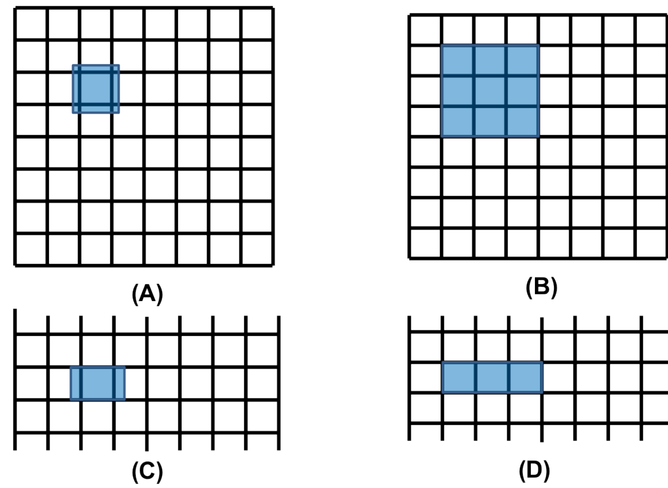

Figure 6.2: Stencil computation

update fit into two briquettes in both ends of the pencil. In the pipelining process, the computation performs the following operations: fetching one briquette data record from the main memory (the black block in figure); executing computation on one cached briquette (the gray block in figure); transposing one updated briquette then writing it back to main memory (the dotted block in figure). Details of the briquette-pencil pipeline are presented in Chapter 5. Briquette-pencil pipelining not only optimizes the on-chip memory utilization, it also increases the computational intensity significantly.

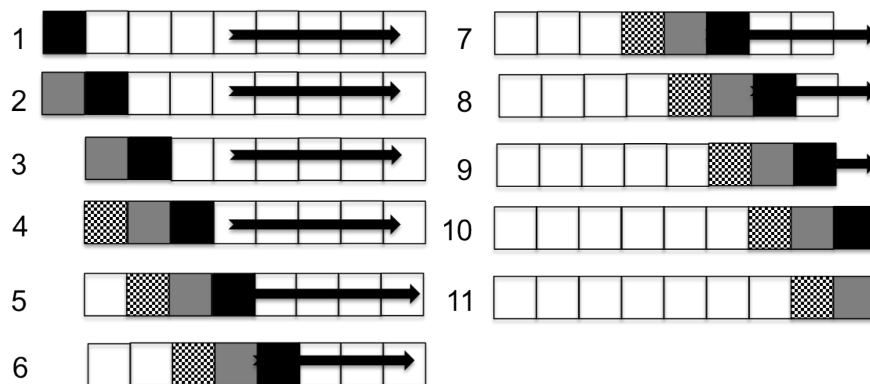

Figure 6.3: Briquette-pencil pipeline

6.3 Briquette-block pipeline (3-dimensional pipeline)

The briquette-pencil pipeline focuses on delivering high performance for a single-thread computation. The computation is executed by a single thread in a CPU core, and all the contents required for the computation are stored in the private on-chip memory (usually L1 and L2 cache memory). As long as all briquette-pencil updates are independent, the parallelization is performed at the granularity of a briquette-pencil. Fig. 6.4 shows the parallelization via briquette-pencil pipelining in a single compute node. All computing threads iteratively update briquette-pencils in the same sweeping direction. After completion in one sweeping direction, all compute threads immediately start the iterative operations in the next sweep direction. Six memory operations for each briquette(one load and one store in each 1-dimensional update) are unavoidable because of the limited cache size and memory locality. If the memory traffic to the distant memory can be reduced, a much better computational performance and higher computational intensity can be foreseen as predicted in Fig. 6.1. Two memory operations for each briquette (one load and one store) would be sufficient for a 3-dimensional update when sufficient cache memory space is available. The limited cache memory size in current processor designs immediately exposes the practical challenges. This second pipelining strategy, briquette-block pipelining, is developed to reduce the memory traffic for high computational intensity.

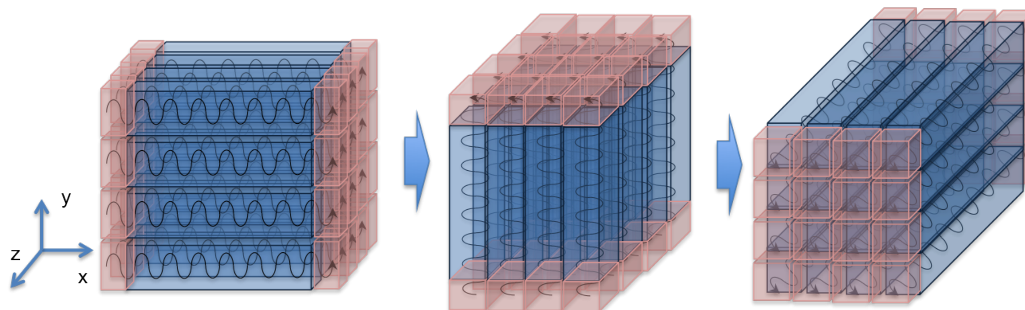

Figure 6.4: Parallelization briquette-pencil pipeline

In recent multi-core processor designs, each computing unit has a small private L1 data cache (16 KB $\sim$ 32 KB) and a larger L2 cache memory. Depending upon the CPU

vendor, the L2 cache may be private (256KB/core for Intel Nehalem core) or shared (2MB shared by two integer cores for AMD Bulldozer.) We have seen the trend that a much larger L3 cache, or last level cache (LLC), is provided but will be shared by all the cores on the CPU socket. The aggressive memory footprint reduction in the briquette-pencil pipeline can effectively reduce all the data contents to fit into the L1 and L2 cache space[67, 53, 68]. The last level cache with size of several megabytes becomes the ideal place to host a fair amount of updated data. The fundamental concept of the 3-dimensional pipeline is to exploit the large LLC and relieve the memory bandwidth pressure.

To implement the 3-dimensional pipeline, a temporary array is reserved to store a group of briquettes. The size of this array is determined by the size of the last level cache, the number of threads used for the computation, and the numerical algorithm. The array is used to store a reasonably large subdomain in the grid. In the beginning of the computation for this subdomain, the briquettes are fetched (this is a compulsory fetch) and copied into this temporary array. The updated briquettes are written back to the same temporary array in last level cache and all subsequent dimensional updates fetch their input from the last level cache. Fig. 6.5 shows the implementation of 3-dimensional pipelining on the AMD Interlargos processor. Up to 8 concurrent threads from a processor are allowed in the computation. Each small square in the figure represents a briquette-pencil update. The temporary array in LLC can store $8 \times 8 \times 8$ briquettes and the 3-dimensional pipeline can update a grid of $6 \times 6 \times 6$ briquettes. The same figure will be used to illustrate the details of the 3-dimensional pipeline implementation.

The pipeline implementation starts with the first 1-D sweep, or X-pass update in Fig. 6.5 (A). With the assumption that the X-directional update requires transverse data from neighboring briquettes in the Y- or Z- dimensions, at least 4 adjacent briquette-pencils(in both directions for the Y- and Z-directions) are required in the computation. 8 parallel threads will be updating the 8 briquette-pencils in the same X-Y briquette-plane concurrently, and the computation iterates from the left to the right in the Z-direction. In each single iteration, the computation prefetches data(the black squares), updates

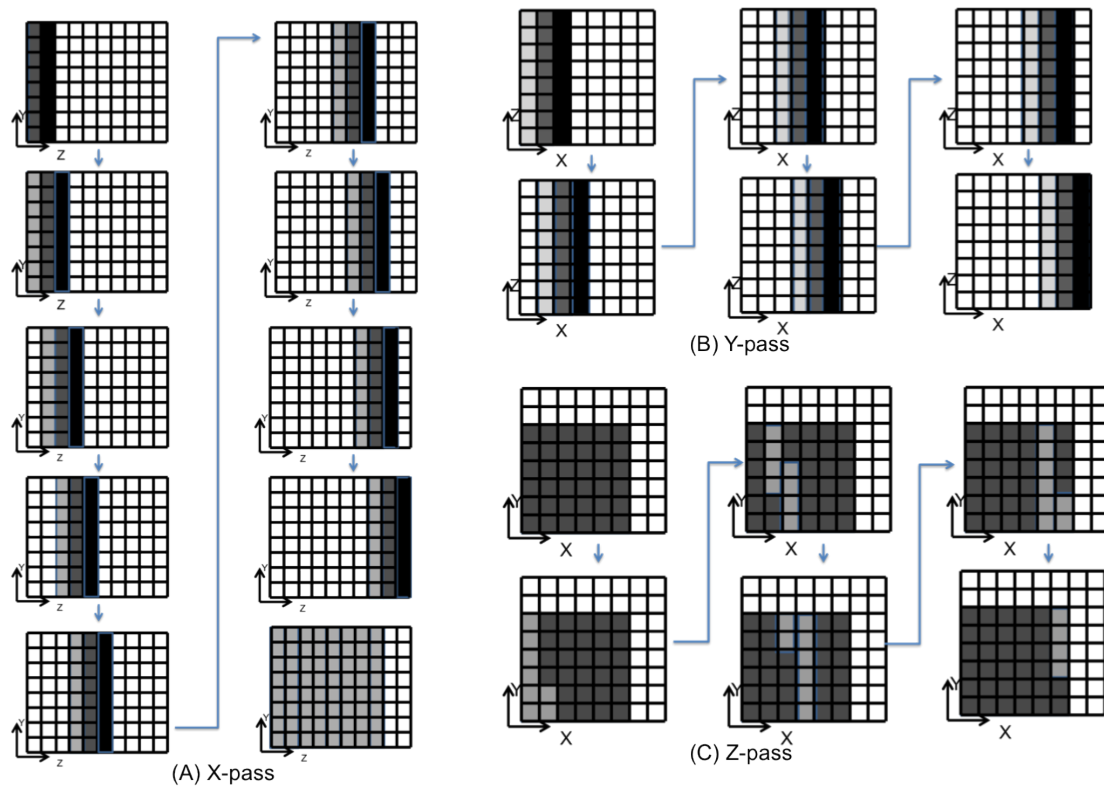

Figure 6.5: 3-D pipeline

briquette-pencils (the dark gray squares), and writes updated data into the temporary array (the light gray squares). This guarantees that there are always 3 X-Y briquette-planes fetched for computation and the 8 threads can parallelize the computation in the X-Y plane. Once a briquette-pencil update is completed, we overwrite the pre-existing data in the temporary array with the updated data (shown in light gray squares). After the completion of 8 briquette-pencil updates, shown in the bottom-right diagram in Fig. 6.5 (A), the pipelining will use the updated briquette-block for the computation in next sweep direction. Meanwhile, there are 2 briquette-planes with unmodified data that will be kept and used for the next 3-D briquette-block update. Similar to the briquette-pencil pipeline, this temporary array is implemented as a circular buffer in this 3-dimensional pipeline.

The next 1-D sweep, or Y-pass update is shown in Fig. 6.5 (B). Similar to the

previous step, transverse data is required in the computation. The pipeline again fetches data from the temporary array, performs the update computation and writes data back to the temporary array. In contrast to the previous directional update, the input data is cache resident in LLC and can be fetched directly from the last level cache. After the completion of 6 briquette-pencil updates in the Y-direction, the pipeline is ready for the last directional update.

The last 1-D sweep, or Z-pass update, performs computation with the updated data from the previous step. Assuming the numerical algorithm needs no transverse data for this directional update, all briquette-pencil updates are independent and can be parallelized freely by 8 concurrent threads. Fig. 6.5 (C) illustrates one example of parallelization. After the completion of the three updates, the final updated data is written back to main memory, and the temporary array in last level cache is reused in the next briquette-block update.

Using this implementation to update a briquette-block of 6^3 briquettes, a total of 756 briquette updates ($3 \times 6 \times 6 \times 7$) are performed. With the 3-dimensional pipeline presented earlier, an efficiency of 73% is achieved with an overall number of 1036 briquette updates ($8 \times 8 \times 7 + 8 \times 6 \times 7 + 6 \times 6 \times 7 = 1036$). The temporary array is implemented as a revolving buffer for the Z directional update. This makes only 840 briquette updates ($6 \times 8 \times 7 + 2 \times 6 \times 6 \times 7 = 840$) required, with 81% efficiency, for the following briquette-block updates. Although there is 20% to 30% overhead in the computation, the number of memory operations to the distant memory in a 3-dimensional pipeline is reduced by a factor of 3.

6.4 Performance result

The briquette-pencil pipeline has become a fundamental technique adopted in our CFD application development. We have done an elaborate performance study of the briquette-pencil pipeline with multiple CFD applications. A significant performance improvement can be delivered solely by the briquette-pencil pipeline. A best-achieved performance is 30% of the 32-bit peak performance on an Intel Nehalem CPU with 2.93

GHz clock speed. With increasing complexity in the numerical algorithm and more physical-state variables involved in the computation, it is more challenging to keep the computation at this high performance.

In order to evaluate the 3-dimensional computational pipeline presented in this dissertation, we perform a series of experiments with multiple CFD applications developed in our team. Two hardware systems are used in these performance studies. The first platform is a cluster of dual quadcore Intel Xeon x5570 CPU (23.44 Gflops/s/core peak 32-bit performance) nodes. The compiler used on this platform is the Intel compiler in version 9.1. The second platform is the Cray XE6 system with two 16-core AMD Opteron 2.3GHz Interlagos processors in a single compute node. The compiler for this Cray system is the Cray compiler in version 8.0.3. An Intel Nehalem core has one 32 KB private L1 data cache, and a 256 KB L2 private cache. An 8 MB L3 cache is shared among the 4 cores on a socket. The 128-bit ALU in the Nehalem core can perform a 4-way SIMD execution using the SSE instruction on each cycle. Two integer cores in one AMD Interlagos module share a 256-bit ALU and are able to perform one 8-way SIMD execution using an AVX instruction, or two 4-way SIMD executions using SSE instructions. Each Interlagos integer core has a 16 KB private L1 cache. Two cores in a module share a 2 MB L2 cache and also a 2 MB L3 cache. The aggregate L3 cache is 8 MB in one AMD Interlagos socket. The performance study presented in the following focuses mainly on the performance impact brought by computational intensity. It also evaluates the benefits gained from the 3-D computation pipeline presented in this paper. The performance comparison between different CPUs from different vendors is outside the scope of this paper. The performance differences could be from the fundamental difference in hardware design, such as L1 cache memory size, cache writing policy, and memory bandwidth.

6.4.1 PPM advection

The first application chosen in the performance study is the PPM advection. This same application is used to generate the performance study shown in Fig. 6.1. Details of this application can be found in the earlier paragraph. This application applied with

the briquette-pencil pipeline delivers a 6.7 Gflop/s (28.6% of 32-bit peak performance) performance on a single Nehalem core having two threads running on a single core. On the Cray XE6 system, it delivers 3.6 Gflop/s/module (9.8% of 32-bit peak performance) with two threads on one AMD Interlagos module. Since no transverse data is needed for this application, we implement the 3-D computational pipeline following the simpler example described in previous section. The 3-D pipelined code delivers almost identical performances for both CPUs. But the profiling analysis shows higher percentage of execution time taken for the OpenMP barrier. Since each grid cell in the PPM advection reads in and writes out only one variable, it might not be able to expose the issue of the limited memory bandwidth. The applications with more complexity and higher bandwidth requirements are preferred in this study.

6.4.2 PPM-star simulation

The second example code we use here is the PPM gas dynamics code that has been used to run the stellar convection problems cited in [69, 70]. This is a fully functional 3-dimensional code that has been run on over 98,000 CPU cores to simulate the flow of the helium shell flash of a giant star. Running this application with 64-bit computation delivers 2.38 Gflop/s (20.3% of 64-bit peak performance) per Nehalem core and 1.09 Gflop/s (5.9% of 64-bit peak performance) per AMD core. Our choice here of 64-bit computation is driven by the accuracy requirement of the stellar hydrodynamics applications that this code targets.

The numerical algorithm in PPM-star needs no transverse data, but it takes 15 input variables and generates 15 output variables. Significant memory traffic can be expected in the briquette-pencil pipeline implementation. In the 3-dimensional pipeline implementation, the temporary array contains 10^3 briquettes with briquette of 4^3 grid-cells, that is 3.7 MB in size. We have also modified the numerical algorithm slightly to increase the accuracy in 32-bit computation. The 3-D pipelined code running in 32-bit computation delivers 7.3 Gflop/s (31% of 32-bit peak) per Nehalem core, and 2.84 Gflop/s (15.4% of 32-bit peak performance) per AMD interlagos core. Comparing the achieved percentage of peak performance, a 1.5× of performance improvement is

seen on Intel CPU, and 1.3× for the AMD CPU. The significant performance improvement inspires us to move forward for applications with more complex algorithms using transverse data.

6.4.3 Inertial confinement fusion

In the next example, we are working with a full, 3-D PPM gas dynamics code, which tracks multiple fluid constituents in the presence of strong shocks. This code simulates a test problem that corresponds to the unstable compression of fuel in a laser fusion capsule. A performance study using this application has shown that the achieved performance on the Intel Nehalem CPU is 4.33 Gflop/s per core, which is 18.5% of the 32-bit peak performance [71]. A recent performance test on the Cray XE6 system achieved 2.54 Gflop/s per AMD Interlagos module, which is 6.9% of the 32-bit peak performance. A much lower performance efficiency is delivered with this much more complex application than with the earlier, simpler examples.

Following the 3-dimensional pipeline strategy discussed above, the coordinated 3-dimensional pipelined code improves the performance up to 6.5 Gflop/s (28% of 32-bit peak) per Nehalem core using two simultaneous threads in a single core. It delivers a 1.5× performance improvement for this full, 3-D gas dynamic code. On the AMD platform, a performance of 4.54 Gflop/s per AMD Interlagos module is achieved, which is 12.3% of its 32-bit peak. A higher performance improvement of 1.8× is gained on the Cray XE6 system using the proposed briquette-block pipeline.

6.5 Discussion

The key idea proposed in this section is to increase the computational intensity for more efficient computation. The first computational pipelining strategy, the briquette-pencil pipeline, has shown significant performance impact in multiple CFD applications. The second computation pipelining strategy, the briquette-block pipeline, is built upon on top of the briquette-pencil pipeline to reduce the unnecessary memory traffic to the maximum extent. The prerequisite hardware support for the briquette-block pipeline is

a larger cache memory space. The discussion of several practical concerns will be helpful for the further development and implementation using the briquette-block pipeline.

briquettes in X or Y	briquettes in Z	Mwords in/out	Mflops	intensity	temporary (MB)	overall efficiency
6	6	0.7109	73.77	103.8	2.66	77%
6	24	2.4688	245.53	99.5	7.44	78%
10	6	1.7109	187.71	109.7	5.98	83%
10	20	5.0469	528.67	104.8	14.34	85%
14	7	3.5898	400.06	111.4	11.69	87%
14	14	6.6797	722.07	108.1	19.13	88%
14	21	9.7695	1044.09	106.9	26.56	89%
18	12	9.2656	1019.73	110.1	26.56	90%
2	1	0.0508	4.42	86.9	0.33	47%
2	2	0.0703	5.98	85.1	0.40	50%
6	1	0.2227	26.06	117.1	1.33	65%
6	2	0.3203	35.61	111.2	1.59	72%
6	3	0.4180	45.15	108.0	1.86	74%

Table 6.1: 3-D pipelining

The first concern is the cache memory space. Table 6.1 quantifies the tradeoffs involved in attaining high computational intensity by performing entire grid block 3-D updates using a 3-D briquette array (ddtemp) that resides permanently in a shared (L3) cache.

Comparing to a 1-D single-pencil update with computational intensity of 30.9 flop/-word, most of the rows in Table 6.1 show a triple intensity. Having all the CPU designs with different L3 cache size, this table shows the appropriate configuration for the cached temporary array. As shown in the table, the light gray rows are appropriate selections for CPUs with 8 MB L3 cache. For the same reason, the mid gray rows are for the CPUs with16 MB L3 cache, and dark gray rows for the CPUs with 32 MB L3 cache. The two CPUs used in the performance experiments have 8 MB shared L3 cache space are expected to have roughly 80% efficiency. The performance results have shown 1.5× to 1.8× improvement using the briquette-block pipeline. Applying briquette-block pipeline to CPUs with larger L3 cache space, like the IBM Power-7 CPU, is expected to deliver

more promising improvement due to the high efficiency.

The second concern is thread coordination. The examples we used to illustrate the strategy, and the applications we chose for the performance study all use at most 8 threads to perform the briquette-pencil updates. Referencing the example shown in Table 6.1, thread synchronization has to be implemented to make sure each thread has all the transverse data ready for its briquette-pencil update. For many-core processors, like the Intel MIC architecture, challenges in thread coordination using a huge number of threads is unavoidable. Current multi-thread programming models might not be sufficient to handle these new difficulties.

The third concern comes from the numerical algorithm. From the previous study with the briquette-pencil pipeline, we have found that knowledge from domain experts is needed to assist the pipeline transformation. The same statement holds for briquette-block pipelining. First of all, the algorithm has to be directionally split to gain the advantages from both computational pipelining strategies. Application developers have to evaluate the feasibility of using the briquette structure, and they must be willing to re-pack the data into the briquettes. The stencil distance required in the algorithm determines the briquette size. It further dominates the size of the temporary array used to perform the 3-D pipeline. These concerns in algorithm design need to be addressed in future research.

The last concern comes from the software tool. We have built our own tool to perform code transformations used to implement the briquette-pencil pipeline. This tool has shown its usefulness to assist CFD application developers to generate pipelined and optimized code. A continuous development of this tool to support new pipeline strategies is in the development roadmap. With more complexity brought by the briquette-block pipeline, a more robust tool and development platform is required.

Chapter 7

Transformation Framework

7.1 Source-to-Source transformation

We have decided to adopt the source-to-source approach to implement the transformation framework. The reasons for and benefits of the source-to-source approach are summarized in the following:

1. Source-to-source transformation is popular in language translation. Without implementing the applications in new languages, the transformation can minimize the programming efforts for the target languages. In this dissertation, we rely on the translation to implement CFD applications on HPC systems with only limited language support, e.g. IBM Cell processor supports only C/C++ language.

2. Optimization capabilities vary among vendors' and research compilers. Performing optimizations following the user's directive might not be feasible. After code generation to a binary format, performing additional optimizations in binary format is challenging. Source-to-source transformation performs optimizations and generates output in the representation of a high-level language. Users can perform available optimizations freely and execute multiple iterations of source-to-source transformations. The transformations can be both hardware and compiler independent. The optimization portability can be maximized by source-to-source

transformation.

Fig. 7.1 summarizes the design of the transformation framework. The transformation framework needs to have the capabilities to parse source code, perform transformations, and generate optimized output. Domain-specific directives are inserted into the source code to direct the optimizations. With higher maturity in the transformation framework, some directives can be removed and replaced with automatic optimizations in the future. Compiler techniques are adopted to assist optimization and perform required analysis. Optimizations are developed within the framework and transformations are performed in the intermediate representation (IR), or the abstract syntax tree (AST). The ideal framework also needs to support multiple programming languages to maximize the optimization portability. After completing the source-to-source transformations, the optimized, transformed output code is submitted to the vendor's compiler.

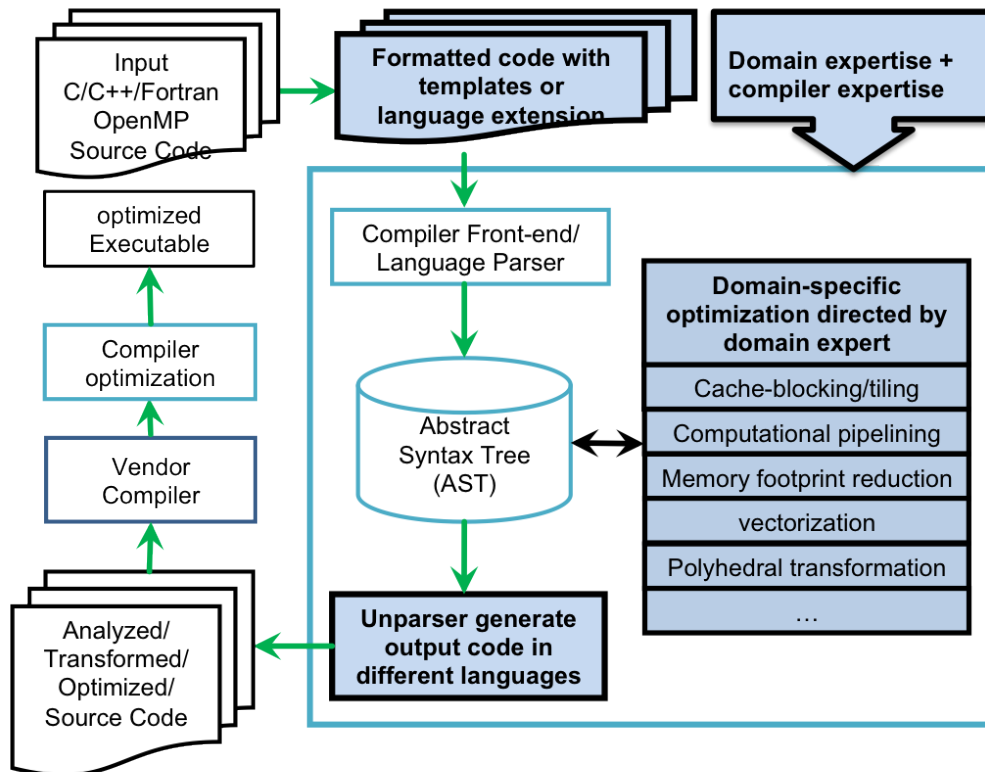

Figure 7.1: domain-specific transformation framework

There are many existing research projects focusing on high-level code transformation to improve performance and generate automatic parallelization [72, 11]. Research compilers also provide abilities to generate source-to-source translation for verification purposes[10]. After a series of experiments with available open-source infrastructures, we have found most of them focus on targets and programming languages different from our requirement. The different perspectives and views in code transformation create a steep learning curve and make the addition of new features difficult. We have therefore decided to adopt ANTLR (ANother Tool for Language Recognition) [73, 74] as our preliminary infrastructure, and to build the translation tool from scratch. We later found the design of the ROSE compiler [7] fulfills the needs in both source-to-source transformation and domain-specific optimization. With broader language support and more compiler techniques involved, the ROSE compiler is identified as a good development platform. The following sections describe the implementations of the proposed transformation framework.

7.2 Source-to-Source implementation

7.2.1 ANTLR-based tool

ANTLR is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. In contrast with Lex and Yacc, ANTLR includes several features in the followings:

1. ANTLR generates the LL(*) recognizer, extension of LL(k) that uses arbitrary lookahead to make decisions, to parse the source code.

2. ANTLR takes abstract syntax tree (AST) construction rules to create an AST as an intermediate format in transformation. It also generates graphical tree output to verify the tree transformation result.

3. ANTLR requires language grammars to create parsers and AST generators for different programming languages. In our current implementation, we implement

a subset of Fortran grammar to support our scientific applications written in Fortran.

4. ANTLR supports predicates that implicitly specify the precedence of the conflicting alternatives and provides flexibility in generating transformations.

5. StringTemplate[73, 75] is tightly integrated with ANTLR to create a structured output format and greatly simplifies the work in output generation. Through different templates, code can be transformed to different expressions.

The following additional features are integrated in the translation tool developed by our team to enable our specific transformations:

1. A symbol table is designed to collect details of the application. The translator will reference the symbol table to retrieve information for the transformation. Variable and subroutine declarations will also be based on the symbol table.

2. User directives are required for the translation tool to guide the transformations. In addition to standard directives, such as OpenMP directives, the framework allows customized directives for optimization guidance. This auxiliary information can be parsed and stored for different purposes in different phases of the transformation.

3. An EQUIVALENCE statement in the Fortran language is used to specify the sharing of storage units by two or more entities in a program unit. To simplify the alias analysis in this framework, the symbol table keeps track of all entities that share storage units.

4. Instead of generating only one single output language, we create multiple backend translations for different programming languages and language extensions for different architectures.

The implementation with ANTLR will take Fortran source code with user directives inserted as input. It will generate AST and a symbol table with all the user directive

information stored in them. In the transformation process, the translator will follow guidance from the user directives to generate the expected output form. All transformations are done in high-level expressions; the user can verify and compare the output from the graphical interface at each step of the transformation. After the transformations are completed, the transformation generates Fortran output by default. Upon the request of the user, the translator will also generate output in different languages. The latest implementation with ANTLR consists of the following components: (1) Fortran subroutine inliner, (2) on-chip memory optimization that can perform briquette-pencil pipelining, (3) vectorization framework that can support multiple SIMD instruction sets (SSE, AVX, Larabee Instruction, AltiVec, QPX), and (4) a programming language translator.

7.2.2 ROSE compiler

The ROSE compiler is a open-source compiler infrastructure developed at the Lawrence Livermore National Laboratory to build source-to-source program transformation and analysis tools. The earlier research work in ROSE development focused in libraries written in the C/C++ language. The supports for Fortran and other programming languages were later integrated into ROSE platform. The compiler packages consist of three major parts, front-end, mid-end and back-end, and several auxiliary features. The front-end adopts a commercial parser, EDG parser, to provide parsing capabilities for C and C++ languages. A research parser, the Open Fortran parser [76], is used to support the Fortran language. The IR in ROSE is SageIII, to acknowledge the Sage++ and SageII projects by Dennis Gannon and Karl Kesselman in the early 90's. Similar to most compiler frameworks, the mid-end consists of a group of transformations and program analyzers. The back-end supports unparsing for multiple output languages. A collection of APIs are provided in the ROSE compiler to perform tree traversal, code transformation, and trigger code analysis.

A variety of research projects have been built based on the ROSE infrastructure. For example, POET[11, 77] is a scripting language designed for domain-specific optimization, and PolyOpt (C and Fortran) is the optimization tool using the polyhedral

transformation model. Transformation tool developers have the flexibility to reuse existing optimizations and analysis. We have implemented a subset of the optimizations in the ROSE compiler framework. Meanwhile we also foresee more research potentials using the ROSE compiler. For the following reasons, the ROSE compiler is identified as a good infrastructure for the transformation framework.

1. It has a state-of-the-art source-to-source compiler framework with on-going development and maintenance.
2. It supports major programming languages used in scientific programs.
3. A variety of optimizations and analytical models are available in the ROSE compiler.
4. The IR design in the ROSE compiler is compatible with our implementation with ANTLR. The development effort in the migration process therefore can be minimized.

7.3 User-directive interface

The transformation framework integrates user directives in the compilation process. The directives can be categorized as follows: (1) identifying critical computation regions, (2) providing domain-specific information, (3)performing transformations, and (4) identifying special functions. The directives developed in the current implementation are listed in Table 7.1.

In the current transformation framework, transformations discussed in the followings are invoked by the user directives. Subroutine inlining is first executed for every subroutine call marked with the "INLINE" directive. The code size will be expanded for further optimization according to the developer's request. The framework then takes the directive "PIPELINE" to identify the beginning of the pipeline region. All nested loops in the pipeline region with the "LONGITUDINAL LOOP" directive will be fused as described in Chapter 5. The directive "ELIMINATE REDUNDANT ITERATIONS"

DIRECTIVE	PURPOSE
INLINE	Inlining all the relevant subroutines
PIPELINE	Identifying the specialized pipeline loop
LONGITUDINAL LOOP	Identifying all the computational loops over planes
ELIMINATE REDUNDANT ITERATIONS/SUB SCRIPTEXP	Removing redundant reads from the main memory
PREFETCH BEGIN/END	Identifying the pipeline-prefetching region
DOUBLEBUFFER	Identifying the variables that require double-buffering
VECTOR ALWAYS	Identifying the vectorizable loops
DMA	Identifying the data movement to/from main memory
UNROLL	Unrolling the loops
SIMD UNROLL	Unrolling the SIMD format

Table 7.1: List of directives used in optimization

identifies and removes the redundant memory reads. "PREFETCH BEGIN/END" and "DOUBELBUFFER" are used to identify the prefetching region and the variables that will need double buffering for the prefetch. "PREFETCH" also identifies the location to insert prefetching intrinsic functions supported by the latest SIMD engines. Vectorization is performed in loops identified with "VECTOR ALWAYS". Two types of loop unrolling are available before (identified with "UNROLL") and after (identified with "SIMD UNROLL") vectorization. The directive "DMA" is used to identify data movement between the main memory and the on-chip memory. It is reserved for architectures like the Cell processor. This same directive can be used to identify the position for software prefetching instructions.

Directives provide a lightweight interface for application developers to convey domain knowledge to the compilation process. Although it slightly increases the programming burden by inserting required directives into source code, the reward in performance improvement is significant. As the development evolves, several directives could be replaced by automatic optimizations and analysis, and innovative directives will be designed for new optimization purposes.

Chapter 8

Discussion and Performance Study

We have implemented the CFD applications and applied the optimization strategy presented above on the leading large-scale computing systems. The basis for the code transformations that we exploit in this dissertation is experience in recent years with the IBM Cell processor. In this chapter, we therefore briefly describe this processor and identify the significant respects in which we can view all modern computing devices in this same way. We select representative CFD applications to perform experiments on multiple HPC platforms. A quantitative study is presented and performance results are discussed.

8.1 Viewing modern processors from a Cell programmer's perspective

The IBM Cell processor contains one PPU and eight SPUs. Each SPU is a 4-way SIMD processor that operates on quadword (128-bit) operands. Each SPU has a 256 KB on-chip local store for both data and instructions. At 3.2 GHz, each SPU offers 25.6 Gflop/s peak performance. The SPU is equivalent to a vector machine that operates upon data stored in its on-chip memory. Programmers run the same program (SPMD)

on each of the 8 SPUs, and then explicitly manage the prefetching and writing back of data into and out of the local stores. The PPU is used to perform logistical work such as data initialization and synchronization.

The multicore CPUs chosen in this study are the processors from Intel, AMD, and IBM. Similar to the Cell processor, each core in any of these CPUs is a 4-way, or 8-way SIMD processor that operates on packed operands. Each core has a private L1 cache (data and instruction) and private on-chip L2 cache. SMT allows multiple threads to share the hardware resource and to execute concurrently. In contrast to the explicit on-chip memory management on the Cell processor, multicore CPUs have implicit cache management that supports cache coherency.

The Nvidia C2070 Fermi GPU contains 14 SMs. Each SM is equivalent to two 16-way SIMD engines. Each SM can be viewed as two Cell SPUs, each having one 16-way SIMD engine. Therefore, the Fermi GPU is as a kind of Cell processor with 28 SPU-like vector engines, each with private on-chip memory. In place of the SPU's 256 KB local store, the SM has a variety of on-chip stores each with different uses and limitations. There is an 8 KB cache working set for constant memory, 64 KB configurable on-chip memory (L1 cache memory and shared memory) in addition to a large number of registers. It is natural for a Cell programmer to consider running the vector code for each SPU on a GPU's SMs, but instead of using vectors of length 16, a larger vector length will be needed and the complexity of the SM's on-chip memory must be accommodated. Despite the slower GPU clock, the 16-wide SIMD operation of each SM and the large number, 28, of SIMD engines leads to a very high peak performance of 1030 Gflop/s with 16 multiply-adds performed on each SM on every clock tick. This makes conversion of Cell code to the GPU very attractive.

We choose to program CPU cores and the GPU's SMs, as we do Cell's SPUs as vector machines. We can usefully compare these new vector engines to classic Cray CPUs, for which the scientific community has decades of experience. The primary distinction of these vector machines relative to the Cray classic is that both the SPU and the SM have local, on-chip memories, while the Cray-1 operated upon vectors in main memory and had only 8 (vector) registers in the CPU. A key figure of merit is therefore the ratio of

main memory bandwidth to peak processing power. The Cray-1 achieved half its peak floating point performance on all our hydrodynamic codes. This was possible because its 8 vector registers made greater memory bandwidth unnecessary. If we think of the local on-chip stores of the SPU, CPU core, and the SM as providing space for vector registers, then all these vector engines have much higher register capacities than the Cray-1. In order to approach Cray-1 levels of performance (as fractions of peak) on these devices, programmers must therefore find ways to exploit this on-chip storage. If the SM in GPUs had as much on-chip memory relative to its natural vector length as the SPU, we should be able to extend our Cell programs to the GPU with relatively little effort. This conclusion follows from the two devices having nearly the same memory bandwidth to peak performance ratios. However, the SM has very much less storage space on chip than the SPU. We must therefore develop a different programming strategy for the GPU that can address this on-chip memory deficiency.

Exploiting the SIMD engines and the on-chip memory is the key to high performance computation on the Cell processor. We have argued above that the same considerations are key to achieving high performance on modern processors. For efficient usage of these hardware features, several performance challenges have to be tackled. These challenges include SIMD engine programming, on-chip memory utilization, and the programming model.

1. **Programming for the SIMD engine:** On the Cell SPU, all floating point arithmetic is done in the SIMD mode, which is a performance benefit. Cell programs must use SIMD intrinsic functions, which are C language extensions, to perform the SIMD operations. Our vectorization framework therefore is built to perform vectorization for the Cell SPUs. The same framework is able to perform vectorization on the latest SIMD engines. The briquette data structure naturally exposes long vector lengths for GPU computation. GPUs adopt a single instruction multiple thread (SIMT) programming model, similar to SIMD in CPUs, and rely on new language extensions to execute the parallelization. With sufficient vector length in computation, programming efforts for GPUs can be minimized.

2. **On-chip memory optimization:** Our approach for the Cell processor is to keep all the operands in its local store via aggressive memory footprint reductions. Multicore CPUs implicitly manage their multiple-level cache memories, but low performance results unless the program is very carefully written. We use the same strategy, therefore, for both the Cell processor and for a multicore CPU, keeping all the operands in the cache memory. We find that this approach delivers very high performance on both platforms. We minimize the on-chip workspace requirement by using very short, but aligned, vectors and by an extremely aggressive pipelining transformation of the code that has been described in chapter 5. Today's GPUs require vector loop lengths of 64 words, and therefore we run our vector loops for these target devices over entire grid briquettes of 64 cells. Each vector loop for the GPU runs over 4 grid planes, and is aligned on grid planes, but these 4 planes may not all reside in the same grid briquette. The element of processing for a single thread block (a group of 64 GPU threads) of control is therefore either one briquette grid plane or one entire briquette's worth of such grid planes. We keep in the on-chip memory only the minimum amount of live data needed to accomplish the complete update of this amount of information. To do this, and to get the benefit of the reuse of all the intermediate quantities that we must compute in order to accomplish this update, we aggressively pipeline the entire update computation. This update then occurs in a single, extremely large outer loop containing a large number of inner loops over 16 or 64 elements.

3. **Data fetching and writing:** Fetching new data into the on-chip memory and writing the updated data back to main memory makes up the entire memory traffic for the code implementation. To assure that this data traffic can be overlapped with computation, we need to prefetch data in advance. This is easily done on the Cell processor, but on CPUs today, we can only give hints to the vendor compilers about this prefetching. On future hardware, we expect that prefetching will become sufficiently important for performance that intrinsic function calls will be provided to enable us to control this essential part of the algorithm, as we do

today on Cell and also on GPUs.

4. **Handling Different Programming Models and Languages:** The Cell processor not only brought innovation in hardware design, but also in its programming model. Cell Programmers must offload their computationally intensive code onto the SPUs. Explicit control of the memory hierarchy must be applied to move data through the DMA channel and to manage the on-chip memory footprint. Intrinsic functions are also required for SIMD execution. Multiple levels of parallelism can be exploited concurrently, including data parallelism on the SIMD engine, thread parallelism on the SPUs, and task parallelism on multiple PPUs and host processor cores. Finally, message relaying has to be applied in the cross-node MPI implementation. Much research has addressed these programming hurdles. We encounter all of these issues to one extent or another in targeting today's computing platforms. Our strategy relies on our source-to-source transformation framework to automate the most tedious and difficult aspects and to produce output code for all targets from a single input source.

   On the CPU platform, the programming model is well known. The CPU's support for multiple languages gives programmers flexibility in implementing their applications. However, tuning the performance for the CPU platform is not trivial. Programmers must take into account hardware features and compilation details.

   The CUDA SDK has been developed for programming the Nvidia GPU. The C language for CUDA includes extensions to support special GPU hardware features. Programmers must manage the GPU thread hierarchy, manipulate the varieties of memory in the memory hierarchy, manage data movement, and handle synchronization with this new programming language.

In summary, the code implementation for these architectures follows these steps: (1) we prefetch a small block of data from main memory to on-chip memory, (2) we unpack the data to produce a number of short vectors that reside in the on-chip memory, (3) we update the data via many vector floating point operations, (4) we repack the updated data into a new data record, and (5) we write the new record back to main memory.

Achieving uniformly high performance on these platforms is difficult. Nevertheless, we find that a single approach, with some variations, gives good results on all these different systems. Performance portability is possible through the same approach.

8.2 Performance study

We achieve a performance of 7.7 Gflop/s/SPU, 33% of the 32-bit peak performance, running our PPM code on the Cell processor. When we add in the data movement between main memory and on-chip local store that we need to run a sizable problem, the performance falls by about 16% (to 6.3 Gflop/s/SPU.) When we change the algorithm to a multifluid one not studied in this performance comparison, for which the SPU demands 3 times the data from main memory but performs only 50% more work, the performance falls another 29% (to 4.5 Gflop/s/SPU.) Using multiple Cell processors in the large IBM Roadrunner system, the delivered performance is between 3.4 Gflop/s/SPU (with thousands of Cell processors) and 3.9 Gflop/s/SPU (with hundreds of Cell processors.) In this chapter, we present an extended performance study of the Intel and IBM multi-core processors and of the Nvidia GPU.

The CPU study is implemented on an Intel cluster equipped with dual quad-core Intel Xeon x5570 and 24 GB main memory in each node. This quad-core processor runs at 2.93 GHz clock speed and the peak performance is 93.8 Gflop/s/processor. The compilers for the Intel device are the Intel Fortran and C compilers in version 12. The second CPU platform is an IBM workstation equipped with four 8-core Power-7 processors. This Power-7 processor runs at 3.86 GHz with a 247 Gflop/s/processor peak performance. The compilers for this Power-7 machine are the XL Fortran compiler, version 13.1, and the XL C compiler, version 11.1. The GPU study is implemented on an Intel workstation equipped with a quad-core Intel i7-950 and 12 GB main memory. The GPU is the Nvidia C2070 with ECC turned off. The software package in this workstation has CUDA SDK 4.0 and GCC and the GNU Fortran compiler in version 4.3.4.

Each CFD code counts the flops it performs, using manual counts for each vector

loop that are coded into the algorithm. Three flops are assigned to each reciprocal evaluation and none to each vector logical operation (or CVMGM.) For the GPU, each GPU thread counts its own flops. To sum up the overall flops number from hundreds of GPU threads causes high overhead in the execution time. Therefore, we take the flop count from CPU to evaluate the performance on GPU. The flop count from GPU is used to verify the counting correctness. The percentage difference in flop counts between GPU and CPU is within only a few percent.

8.2.1 Multi-core CPUs

PPM advection experiment result: The first performance result is from the PPM advection code. To start the performance study, we first take the code with no optimizations applied. The computation is expressed in a sequence of triply nested, vectorizable loops extending over a regular 3-D Cartesian grid. An outer code performs this computation on a sequence of grid briquettes of the whole, with the number of grid cells, nsugar, on each side of the cube given as a parameter. We measure the performance using nsugar = 64 and 16. The computation is parallelized with 8 OpenMP threads in a single computational node. Running these 8 threads exacerbates the memory bandwidth limitation, which is most strongly felt for the case of nsugar = 64. For nsugar = 16, each grid briquette fits entirely into the cache memory, so we achieve a vectorized and cache-blocked implementation. In measuring the performance of this implementation, we have not given credit for the redundant computation performed in updating one grid briquette that is performed once more in updating the next briquette in the processing sequence. This redundant computation is a valid cost of this style of cache-blocked computation, and we account for it in this way. The leftmost group in Fig. 8.1 shows the result for nsugar = 64, we obtain more or less the performance we would expect if we were to assign each briquette, with no cache blocking, to a separate MPI process, with MPI message passing occurring through on-node data copies in the shared memory of a single workstation. In this implementation we perform less redundant computation from one grid briquette update to the next, but the entire update no longer fits into the on-chip cache, so that the delivered performance is lower. Both Intel Nehalem CPU and

IBM Power-7 processor, having the same on-chip memory size, show low performance from the diagram.

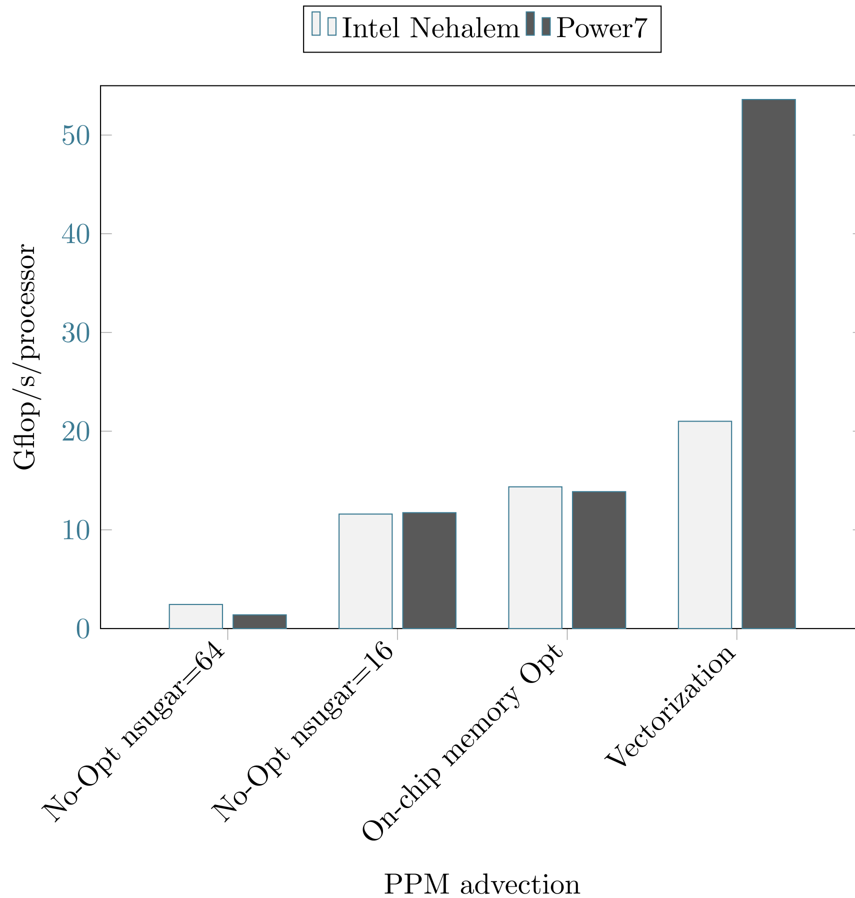

Figure 8.1: PPM advection performance on CPUs

Next, we perform memory optimization by a combination of manual and automated processes. We process briquettes (of size of 4×4×4) in sequence and we pipeline the processing completely. During the time that we update a grid briquette entirely using cache resident data, we are pre-fetching the next briquette in the sequence. We also utilize partial results of one briquette update to perform the next briquette update, so that there is no redundant computation between the two updates. For this code expression, we find the greatest delivered performance from running 16 simultaneous

threads on a dual-quadcore-CPU workstation. Comparing the result from the 2nd and 3rd groups in Fig. 8.1, about 25% of the performance improvement is delivered by the memory optimization on both Intel and IBM processors.

The last case that we study here is the result from our vectorization framework applied to this second, transformed and pipelined Fortran code. The rightmost group in Fig. 8.1 shows the transformed codes delivering the best percentage, 22.4%, of 32-bit peak performance on the Intel CPU and 21.7% on the Power-7. The same SIMD code used on the Intel processor can be compiled with the GCC compiler and result in a similar performance. From this study on CPUs, we see that devices from both vendors deliver similar percentages of peak performance after our code transformations, except that of course Power-7 has twice as many cores. However, the Power-7 shows worse single core performance before the vectorization transformation is applied. The conditional vector merge functions can be treated as branches and jumps in a vectorizable loop. These branches restrict most compilers from performing vectorization. The Intel compiler has an aggressive approach to partially vectorize and leads to the performance difference between Intel and IBM devices in Fig. 8.1.

PPM single-fluid experiment result: The second performance test is using the PPM single-fluid code. We start the experiment with a grid of 128^3 cells. The computation is parallelized with 8 OpenMP threads and a brick of 64^3 cells is assigned to an individual thread. This case corresponds to our earlier experiment with the PPM advection code using nsugar=64. The performance is limited by the available memory bandwidth and only 2.8% (Power-7) and 3.8% (Intel) of the 32-bit peak performance is achieved. After the on-chip memory optimization and the computation pipelining, the optimized code has 3× (Power-7) to 4× (Intel) speedups in performance. The center group in Fig. 8.2 shows the result for both CPUs. From the rightmost group in Fig. 8.2, the SIMD codes generated from our transformation tool gain another 2× speedup to deliver the 27.8 Gflop/s (30% of 32-bit peak performance) on the Intel CPU and 54 Gflop/s (22%) on the Power-7. The achieved performance from this code on both devices is the best in our performance study. The PPM single-fluid code has 6 physical variables as input and another 6 variables for output. It needs 24.75 KB of on-chip memory space

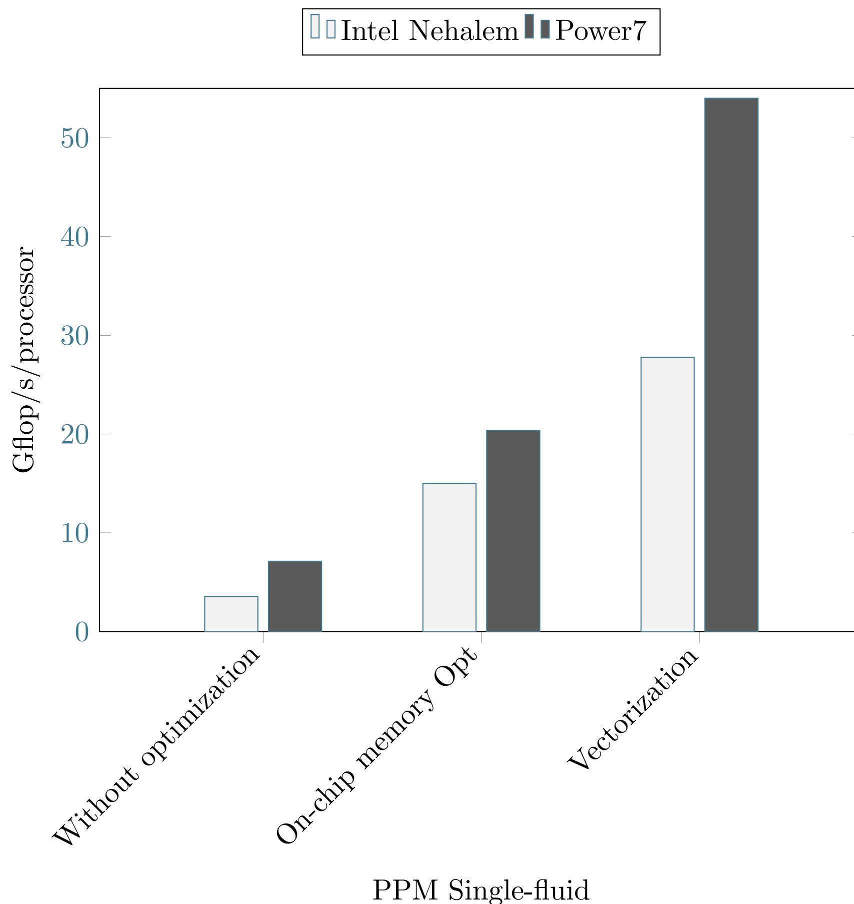

Figure 8.2: PPM single-fluid performance on CPUs

for each thread to store the intermediate results. The high computational intensity of 68 flops per word pushes the computational performance to this very high value.

8.2.2 GPGPUs

PPM advection experiment result: To implement the advection code for the GPU, we take the code with all optimizations and transliterate it into a CUDA code. We set up the briquette size to be 4^3 cells and use 64 GPU threads, in a thread block, to update a briquette pencil. Each temporary array needs a memory size of 8 grid planes, or 0.5 KB, because the size of the difference stencil is 4. The advection code needs

more than 20 temporary arrays and will generate 13 KB of shared memory usage for a thread block. CUDA would allow only 3 thread blocks in a single SM. Therefore, the delivered performance is only 33 Gflop/s on a Fermi GPU (leftmost histogram in Fig. 8.3). This result includes an overhead to copy the visualization data to the CPU memory. Excluding this data transfer to evaluate the performance, 35 Gflop/s is achieved.

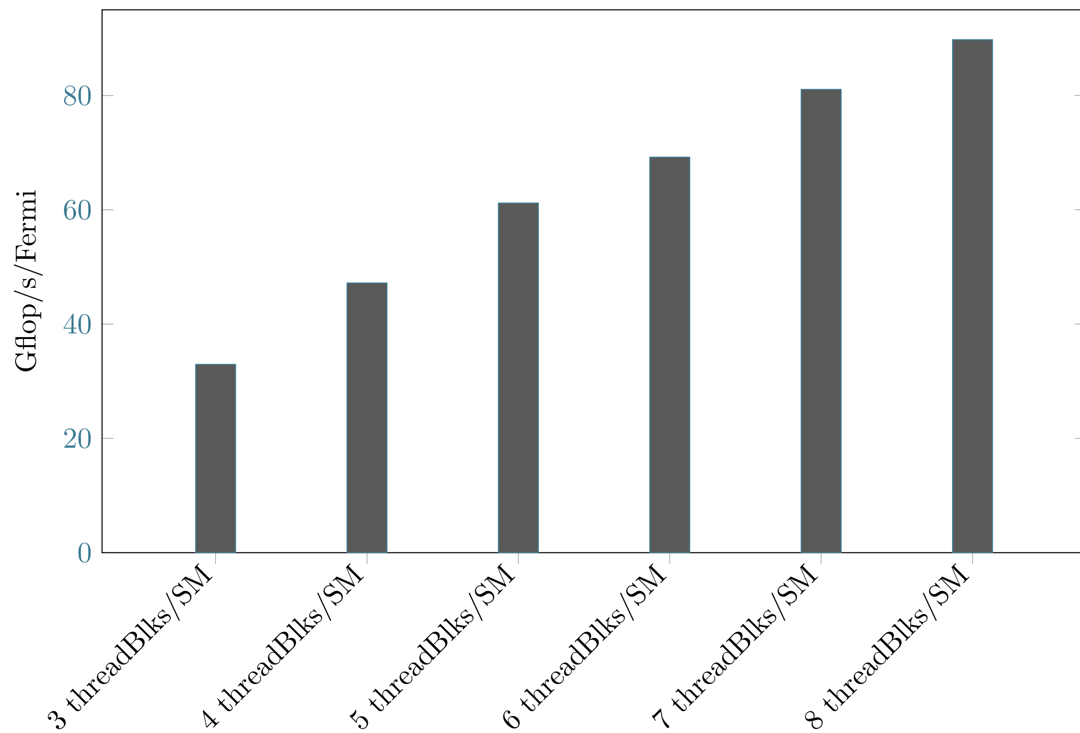

Figure 8.3: PPM advection performance on GPUs

We have to apply a different memory footprint reduction to reduce the shared memory usage in a thread block. In this memory reduction process, temporary variables that can be re-computed on the fly will be eliminated from the shared memory space. In the CPU computation, the larger cache memory space grants a programmer the flexibility to store more intermediate results. These cache-resident results can be reused with high computation efficiency. For a device with smaller on-chip memory, like the GPU, re-computing the temporary variables to save the on-chip memory space could lead to

a better result in performance. However, identifying these variables for the memory reduction is not trivial in a CFD application. Through iterations of memory reduction, the minimal size required in shared memory shrinks to only 5.5 KB in one thread block. The occupancy climbs up to 33% and the delivered performance becomes 90 Gflop/s, which is 8.7% of the 32-bit peak performance. Fig. 8.3 shows the performance results from a Fermi GPU. The histograms from left to right show the performances with different occupancy. The PPM advection code has only one input variable and one output variable. With all the temporary results stored in the on-chip memory, memory traffic does not saturate the available memory bandwidth. If we use the "-fast-math" compiler option, a performance of 114 Gflop/s is reported. However, this option uses less accurate mathematical functions and can lead to incorrect results in the numerical computation.

PPM single-fluid experiment result: To implement the PPM single-fluid code for the GPU, the 24.75 KB required on-chip memory space for a CPU thread exposes the challenge immediately. To achieve better performance for the GPU, additional optimizations are applied to efficiently exploit the shared memory. PPM applications use coupled equations in the algorithm design. The same amount of input can generate much more temporary results and store them in the on-chip memory. These cache-resident results can be referenced for the later computations in a most efficient way. There are only a limited number of temporary results that can be stored in the shared memory. To exploit the shared memory for the PPM single-fluid code, we minimize the number of temporary results by de-coupling the equations and dividing the program into several episodes of computation. Each episode is like a subroutine and is called from a main computational kernel. In the GPU implementation, we only generate a few GPU kernels, and each of them will perform a series of episodes of computation. This will effectively eliminate the overhead caused by kernel launching and terminating. A single episode walks through the grid pencil to generate results used in the later episodes, or the final results. The inputs and outputs in an episode will have the size of a full grid pencil and be permanently stored in the global memory. Each thread block reserves a private space in the global memory to store a group of pencil-sized arrays. We allocate

112 memory spaces for the maximum of 112 thread blocks (8 thread blocks/SM $\times$ 14 SMs/Fermi) that can execute concurrently. A thread block can reuse the same group of pencil-size arrays to perform multiple pencil sweeps. Intermediate results generated within an episode are cached in the shared memory. As shown in Fig. 8.4, an episode takes input from the pencil-sized array and fetches data into a group of registers. Once the data is used in computation, it is copied into the shared memory. A thread block, 64 threads, updates a 4^3 briquette and writes updated data back to the global memory. The limited on-chip memory size restricts the number of input and output variables for each individual episode. We then apply the optimizations, used in the CPU study, to all the episodes to maximize the computational intensity. Only after this additional work, the shared memory is used to host the temporary arrays and used for the data prefetching as we discussed earlier. The achieved performance is 33.4 Gflop/s. This result includes the same overhead as the PPM advection code. 57.1 Gflop/s is achieved by eliminating this data transfer. The right-hand group in Fig. 8.5 shows these result with the optimizations.

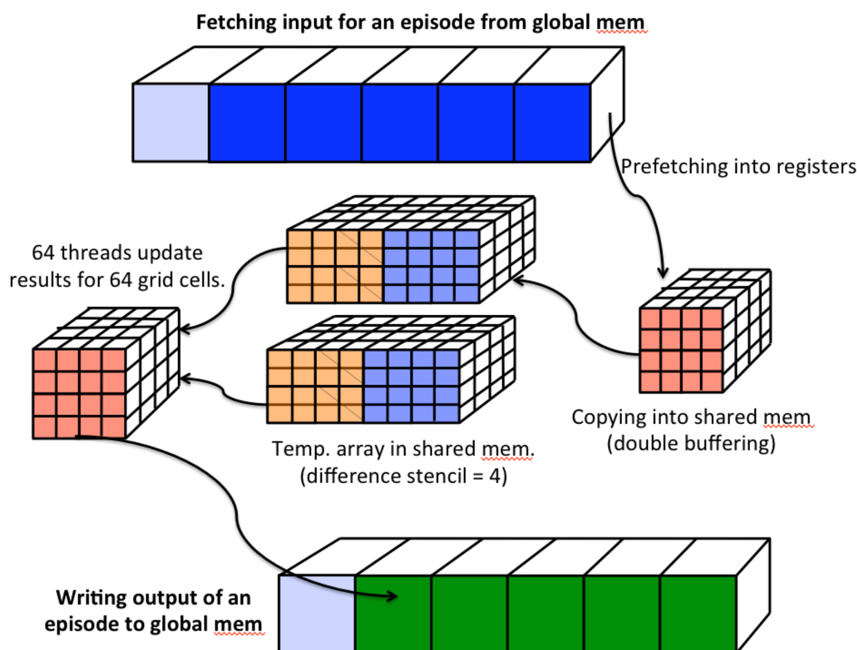

Figure 8.4: illustration of the GPU implementation

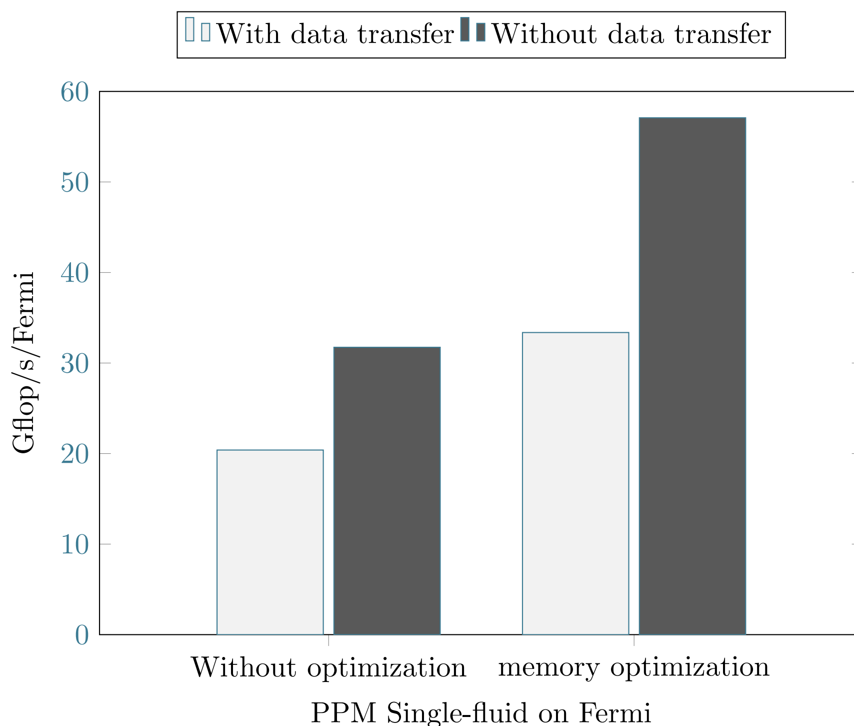

Figure 8.5: PPM single-fluid performance on GPU

To demonstrate the usefulness of our optimizations on the GPU, we transliterate the single-fluid code without any source code optimization into CUDA. This keeps all the temporary results in the global memory and has no shared memory usage. We also configure the on-chip memory to have only 16 KB shared memory and 48 KB L1 cache. Fig. 8.5 shows the comparison between this approach and the optimized result. The delivered memory throughput is close to the available memory bandwidth. The resulting performance is 20 Gflop/s (with data transfer) and 31.7 Gflop/s (without data transfer).

8.3 Characterizing the multifluid gas dynamics application code

The grid cell update for a single 1-D pass is characterized quantitatively in Fig. 8.6. In the first column, descriptive names of individual routines, or kernels, are listed. It is natural to encapsulate each of these kernels as a subroutine. The table in Fig. 8.6 characterizes each "kernel" into which we have decomposed our entire multifluid gas dynamics computation. For each, the number of flops per grid cell updated is given, and this is also broken down into which kind of flops these are. It is assumed that each divide is actually a reciprocal evaluation followed by a multiply. Also the implied subtracts in certain logical comparisons are counted. Reciprocals are counted as 3 flops, as they were on the Cray-1. Reciprocals can be performed this fast on no hardware in existence today. The actual cost of a single-precision reciprocal is about 14 flops. This fact, as well as the fact that we give no credit in flops for vectorizable logic operations, explains the differences in measured performance on the various kernels in our code. For example, the kernel $ppb10constrainx$ is extremely logically intensive and it runs at roughly half the performance of most of the other kernels. But this logic is essential to the value of the PPB algorithm, as it permits multifluid interfaces to be described on the grid as nearly one grid cell across, with very few associated numerical artifacts. Vectorizable logic operations are listed in this table under the column "cvmgms," the name for these functions in classic Cray programs. They are vector mask operations, taking two vectors as inputs and using the sign of a third vector to select from one or the other of the inputs to form the output vector.

The column labeled "Intel Westmere Gflop/s/core" in Fig. 8.6, with its entries given in bold, contains derived performance measurements for the individual code kernels. These were measured by executing the kernel the number of times listed in the column "N X this routine" while all the other kernels in the sequence were executed only once. The aggregate flops per cell that resulted and the elapsed seconds per timing interval are listed at the right in Fig. 8.6. From this information, the flops per grid cell for one execution of the kernel, given in the left-most column, and from the data in the

next-to-last row for a single iteration of all the kernels in sequence, we may derive the individual kernel performance given in bold. This is a measurement of how fast the CPU core can perform the kernel working with all necessary data already in its L-1 cache and all instructions in its instruction cache. In this sense, these values in bold are "never to be exceeded" numbers. The last bold number in this column then gives the performance we could achieve for this code if every kernel could run at this never-to-be-exceeded rate. This comes out to be 7.40 Gflop/s/core, or 44.4 Gflop/s (7.4×6) for a single Westmere CPU. The actual performance appears in the next column to the right, namely 4.33 Gflop/s/core. Below it, at 6.22 Gflop/s/core, we have the performance we could achieve on this code if there is no cost for fetching and writing back data to main memory. This number is achievable if an optimal prefetching scheme can be implemented.

The columns in Fig. 8.6 labeled "input temps," "output temps," "scratch," and "intensity" characterize the data I/O properties of the various kernels. None of the computational kernels read or write data from the off-chip memory in the implementation for a CPU. The kernel "fetch DD" performs the data prefetch mentioned above. The necessary data is packed into grid briquette records in main memory. Each such record contains all we need to know about a grid briquette of 4^3 cells. This can be read or written in a single sequential data transfer, which we express in standard Fortran as an array copy as discussed above. The algorithm requires that we read in an entire briquette record for each briquette we will update, as well as the first third of 4 neighbor briquettes that are adjacent to it in the two transverse grid dimensions. The unit of data in Fig. 8.6 is a single grid plane of 4×4 cells, but because we fetch entire briquette records once every 4 main loop traversals (executing all the computational kernels in sequence on each traversal), we have divided the size of the data transfers in the "fetch DD" and the "storeDDnu" kernels (not shown in the table) by 4 to match the other data entries in the table. StoreDDnu simply transposes a single briquette record (with 15 variables) and writes it to main memory. The last 3 columns in Fig. 8.6 give the numbers of single grid plane vectors, of length 16 words each, that serve as input, output, and scratch work space for each kernel. From these and the number of flops performed in the kernel, a computational intensity is computed in flops per input plus output word.

For the GPU, these "temps" are placed in the global memory of the device, but for the CPU, they are all resident in the L-2 cache. Several of the kernels inevitably have low computational intensities and would have high costs in efficiency using the GPU design. As noted earlier, the computational intensity of the entire algorithm is very high. This is given on the bottom row, labeled "TOTAL," in Fig. 8.6.

The items in Fig. 8.6 and Fig. 8.7 listed at the left as kernels are not Nvidia kernels. For an Nvidia kernel, a global barrier is required upon completion, followed by a very substantial startup cost for the next kernel. For OpenMP programmers, this is like ending and reestablishing a parallel region with performance overhead involved. With the implementation executing on the Fermi GPU, doing it is devastating to performance. Instead, the kernels in the tables here are identifiable episodes of computation with explicit inputs and outputs that must in one way or another be cached. ($Interprhopux$ is actually implemented as 3 such kernels.) The entire code, as it executes on the GPU for a complete update of all the grid cells, is implemented as a single Nvidia kernel.

Our kernels require caching of outputs, which will serve as inputs for later kernels. There is essentially no place to cache these inputs and outputs on the GPU chip. We therefore are forced to cache them in global memory. To make this caching scheme efficient, we must execute each kernel not just on a single grid briquette, or on a single grid plane of 16 cells as we do on the CPU, but on an entire grid pencil of many briquettes arranged in a line. This strategy allows us to prefetch our inputs one briquette at a time and to write back the outputs continuously with this same granularity of 64 words per CPU thread or GPU thread block. We have been able to prefetch no more than 15 inputs per kernel. To prefetch the inputs, we need to place in the CUDA code an assignment of a variable on the stack (a "register" variable) to a briquette of the input data (in CUDA this looks like assigning one word on the stack to one word in global memory). As long as this variable is not used, the SIMD engine will not wait for this data transfer to be complete until we assign this register variable to a variable in the programmer-managed, on-chip "shared" memory. When this assignment occurs in the program, the hardware will wait on the completion of the data transfer.

The proposed strategy for executing one of the kernels, or code sections, on a GPU

kernel	flops per grid cell	adds + mults	recips	sqrts + exps	cvmgms	input temps	output temps	scratch	intensity	Intel Westmere Gflop/s/core	West-mere speed w N x this	N X this routine	aggre-gate flops/cell	elaspsed secs	implied nanosecs per kernel
fetch DD to d	0	0	0	0	0	0	35	0	0.00						
unpack d	0	0	0	0	0	25	37	0	0.00						
difuze1	20	17	1	0	13	14	2	10	1.25	**4.29**	4.30	1000	71928	187.28	4.66
difuze2	203	123	8	4	44	25	2	30	7.52	**7.54**	7.13	100	69432	108.89	26.93
fvscses	84	55	5	1	5	7	23	1	2.80	**10.42**	9.44	250	71237	84.44	8.06
interpriemann	25	22	1	0	0	10	10	0	1.25	**7.14**	6.80	800	59702	98.27	3.50
ppmdntrf0vec	32	32	0	0	10	3	3	13	5.33	**7.57**	7.16	640	66072	103.14	4.23
ppmdntrf0vec	32	32	0	0	10	3	3	13	5.33	**7.57**	7.16	640	66072	103.14	4.23
ppmdntrfavec	68	62	2	0	24	5	3	11	8.50	**6.46**	6.34	640	142093	250.63	10.53
ppmdntrfavec	68	62	2	0	24	5	3	11	8.50	**6.46**	6.34	640	142093	250.63	10.53
ppmintrf0vec	34	34	0	0	11	3	5	12	4.25	**6.50**	6.27	640	73710	131.41	5.23
ppmintrf0vec	34	34	0	0	11	3	5	12	4.25	**6.50**	6.27	640	73710	131.41	5.23
interprhopux	160	160	0	0	36	18	16	4	4.71	**7.33**	6.87	100	54805	89.22	21.84
riemannstates0+1	174	108	4	0	60	13	8	12	8.29	**4.89**	4.86	170	53524	123.23	35.57
riemann0+1	110	61	7	2	9	18	2	2	5.50	**15.82**	13.31	200	73913	62.13	6.95
ppb10constrainx	78	69	3	0	50	3	3	12	13.00	**4.06**	4.08	250	63604	174.53	19.23
ppb10fv	105	102	1	0	14	4	18	5	4.77	**11.30**	10.10	200	69342	76.82	9.29
ppb10fvy+fvz	146	146	0	0	37					**8.11**	7.63	150	58037	85.02	18.00
fluxes	106	94	4	0	22	31	13		2.41	**12.18**	10.75	200	74711	77.75	8.70
cellupdate	126	102	8	0	11	28	23	5	2.47	**13.26**	11.74	200	80242	76.45	9.50
diffusion	124	106	6	0	7	25	23	2	2.58	**12.37**	10.87	170	68064	70.02	10.03
cour	1	1	0	0	1	1	1	0	0.50	**0.24**	0.26	20000	245046	219.06	4.18
Vis Subtotal	629	284	17	21	33	32	6	11	16.55						
TOTAL	1676	1422	52	7	399	341	272	183	33.52	**7.40**	**4.33**	1	5001	425.11	
											6.22	100	499267	3029.29	

Figure 8.6: Characterization of the multifluid PPM gas dynamics code and its performance on a multicore CPU. Speeds are quoted in Gflop/s/core, measured as the code is executed in parallel by all 6 cores of the Westmere CPU.

is therefore clear. We carefully design the kernel to have fewer inputs and outputs, to do as much computational labor as possible, but without using more than a handful of intermediate results that cannot be instantly consumed. Then we proceed down a single briquette pencil (for Fig. 8.7, it is $4 \times 4 \times 128$ cells) processing one grid briquette of 4^3 cells at a time. At the beginning of our loop over grid briquettes, we prefetch the next input briquettes and copy over the previous ones into the on-chip shared memory. Then we execute the kernel code, which we pipeline as we do for the CPU in order to produce the greatest possible reuse of on-chip data. It is important to note that the data prefetch uses twice as much on-chip data storage as we would need without prefetching. However, for the kernels in Fig. 8.7, this prefetching delivers more than a doubling of the performance, even though it forces us to have only 2 threads per SIMD engine running simultaneously rather than 4. Because the delivered performance increases nearly linearly with the number of simultaneous threads, for the same code implementation, it should be evident that if the on-chip data storage capacity is doubled without also doubling the number or width of the on-chip SIMD engines, the code's performance would also double.

In Fig. 8.7 we report results for our entire multifluid gas dynamics code. We have applied the strategy just described, but this is only a first effort. From the individual speeds of our kernels in Fig. 8.7, it is clear that the final two kernels, in which we have not yet implemented our prefetching strategy, pull down the overall measured performance substantially. The only other examples of relatively slow kernels involve several inputs and outputs but very little computation. With careful rearrangement and more thought about which temporaries can be recomputed, the impact of these kernels should be diminished. Measuring the overall performance of the code not including these last two unfinished kernels results in an overall performance of 14.93 Gflop/s, as shown in Fig. 8.7 in bold, rather than of 14.1 Gflop/s including them. The time-weighted average of all but the last 2 kernels in Fig. 8.7 is 22.45 Gflop/s. This is the same phenomenon that we saw from the analysis of Fig. 8.6 data for the CPU. The hardware prefers to do a tiny portion of the job over and over rather than to execute a complete stream of instructions.

The set of kernels listed in Fig. 8.7 include the 6 kernel executions all of which are variations on the kernel in the PPM advection code example discussed earlier. We therefore had already devoted considerable effort to implementing them on the GPU. However, the performance numbers in Fig. 8.7 are roughly a third of the levels reported for the advection code. In that code, we were able to package the advection calculation so that it fetched only one data briquette and wrote only one data briquette for every grid briquette it updated. This produces an artificially high computational intensity for the advection kernel, although it does indeed match fairly closely the intensity of the entire multifluid gas dynamics computation for a CPU with a cache. We see in Fig. 8.7 six such kernels listed with computational intensities that are lower than for the advection code by factors of 3 or 4. This results from their use within a code that is solving coupled partial differential equations. To handle the coupling of the equations, we are forced to have each PPM interpolation kernel produce as output the 3 coefficients of its interpolation parabola for use later in the code. Additionally, there is cross coupling between the interpolations themselves that is expressed through input or output flag vectors that mark cells in which special constraint operations must be applied. The coupling of the equations is also felt in the first and last kernels on the list in Fig. 8.7. The first kernel ingests the fundamental fluid state variables together with sound speed and other values that have been computed earlier. From these it constructs values of the Riemann invariants, which are the signals that are transported through the fluid along sound wave paths and along streamlines. This is why the first kernel on the list has so many inputs and outputs. After interpolating parabolae for the Riemann invariant signals, in the final kernel in Fig. 8.7 we put these quantities back together to produce consistent, constrained interpolation parabolae for the densities, pressure, and velocity components. Again we have a huge number of inputs and outputs, but this cannot be avoided.

kernel	flops	adds	mults	recips	sqrts	exps	cvmgms	input globals	output globals	input temps	output temps	scratch	intensity	Fermi GPU total speed (Gflop/s)	GPU elapsed secs	Global memory read/write throughput (GB/s)	GPU iteration
Diversify	0	0	0	0	0	0	0	35	0	25	37	0	0.00	0.00	1.16		1
difuze1	20	16	1	1	0	0	13	0	0	14	2	10	1.25				
difuze2	203	56	67	8	4	0	44	0	0	25	2	30	7.52				
difuze1+difuze2	223	72	68	9	4	0	57	0	0	25	2	30	8.26	42.41	10.66	21.25	1000
vortdivu	35	16	5	0	1	0	0	0	0	15	2	0	2.06				
fvscses1	36	10	20	2	0	0	2	0	0	6	11	1	2.12	17.74	4.41	32.31	1000
fvscses2	48	9	16	3	1	0	3	0	0	7	15	1	2.18	20.61	5.51	33.50	1000
fvscses1+2	84	19	36	5	1	0	5	0	0	7	23	1	2.80	17.16	11.59	21.71	1000
interpriemann	25	14	8	1	0	0	0	0	0	10	14	0	1.04	7.16	8.25	24.42	1000
ppmdntrf0vec	32	16	16	0	0	0	10	0	0	3	2	13	6.40	28.88	2.27	18.50	1000
ppmdntrf0vec	32	16	16	0	0	0	10	0	0	3	2	13	6.40	28.88	2.27	18.50	1000
ppmdntrfavec	68	31	31	2	0	0	24	0	0	5	3	11	8.50	27.25	5.37	12.49	1000
ppmdntrfavec	68	31	31	2	0	0	24	0	0	5	3	11	8.50	27.25	5.37	12.49	1000
ppmintrf0vec	34	18	16	0	0	0	11	0	0	3	5	12	4.25	29.93	2.78	24.10	1000
ppmintrf0vec	34	18	16	0	0	0	11	0	0	3	5	12	4.25	29.93	2.78	24.10	1000
interprhopux	160	88	72	0	0	0	36	0	0	19	16	4	4.57	18.69	20.33	14.44	1000
riemannstates0	54	22	26	2	0	0	26	0	0	11	5	12	3.38	20.43	5.79	23.19	1000
riemannstates1	66	26	34	2	0	0	34	0	0	13	8	12	3.14	10.65	13.72	12.84	1000
riemann0	9	6	3	0	0	0	1	0	0	8	2	0	0.90	11.96	1.63	51.35	1000
riemannstates0+1	120	48	60	4	0	0	60	0	0	13	8	12	5.71	12.66	21.18	8.32	1000
riemann1	101	25	27	7	2	0	8	0	0	18	2	2	5.05	42.30	8.99	18.67	1000
diversifymom	0	0	0	0	0	0	0	0	0	9	9	0	0.00				
ppb10constrainx	78	41	28	3	0	0	50	0	0	3	3	12	13.00	43.92	3.88	12.97	1000
ppb10fv	105	42	60	1	0	0	14	0	0	4	18	5	4.77	26.59	8.38	22.01	1000
ppb10fvy	88	33	55	0	0	0	22	0	0	20	4	8	3.67	34.77	5.27	38.19	1000
ppb10fvz	58	23	35	0	0	0	15	0	0	20	3	7	2.52	23.46	5.16	37.41	1000
ppb10fvy+fvz	146	56	90	0	0	0	37										
fluxesrhop	78	23	43	4	0	0	16	0	0	20	7	8	2.89	11.38	14.91	15.19	1000
fluxesuten	28	11	17	0	0	0	6	0	0	16	6	5	1.27	9.14	6.50	28.39	1000
fluxes (both)	106	34	60	4	0	0	22			31	13		2.41	9.84	23.56	15.67	1000
cellupdate	126	49	53	8	0	0	11	0	0	28	23	5	2.47	10.90	25.49	16.78	1000
diffusion	124	45	61	6	0	0	7	0	0	25	23	2	2.58	7.50	35.46	11.35	1000
Store Ddnu	0	0	0	0	0	0	0	0	15	15	0	0	0.00				
TOTAL	1425	575	638	38	7	0	380	35	0	256	182	179	40.71	14.93	223.55	16.44	1000

Figure 8.7: Characterization of a portion of the PPM gas dynamics code and its performance on a many-core GPU.

Chapter 9

Study With Latest Compiler Technique

Computation in scientific applications using structured grids can be expressed in a sequence of nested loops extending over a regular multidimensional grid. A majority of research efforts to optimize these applications focuses on loop optimizations. The research focuses span memory locality enhancement, parallelization, and vectorization. Loop optimizations would be beneficial to scientific applications with structured grids. Based on our research experience, existing loop optimizations in present research and commercial compilers face challenges to deliver comparable performance to our manual optimization strategies. The reasons include limited analytical capability, lack of knowledge in application design, and engineering restrictions in compiler implementation. As a result, all the optimizations in this dissertation are implemented in our source-to-source framework and heavily rely on optimization guidance from domain experts.

In this chapter, we seek a strategy through the use of a state-of-the-art loop optimization framework to reproduce our optimization strategies and see if we can deliver comparable performance. The difficulties encountered in the process are presented and feasible solutions are discussed. This study has two major contributions: providing enhancement to domain-specific program optimizations and showing the potential of implementing our optimization strategies in existing compiler frameworks.

9.1 Polyhedral framework

The polyhedral framework is chosen as the state-of-the-art loop optimization framework in this study. The polyhedral framework is a mathematical framework based on an algebraic representation of programs. It allows a compiler to construct and search for complex sequences of optimizations. With its strength in exact data dependence analysis and powerful loop transformation capability, polyhedral transformation has been adopted by many research compilers for automatic optimization and parallelization.

9.1.1 Methodology

A set of consecutive statements, of which all loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters are defined as a static control parts (SCoP). The loops are represented as a set of inequities that defines a polyhedron. The term polyhedron will be used to denote a set of points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities: $\mathbb{D} = \{x | x \in \mathbb{Z}^n, Ax + a \geq 0\}$, where x is the iteration vector (the vector of the loop counter values), A is a constant matrix and a is a constant vector, possibly parametric. The iteration domain is a subset of the full possible iteration space $\mathbb{D} \subseteq \mathbb{Z}^n$. Due to the rigid mathematical representation, there are restrictions to form a SCoP, and many factors can easily break the SCoP properties. As polyhedral transformation heavily relies on the SCoP structure, loop transformations are prohibited in non-SCoP loop structures. There are numerous research efforts to relax the restrictions and make the polyhedral model more applicable.

Code 9.1: Example for affine loop

```
for (i = 2; i <= N; i++)
  for (j = 2; j <= N; j++)
    A[i] = pi;
```

After identifying all SCoPs in an application, three special representations will be generated for each SCoP. A set of affine constraints defines the polyhedron and can be represented as an iteration domain. For example, the set of constraints for Code 9.1 is:

```
i - 2 >= 0
-i + N >= 0
j - 2 >= 0
-j + N >= 0
```

The constraints are represented by "domain matrix $\times$ iteration vector >= 0":

```
[  1  0  0 -2 ]   [ i ]    [ 0 ]
[ -1  0  1  0 ]   [ j ]    [ 0 ]
[  0  1  0 -2 ] * [ N ] >= [ 0 ]
[  0 -1  1  0 ]   [ 1 ]    [ 0 ]
```

The second representation uses scatter function to preserve the ordering information in the polyhedral model. The most common information is scheduling, which uses logical time stamps to keep a logical time that express at which time a statement instance has to be executed. The third representation uses access function to preserve the array access information. For instance, the access function for array access $A[3 \times i + j][j][2 \times i + N]$ will be written as $F_A(i, j) = (3 \times i + j, j, 2 \times i + N)$. We can translate it to the matrix form: F_A(iteration vector) = access matrix $\times$ iteration vector. The matrix will be written in the following way:

```
     [ i ]    [  3  1  0  0 ]   [ i ]
F_A([ j ]) = [  0  1  0  0 ] * [ j ]
     [ N ]    [  2  0  1  0 ]   [ N ]
     [ 1 ]                      [ 1 ]
```

With all the information ready in the polyhedral model, the transformation function, represented in matrix form, can trigger the loop transformations. The polyhedral transformation is represented as $f(\overrightarrow{i}) = T\overrightarrow{i} + \overrightarrow{j}$ or $\{\overrightarrow{i} \rightarrow \overrightarrow{x} | \overrightarrow{x} = T\overrightarrow{i} + \overrightarrow{j}\}$, where T is the transformation function. The matrix representation can merge multiple transformation functions into one single matrix representation. Therefore, the notorious phase ordering constraint in compiler optimizations can be eliminated in the polyhedral transformation.

9.1.2 Polyhedral implementation

Although there is strong mathematical theory behind this powerful framework, there are also many known restrictions. The most well-known restrictions are: (1) loop bounds and conditionals have to be affine; (2) pointer manipulation is not allowed in SCoP; and (3) functions or control flows are not allowed in SCoP. With numerous research efforts to enhance the polyhedral model [78], several constraints have been relaxed and polyhedral transformations become more applicable.

There are some existing implementations of polyhedral model in research compilers. The WRaP-IT tool (WHIRL Represented as Polyhedra Interface Tool) in Open64, Graphite (the Gimple Represented as Polyhedra) in GCC, and Polly in LLVM compiler are available. Commercial compilers like IBM XL compiler and R-Stream Compiler from Reservoir Labs also adopt the polyhedral model to perform optimizations. Several research projects take source-to-source approach and implement transformation tools with the polyhedral model. The most popular projects are PLUTO (an automatic parallelizer and locality optimizer for multicores), and the PolyOpt (a complete polyhedral framework built in the ROSE compiler). The Polyhedral Compiler Collection (POCC) collects the libraries and loop analyzers. Most research projects share these analyzers and libraries to implement the polyhedral model. We choose PolyOpt as the infrastructure to perform loop transformations. The PolyOpt development is based on the ROSE compiler framework and is in line with the source-to-source approach. A branch of PolyOpt with Fortran language support is the most ideal implementation of the polyhedral model for this dissertation.

9.2 Polyhedral transformation

To evaluate the transformation capability provided in the polyhedral model, one version of the PPM advection code with no optimizations is chosen as the input for polyhedral transformation. Appendix B lists the simplified code of this PPM advection application. We submit the example code to PolyOpt/Fortran and verify the following processes performed in the polyhedral model: SCoP recognition, loop fusion, loop tiling,

parallelization, and vectorization.

1. SCoP recognition: Identifying SCoPs is a critical process in the polyhedral model. We have found PolyOpt/Fortran fails to identify SCoPs in the PPM advection code for two main reasons. The first reason is the non-affine conditionals inside the nested loops, and the second one is the private scalar variables or arrays inside the loop structure.

   The non-affine conditionals in the PPM advection code are the conditional merge functions, or CVMGMs used by the Cray vector machine. They are represented as if statements in the Fortran language. These conditionals can be replaced by the conditional operator, represented as '?' symbol, in the C/C++ language, and they are allowed in a SCoP structure. To allow PolyOpt/Fortran to recognize the conditional merge functions in a SCoP, we implement the control predicate in the ROSE compiler to convert control dependences into data dependences. All if statements in nested loops are converted to control predicates prior to the polyhedral transformation. After the transformation, a post-process function restores all predicates back to if statements. The control predicate is also adopted by Benabderrahmane et al. to allow arbitrary conditionals in polyhedral model [78], and by Shih to perform vectorization for control flows [79].

   Scalar variables serve two main purposes in a loop structure: to store temporary values that are independent in all loop iterations and to keep results for reduction loops. A similar situation appears when an n dimensional array occurs in an m dimensional nested loop and $n < m$. These scenarios cause loop-carried dependences and present difficulties in the SCoP recognition. Scalar privatization can be applied to eliminate the loop-carried dependences. However, there is no array privatization supported in the current implementation. Therefore, array expansion is performed to arrays with smaller dimensionality prior to the polyhedral transformations.

   After applying control predicate and array expansion, PolyOpt/Fortran identifies one SCoP for each directional update in the PPM advection code. The SCoP

has 104 statements and 302 loops in it. A total of 628 data dependences are detected by the polyhedral model. The memory consumption for the Fourier-Motzkin elimination in polyhedral analysis exceeds the 16 GB system memory capacity. We are forced to remove few loops from the SCoP recognition to lower the overall memory consumption. A new SCoP having 97 statements, 289 loops, and 619 data dependences will consume roughly 13.5 GB memory in the overall polyhedral analysis.

2. Loop fusion: PolyOpt/Fortran has three loop fusion schemes provided by PLUTO optimization: nonfuse, smartfuse, and maxfuse. These schemes differ in the ways of cutting strongly connected components (SCC) in data dependence graph (DDG). The smartfuse is the default scheme that involves parallelization and loop tiling in the optimization decision[80, 81]. Unlike the fusion performed in our briquette-pencil pipelining, the fused code generated by polyhedral fusion consists of the maximized-fused loop body and several prologues (loops before main fused body) and epilogues (loops after main fused body). A group of *if* statements are inserted to specify the prologues, epilogues, and the main loop body. The complex fused loop structure can be simplified if all the loop bound parameters are replaced with constant numbers. The analysis and compilation time can also be reduced from several hours to just a few minutes.

3. Loop tiling: Polyhedral tiling is performed to enhance both parallelism and data locality. The matrix representations of the iteration domain, scatter function and access function provide the advantages to verify the legality of tiling and sophisticated loop tiling [80, 82]. A set of tiling hyperplanes divide the loop iteration space into tiles. A tiling scheme is legal if each tile can be executed atomically and the valid ordering of all tiles can be constructed. The "–polyopt-fixed-tiling" is chosen to perform loop tiling with a customized tile size. The polyhedral tiling is capable of tiling the loops in the example code with default tile size of 32×32 and several customized 2-dimensional tiles (from 2×2 tile size to 16×16 tile size). However, the polyhedral tiling fails in the 3-dimensional tile

with $4 \times 4 \times 4$ tile size. The polyhedral tiling will face challenges to transform loops incorporating the briquette data structure.

4. Auto parallelization: PolyOpt relies on loop tiling to perform coarse-grained parallelization. Loop tiling in the polyhedral model partitions iteration space into tiles that can be executed concurrently in multiple processing units. An optimal polyhedral model will provide a cost function to search for optimal tile size with minimum inter-processor communication. The numerical algorithm used in the PPM advection example contains no dependence across the transverse directions. Parallelization can naturally be applied to the transverse plane. After polyhedral tiling, the outermost tiled loop in the PPM advection code iterates through multiple tiles in one transverse direction. This same loop exposes the parallelism, and polyhedral parallelization automatically inserts OpenMP pragmas to perform thread-level parallelization.

5. Vectorization: Polyhedral tiling often generates complex tiled code for a native compiler to further analyze and then optimize. Therefore, vectorization is part of the post-process in the polyhedral transformation by moving the parallel loop within a tile's innermost loop and applying required directives to explicitly force vectorization. After the polyhedral vectorization, the innermost loops in the PPM advection code fulfill the criteria for data-level parallelism. However, memory accesses with large strides are found in the innermost loops. As discussed in Chapter 4, the strided memory access will not be ideal for vectorization and can easily introduce large TLB miss rates.

9.3 Comparison and Discussion

9.3.1 Experiment setting

The example code in this study contains three separate subroutines to perform computation in different directional updates. The subroutine for each directional update is formed by a series of 3-dimensional loops with the same memory access pattern. The

loop for the Z direction is always the outermost loop, and the loop for the X direction is the innermost one. With no data transposition performed between directional updates, the streaming memory access pattern has a unit stride in the X direction and the longest stride distance in the Z direction. The fixed stencil pattern uses neighboring array elements only in the sweeping direction. Computation can be parallelized freely in the transverse directions. Misaligned memory access prohibits vectorization in the sweep direction. Therefore, only updates in the Y and Z directions perform optimal vectorization for this CFD example.

This experiment choses the PPM advection code with a grid size of $64 \times 64 \times 64$. 4 boundary cells in all directions are required in the computation. Up to four OpenMP threads in a quad-core Intel Sandy Bridge CPU can run concurrently. Two different compilers are chosen in the study and the highest optimization options are assigned in the compiler commands. The experiment compares the execution time between two visualization outputs, which involves 128 cycles of 6 single-directional updates ($X \rightarrow Y \rightarrow Z \rightarrow Z \rightarrow Y \rightarrow X$). Using the transformation option "–polyopt-fixed-tiling" in PolyOpt is equivalent to the following transformation sequences:

- –polyopt-pluto-fuse-smartfuse: loop fusion with the smartfuse scheme
- –polyopt-pluto-tile: loop tiling with a selected tile size
- –polyopt-pluto-parallel: coarse-grained parallelization
- –polyopt-pluto-prevector: fine-grained parallelization
- –polyopt-generate-pragmas: code generation for directives

The transformation option "–polyopt-scalar-privatization" is also applied to privatize scalar variables in the loop structure. A fused, tiled, parallelized, and vectorized code is expected after the transformation. This study compares the non-optimized input code, the optimized code with the briquette-pencil pipelining strategy, and tiled codes with 6 different tile sizes.

9.3.2 Performance evaluation

Details of the briquette-pencil pipeline and its benefit for computational performance are discussed in Sec. 5. Fig. 9.1 and Fig. 9.2 both show a consistent performance improvement from this pipelining strategy on two different multi-core CPUs. The polyhedral tiling fails in 3-dimensional tiling with a tile size of $4 \times 4 \times 4$ but instead generates a 2-dimensional tiling with a tile size of 4×4. Therefore, the 3-dimensional tiling is eliminated from the comparison. Results from optimized codes in Fig. 9.1 and Fig. 9.2 show that the polyhedral optimizations deliver a lower performance for the PPM advection code. This analytical report from the performance monitoring tool shows all tiled codes have more TLB misses compared to the non-optimized code.

Code snippets in Code 9.2, Code 9.3, and Code 9.4 are extracted from the tiled subroutines, with 4×4 tile size, for the X-, Y-, and Z-directional updates. These three snippets represent the tiled structures after the polyhedral transformations. In both Code 9.2 and Code 9.3, loops are tiled in the Y and Z directions and coarse-grained parallelism is applied in the Z direction. The intra-tile loop $c5$ is moved to the innermost and forms a vector loop in the optimized code. However, the $c5$ loop has the longest stride distance for every array in this nested loop structure. A worse data locality with high TLB misses appears in this loop. In Code 9.4, the polyhedral model performs different optimizations. As the Z direction becomes the sweep direction, referencing adjacent array elements causes loop-carried dependences in the Z direction. The polyhedral model performs loop skewing to generate legal tiling and coarse-grained parallelism in $c2$ loop. The innermost loop for the X direction is unchanged in the transformation and the $c5$ loop with unit stride forms an optimal vector loop in this directional update. Inspecting the transformed codes with different tile sizes, similar tiled loop structures are generated after the transformation.

The causes for lower performance could be categorized as follows:

- **Fusion:** Loop fusion greatly reduces the data reuse distance and maximizes the coarse-grained parallel region. Loops with different dimensionality are not considered fusible in the polyhedral model. For example, a 2-dimensional loop will not

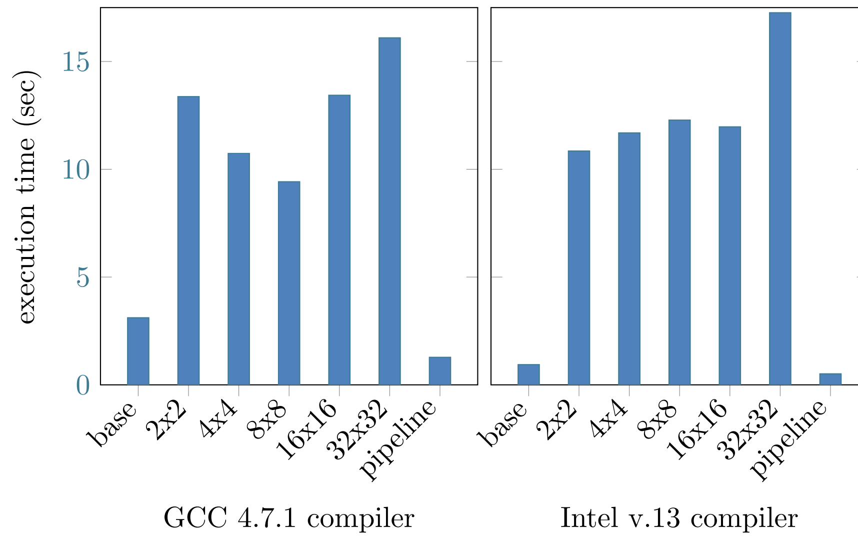

Figure 9.1: Performance results with polyhedral optimization (Intel Sandy Bridge w/ 256-bit SIMD, 2.0GHz)

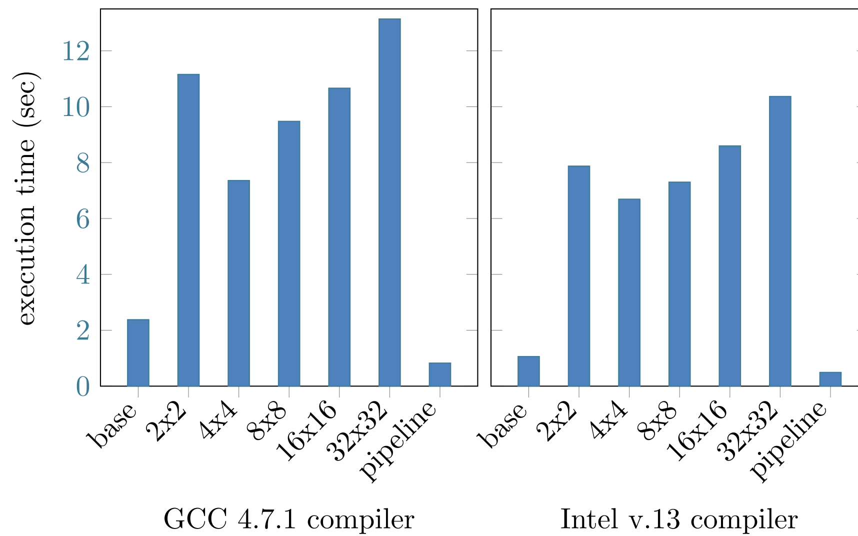

Figure 9.2: Performance results with polyhedral optimization (Intel Nehalem w/ 128-bit SIMD, 2.93 GHz)

be fused into a group of 3-dimensional loops. This generates multiple OpenMP parallel regions in the transformed codes. Expanding the loops with lower dimensionality will be a feasible solution to maximize the loop fusion.

- **Data locality:** Data localities for the tiled code cause high TLB misses and higher cache pressure for the example code. Strided memory accesses easily reference data across multiple pages. And multiple cache lines are fetched into cache memory to form a tile space. Array copying can be applied after the loop tiling to eliminate these performance penalties. This technique has been studied by other researchers and applied in several research implementations [77, 83]. Data is copied into new arrays with the tiled structure. Linear memory access within a tile can be guaranteed after the copying process. Array copying easily expands the overall memory usage and is practical for computation with only a handful of variables. The inevitable computation overhead to perform array copying varies by the array size and the tile structure.

- **Parallelization:** The polyhedral model is able to generate both fine-grained and coarse-grained parallelization. The transformation moves the intra-tile parallel loop to the innermost for fine-grained parallelization. The transformation generates a vector loop with a long stride distance in the experiments. By performing loop interchange in the input code prior to the polyhedral transformation, a better vectorized loop structure with unit stride or shorter stride distance can be obtained. The exact data dependence analysis enables the polyhedral model to perform automatic coarse-grained parallelization. The experiment shows the transformation decision depends on the loop structure in the input code. The outermost loop in the Z-directional update iterates over the sweep direction, and loop-carried dependences are generated after loop fusion. To generate legal parallelization, loop skewing is performed in the iteration space for the Y and Z directions (shown in Code 9.4). The loops are tiled in a rhombus shape and

parallelization happens in the skew diagonals in the Y-Z iteration domain. Suboptimal parallelization could appear in the skewed loop when the size of a diagonal is smaller than the available number of computing cores in a CPU. X- and Y-directional updates have a parallel outermost loop, and the polyhedral model generates parallelization with a square tile structure. Interchanging the loops before the polyhedral transformation could avoid loop skewing in the transformation process.

- **Loop bounds:** The polyhedral model uses the Fourier-Motzkin algorithm to project inner loop iteration variables and determine loop bounds for outer loops. With loop bounds represented by variables (i.e. $nx, ny, nz, andnbdy$), the polyhedral model needs to insert many if statements to control loop execution for specific conditions. Examples can be found in Code 9.2 and Code 9.3. Code generation in the polyhedral transformation expands the code size significantly and introduces complicated control flows in the computation. Replacing these variables with constant numbers greatly simplifies the analysis but it might not be favored in real scientific applications.

Code 9.2: Tiled loop structure in X-directional update after polyhedral transformation

```
!$omp parallel do  private(adiff, azrdif, azldif, ferror, xfr0, xfr01,
!    &                        rhomlr0, c3, c5, c4, c2)
      DO c1 = 0, floor(real(nz) / real(4)), 1
      IF (nbdy >= ceiling(real(-1 * nx + 8) / real(2))) THEN
      DO c2 = 0, floor(real(ny) / real(4)), 1
      DO c3 = -1 * nbdy + 7, nbdy + nx + -2, 1
      DO c4 = max(1,4 * c2), min(ny,4 * c2 + 3), 1
!dir$ ivdep
!dir$ vector always
!dir$ simd
      DO c5 = max(1,4 * c1), min(nz,4 * c1 + 3), 1
       thngy03((c3),(c4),(c5)) = 1.
       dal((c3),(c4),(c5)) = rho((c3),(c4),(c5)) - rho((c3)-1,(c4),(c5))
       dasppm(c3 + -1,(c4),(c5)) = .5 * (dal(c3 + -1,(c4),(c5)) +
     &                               dal(c3 + -1 + 1,(c4),(c5)))
       alsmth(c3 + -1,(c4),(c5)) = rho(c3 + -1,(c4),(c5)) -
     &                               .5 * dal(c3 + -1,(c4),(c5)) -
     &                               sixth * (dasppm(c3 + -1,(c4),(c5)) -
     &                               dasppm(c3 + -1 - 1,(c4),(c5)))
       ...
       absdal((c3),(c4),(c5)) = abs(dal((c3),(c4),(c5)))
       dm((c3),(c4),(c5)) = rho((c3),(c4),(c5)) * deex
       unsmth((c3),(c4),(c5)) = 0.
      END DO
      END DO
      END DO
      END DO
      END IF
      END DO
!$omp end parallel do
```

Code 9.3: Tiled loop structure in Y-directional update after polyhedral transformation

```
!$omp parallel do  private(adiff, azrdif, azldif, ferror, xfr0, xfr01, rhomlr0,
!     &                            c4, c2, c5, c3)
      DO c1 = 0, floor(real(nz) / real(4)), 1
      IF (nbdy >= ceiling(real(-1 * ny + 8) / real(2))) THEN
      DO c2 = ceiling(real(-1 * nbdy + 4) / real(4)),
     &            floor(real(nbdy + ny + -2) / real(4)), 1
      DO c3 = 1, nx, 1
      DO c4 = max(4 * c2,-1 * nbdy + 7),min(4 * c2 + 3,nbdy + ny + -2),1
!dir$ ivdep
!dir$ vector always
!dir$ simd
      DO c5 = max(1,4 * c1), min(nz,4 * c1 + 3), 1
      thngy03((c3),(c4),(c5)) = 1.
      adiff = rho((c3),c4 + -2 + 1,(c5)) - rho((c3),c4 + -2 - 1,(c5))
      ...
      absdal((c3),(c4),(c5)) = abs(dal((c3),(c4),(c5)))
      dm((c3),(c4),(c5)) = rho((c3),(c4),(c5)) * deex
      unsmth((c3),(c4),(c5)) = 0.
      END DO
      END DO
      END DO
      END DO
      END IF
      END DO
!$omp end parallel do
```

Code 9.4: Tiled loop structure in Z-directional update after polyhedral transformation

```
      DO c1 = ceiling(real(-1 * nbdy + -2) / real(4)),
     &          floor(real(nbdy + nz + ny) / real(4)), 1
!$omp parallel do  private(c5, c4, c3)
      DO c2 = max(0,ceiling(real(4 * c1 + -1 * nbdy + -1 * nz + 2) / real(4))),
     & min(floor(real(ny) / real(4)),floor(real(4 * c1 + nbdy + -4) / real(4))), 1
      DO c3 = max(4 * c1 + -4 * c2,-1 * nbdy + 7),
     & min(4 * c1 + -4 * c2 + 3,nbdy + nz + -2), 1
      DO c4 = max(1,4 * c2), min(ny,4 * c2 + 3), 1
      DO c5 = 1, nx, 1
       thngy03((c5),(c4),(c3)) = 1.
       adiff = rho((c5),(c4),c3 + -2 + 1) - rho((c5),(c4),c3 + -2 - 1)
       ...
       absdal((c5),(c4),(c3)) = abs(dal((c5),(c4),(c3)))
      dm((c5),(c4),(c3)) = rho((c5),(c4),(c3)) * deex
       unsmth((c5),(c4),(c3)) = 0.
      END DO
      END DO
      END DO
      END DO
!$omp end parallel do
      END DO
```

9.3.3 Customized experimental setting

The polyhedral model has the ability to fuse loops, parallelize computation, and enhance data localities. However, Fig. 9.1 and Fig. 9.2 show only suboptimal results are generated for the testing codes. The previous subsection presents additional optimizations including array copying, loop interchange, and constant substitution to eliminate the possible optimization concerns in the polyhedral transformation. This subsection presents an extended study to improve the performance using polyhedral optimization.

The loop structure in the example code (shown in Appendix B) exhibits the flexibility

to freely interchange the nested loops. Therefore, customized access patterns in 3-D loops are provided for each single-directional computation.

- **X pass:** When the X direction is the sweep direction, loop-level parallelization can be applied to the outer loops in the Y and Z directions. A 2-dimensional tiling scheme is also applied naturally to these loops and forms the tiled loop structure. The polyhedral model then chooses one intra-tile parallel loop and moves it to the innermost for vectorization. The transformed result in Code 9.2 shows the outermost intra-tile loop, in the Z direction, is chosen to be the vector loop. However, this vector loop has the longest stride distance according to the data layout in all multi-dimensional arrays. With the knowledge that the sweep direction is not ideal for vectorization, moving the intra-tile loop in the Y direction innermost would be a preferred transformation. The nested loops are interchanged to have a new access pattern of $Y \rightarrow Z \rightarrow X$ (from outermost to innermost loops).

- **Y pass:** The unit-stride memory access pattern in the X direction makes the loop in the X direction an ideal candidate for vectorization. The loop in the X direction is moved outermost for both coarse- and fine-grained parallelization. The loop for the sweep direction, the Y direction in this case, is moved innermost to allow the 2-dimensional tiling scheme in the transverse directions. The nested loops therefore have new access pattern of $X \rightarrow Z \rightarrow Y$ in the Y pass.

- **Z pass:** Loop skewing is applied in Code 9.4 for the purpose of coarse-grained parallelization. This transformation might not be optimal due to the suboptimal skewed structure. Following a similar strategy used in the X and Y passes, the loop in the Z direction is moved innermost for the 2-dimensional tiling in the X and Y directions. The loop in the X direction is also moved outermost to generate a vector loop in the X direction after the transformation. $X \rightarrow Y \rightarrow Z$ will be the new order in the Z pass.

In summary, the proposed nested loop structure for the input code has the innermost loop for the sweep direction. The loop with the shortest memory access stride is chosen

from the rest of the loops to be the outermost loop. The polyhedral model is able to generate the tiled, parallelized, and vectorized loops as expected. Fig. 9.3 and Fig. 9.4 show the performance results from the modified loop structure using the Intel Sandy Bridge CPU and Nehalem CPU respectively. Unit stride memory accesses in the vector loop eliminate the gather-scatter operations. A significant performance improvement is found in the new loop structure with a larger tile size on both multi-core CPUs. However, tile sizes smaller than 4×4 deliver worse performance in the Intel Nehalem CPU. There is only a marginal improvement for tile sizes smaller than 4×4 in the Intel Sandy Bridge CPU. To gain a benefit from the 8-way SIMD supported in the Intel Sandy Bridge CPU, the iteration number has to be larger than 8. The same statement holds for the Intel Nehalem CPU, but with a smaller iteration number of 4. Intra-tile loops in smaller tiles will not exploit the SIMD engine with high efficiency. Array copying, discussed earlier, can again provide better data localities and enhance performance for the smaller tile size. A maximal vector length of 4 will be generated from a 2×2 tile size. Data copying therefore can only enhance performance in a CPU with a shorter SIMD width. The performance difference between the GCC compiler and the Intel compiler mainly comes from vectorization. Inspecting the optimization reports generated from both compilers, the Intel compiler definitely performs more aggressive optimization and vectorization.

Loop interchange provides a better loop structure for the polyhedral transformation, but the overall performance is still lower than the performance from non-transformed code. A 32×32 tile size delivers the best performance on both Intel Sandy Bridge and Intel Nehalem CPUs. However, the outermost tiled loop iterates through multiple tiles in only one transverse direction and does not expose enough OpenMP parallelism. The performance monitoring tool reports the low thread concurrency in the computation. Loop coalescing could expand the coarse-grained parallelism by converting the two outermost tiled loops into a single loop. Another reason for this lower performance could be a worse data prefetching from the tiled loop. Hardware prefetching predicts memory access patterns for a limited number of streams and brings in data to cache space in

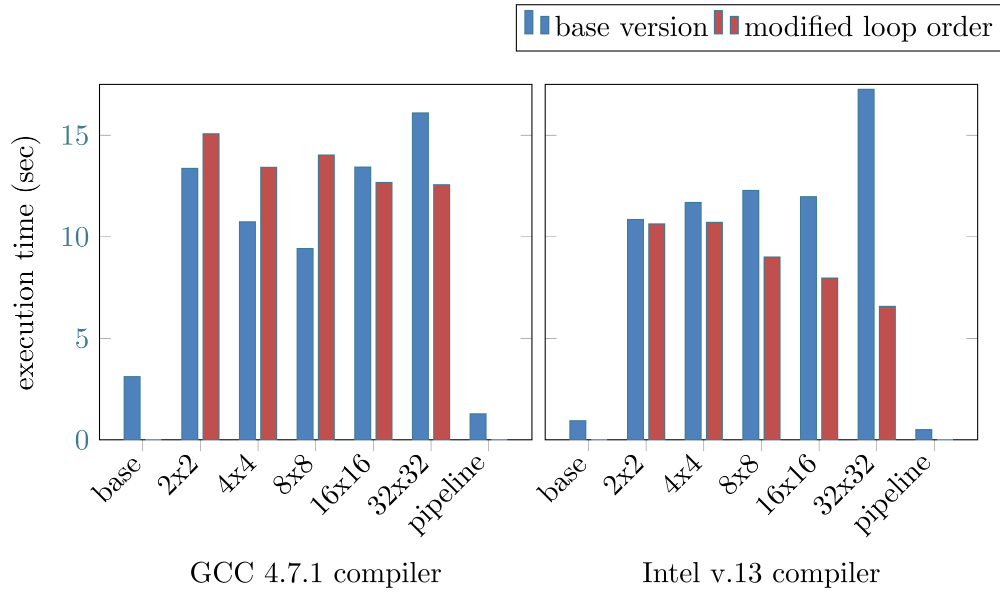

Figure 9.3: Performance comparison with polyhedral optimization (Sandy Bridge w/ 256-bit SIMD, 2.0 GHz)

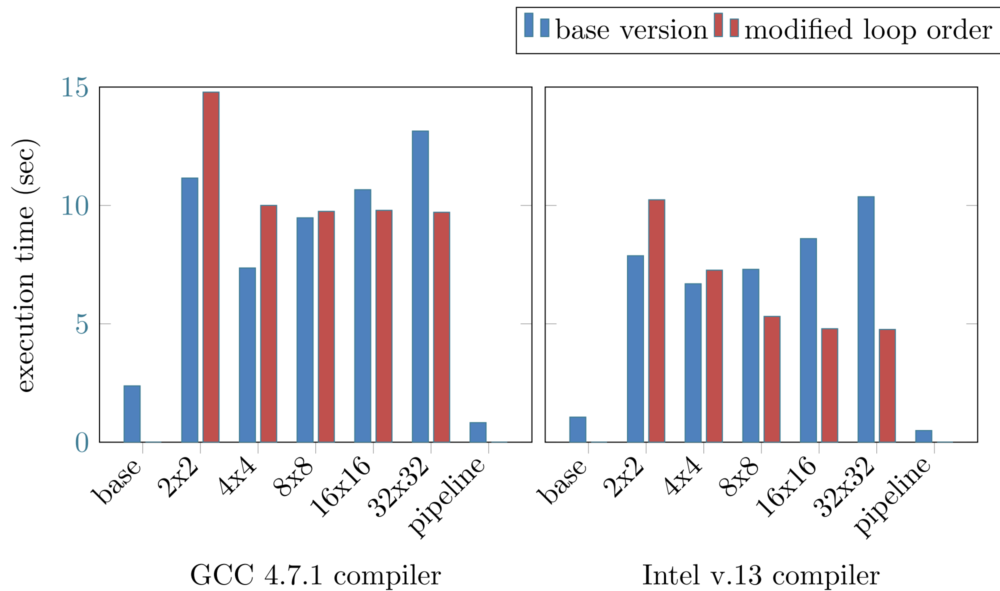

Figure 9.4: Performance comparison with polyhedral optimization (Nehalem w/ 128-bit SIMD,2.93 GHz)

advance. With more than 20 arrays in the computation and with the tiled loop structure, it will be more challenging to exploit hardware prefetching. Software prefetching could be an approach to improve the prefetching scheme.

9.3.4 Conclusion

With a preliminary study using one of the polyhedral implementation, we draw the following conclusions:

1. The polyhedral model provides the potential to perform complex loop transformations and can effectively overcome the phase-ordering concern in compiler optimization. However, the system memory consumption increases significantly with increased code size. The scalability, from the perspective of code size, becomes a major challenge.

2. The legality of transformation is guaranteed in polyhedral transformations. A significant code size expansion could appear in the optimized code and brings a higher pressure to the instruction cache.

3. The powerful tiling capability delivers both auto-parallelization and vectorization in the polyhedral transformation. The study in this dissertation shows suboptimal vectorization in the tiled code. Additional optimizations should be applied to enhance the data locality.

4. To effectively adopt the polyhedral model for research in performance portability, this study suggests the following tasks in future research and development:

   - Including domain-specific knowledge into the polyhedral framework.
   - Addressing the scalability to allow large-scale scientific applications.
   - Program transformations for memory reorganization.

Chapter 10

Conclusion and Future Work

Recent innovations in computer architecture design have resulted in challenges in application portability. Language extensions and new programming models have been developed to accommodate these changes in hardware design. However, programmers must make significant efforts to apply these new programming features and keep the same application running on multiple platforms. Tuning the applications to achieve high performance on multiple platforms is a significant challenge to programmers. To assist application developers achieving high performance on the latest and future HPC systems, this dissertation proposes optimization strategies for CFD applications to achieve performance portability.

First, we have identified the SIMD engines on modern processors as the key to obtaining high performance for scientific application codes. This common component on all platforms makes performance portability possible. To exploit the SIMD engines for high performance computing, a vectorization framework is needed to exploit data-level parallelism using SIMD instructions. The vectorization framework, like other existing compiler vectorizers, focuses on loop-level vectorization. SIMD intrinsic function calls are generated in the vector loops and each intrinsic function is mapped to a SIMD hardware instruction. Without relying on different vectorization schemes on different HPC systems, our tool can achieve vectorization for multiple hardware platforms. These platforms include the IBM Cell processor, IBM Blue Gene/Q and Power 7 multi-core

processors, Intel and AMD's x86 processors, and the Intel MIC architecture. Performance portability with SIMD engines can be achieved in this proposed vectorization framework.

Second, this dissertation proposes an optimization strategy to improve the efficiency of on-chip memory usage. Most CFD codes are expressed in a series of vectorizable loops. A large vector length can lead to bad utilization of the on-chip memory. To maximize the on-chip memory efficiency, we have to keep the vector length short and minimize the memory footprint carefully. We propose a briquette structure and perform memory reorganization exploiting this data structure. The briquette structure serves multiple optimization purposes including exposing SIMD utilization, enhancing data localities, and coalescing memory access. The vectorizable loops with this new memory organization are pipelined over the briquettes in a strip to eliminate redundant unpacking of data records and redundant computation. This briquette-pencil pipeline maximizes both on-chip memory reuse and computational intensity. It also greatly reduces the memory footprint for efficient on-chip memory usage. This optimization strategy makes it possible for data and instructions of the selected CFD applications to fit into the 256 KB SPU on-chip memory. The same benefit is seen for the other multi-core CPUs with 256 KB L2 cache. An average $3.6\times$ speedup from this optimization strategy is shown in the performance study presented in this dissertation.

A third optimization strategy proposed in this dissertation is for the limited memory bandwidth. Increasing computational intensity is the most effective approach to achieve high performance with limited memory bandwidth. The briquette-pencil pipeline greatly increases the cached data reuse and computational intensity for a single-thread of execution. For hardware equipped with last-level cache shared by multiple processing cores, a specialized cache-blocking can exploit the large last-level cache and increase the computational intensity in multithreaded computation. This optimization fuses multiple single-directional updates to minimize the memory traffic to main memory. CFD applications with larger memory footprints achieve higher performance improvements with this optimization strategy.

This dissertation presents strategies for performance portability covering the latest

processors in HPC systems. Motivated by the experience with the IBM Cell processor, a group of optimization strategies and a code translation framework have been developed to assist code generation for high computational performance. These optimizations tackle different programming challenges and generate the memory-optimized, and computation pipelined codes written in different languages with SIMD intrinsic functions. On the CPU platforms, the best-delivered performance in selected CFD applications achieve 30% of the 32-bit peak performance. Applying the strategies to GPU platforms, the best-delivered performances are 57 Gflop/s (for applications with larger memory footprint) and 90 Gflop/s (for applications with small memory footprint). Although only a low percentage of peak performance is achieved on GPUs, these performance measurements are comparable to or higher than other CFD applications implemented on GPUs and reported in the literature cited here. These optimization strategies are applied to large-scale scientific simulations on various HPC platforms [51, 84, 85]. A large simulation run achieves 12% of the peak performance on the Blue Waters machine with a 1.5 petaflop/s sustained performance [85]. This dissertation has shown that source-to-source transformation with domain-specific optimizations makes performance portability possible for the latest CPUs. The following list summarizes the contributions of this dissertation:

- First, we implement the computational fluid dynamics applications on the latest processors, including CPUs and Nvidia GPU, and have achieved uniformly high performance on all devices.

- Second, a strategy using source-to-source code transformations and a translation tool have been developed to achieve high computational performance and make performance portability more feasible.

- Third, we have implemented the PPM gas dynamics applications, applied the optimization strategy, on the leading large-scale computing systems, including Roadrunner (with IBM Cell processors) at Los Alamos National Laboratory, Kraken (with AMD CPUs) at National Institute of Computational Sciences, Itasca (with Intel CPUs) at Minnesota Supercomputing Institute, and Blue Waters (with AMD

CPUs) at National Center for Supercomputing Applications. This work can be extended to computing systems using GPGPUs and many-core processors.

- Finally, this dissertation presents a preliminary study of optimization techniques using the polyhedral model. This study discusses existing optimization obstacles in the latest polyhedral transformations. This study concludes that polyhedral optimization has to overcome a list of challenges to be applicable for scientific applications.

10.1 Future Work

The research presented in this dissertation could be extended in the following areas:

10.1.1 Optimization development

Scientific applications rely on program optimization to achieve their highest computational performance. Optimization development falls into two categories: performing optimization automatically, and designing new optimizations. State-of-the-art compiler frameworks provide capabilities for parsing, analyzing, transformation and code generation. Performing transformation with automation is feasible using existing optimizing compiler. However, the gap between domain knowledge and compiler design is the major obstacle to achieve optimal optimization. Bridging such gap can significantly improve the compilation, and deliver higher performance. Joint efforts between domain experts and compiler developers are needed to deal with the following challenges: (i) uncovering hidden optimization potential in scientific applications; (ii) designing an efficient compilation interface to convey domain knowledge; (iii) implementing new optimizations in existing compiler platforms.

10.1.2 Extending the framework to cover more CFD applications

This dissertation presents strategies to achieve performance portability for those CFD applications that use structured grids. With close collaboration with CFD application

developers, co-designing tools with compiler developers to aid program transformation can make performance portability easier. With minor modifications to make applications conform to the requirements of the proposed framework, we believe the same framework can be made applicable to more scientific applications using structured grids. The applicable candidates may include the meteorology research application – CM1 [64], the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [86] , and the Weather Research & Forecasting Model (WRF)[87]. The applicability of the briquette-like data structures, the feasibility of applying directionally-split algorithms, and the enhancement of both coarse and fine-grained parallelization are the potential research subjects. The proposed co-design strategies for performance portability should also be applicable to CFD applications using unstructured grids or scientific applications in different research domains.

10.1.3 Performance portability for future HPC systems

This dissertation identified the SIMD engine and on-chip memory as key elements that make performance portability with the proposed strategies possible. Many on-going hardware research efforts focus on providing more powerful computing devices to reach the goal of Exascale computation. Reviewing leading HPC systems on the TOP500 list, three paths in hardware design could provide high potential for Exascale computing. They include multicores with powerful CPUs, manycores with simplified low-power computing devices, and the accelerators with specialized processing elements. All have proved their capability to deliver high computational power in existing HPC systems. Current predictions on Exascale systems envision massive parallelism with extreme low power consumption. This dissertation has shown source-to-source transformation with domain-specific optimization can deliver high computational intensity, low memory bandwidth consumption, and high SIMD engine utilization for current systems. We believe a robust source-to-source transformation framework accompanied by domain knowledge should also be able to support performance portability in the future Exascale era.

References

[1] The STREAM benchmark. http://www.cs.virginia.edu/stream/.

[2] S.Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, 2005.

[3] M. Puschel, J. M F Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[4] M. Telgarsky, J.C. Hoe, and J. M F Moura. Spiral: Joint runtime and energy optimization of linear transforms. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 3, pages III–III, 2006.

[5] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12, 2011.

[6] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.

[7] Daniel Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[8] Yixun Liu, E.Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[9] Huimin Cui, Jingling Xue, Lei Wang, Yang Yang, Xiaobing Feng, and Dongrui Fan. Extendable pattern-oriented optimization directives. *ACM Trans. Archit. Code Optim.*, 9(3):14:1–14:37, October 2012.

[10] Open64 compiler. www.open64.net.

[11] Qing Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[12] Qing Yi. Automated programmable control and parameterization of compiler optimizations. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

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

[14] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 50–64, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, Supercomputing '88, pages 98–105, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[16] Zhang Yuanyuan and Zhao Rongcai. An open64-based cost analytical model in auto-vectorization. In *Educational and Information Technology (ICEIT), 2010 International Conference on*, volume 3, pages V3–377–V3–381, 2010.

[17] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. *SIGPLAN Not.*, 41(6):132–143, June 2006.

[19] Daniel S. McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. Automatic SIMD vectorization of fast fourier transforms for the larrabee and avx instruction sets. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 265–274, New York, NY, USA, 2011. ACM.

[20] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 169–178, New York, NY, USA, 2005. ACM.

[22] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. *SIGPLAN Not.*, 39(6):82–93, June 2004.

[23] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[24] VAST compiler. http://www.crescentbaysoftware.com/.

[25] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Rashindra Manniesing, Ireneusz Karkowski, and Henk Corporaal. Automatic SIMD parallelization of embedded applications based on pattern recognition. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 349–356, London, UK, UK, 2000. Springer-Verlag.

[27] Gilles Pokam, Stéphane Bihan, Julien Simonnet, and François Bodin. SWARP: a retargetable preprocessor for multimedia instructions: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):303–318, January 2004.

[28] Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. Scout: a source-to-source transformator for simd-optimizations. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 137–145, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] Clang:a C language family frontend for LLVM. http://clang.llvm.org.

[30] Serge Guelton. SAC: An efficient retargetable source-to-source compiler for multimedia instruction sets. http://www.cri.ensmp.fr/classement/doc/A-429.pdf.

[31] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. CUDA-lite: Reducing GPU programming complexity. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 1–15. Springer-Verlag, Berlin, Heidelberg, 2008.

[32] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[33] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[34] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. *SIGPLAN Not.*, 41(11):325–335, October 2006.

[35] Michael Kistler, John Gunnels, Daniel Brokenshire, and Brad Benton. Petascale computing with accelerators. *SIGPLAN Not.*, 44(4):241–250, February 2009.

[36] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[37] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel computing experiences with CUDA. *Micro, IEEE*, 28(4):13–27, 2008.

[38] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):78–90, January 2011.

[39] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45(6):86–97, June 2010.

[40] N. Sundaram, A. Raghunathan, and S.T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, 2009.

[41] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 213–224, Washington, DC, USA, 2010. IEEE Computer Society.

[42] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[43] G.S. Murthy, M. Ravishankar, M.M. Baskaran, and P. Sadayappan. Optimal loop unrolling for GPGPU programs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, 2010.

[44] Guochun Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez. Direct self-consistent field computations on GPU clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–8, 2010.

[45] Guochun Shi, Steven Gottlieb, Aaron Torok, and Volodymyr Kindratenko. Accelerating quantum chromodynamics calculations with GPUs. In *2010 Symposium on Application Accelerators in High-Performance Computing (SAAHPC10)*, 2010.

[46] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model asuca production code. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010.

[47] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, 2008.

[48] H.P. Hofstee. Power efficient processor architecture and the cell processor. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262, 2005.

[49] B. Flachs, S. Asano, Sang H.Dhong, H.P. Hofstee, G. Gervais, Roy Kim, T. Le, Pe-ichun Liu, J. Leenstra, J. Liberty, B. Michael, Hwa-Joon Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, Vandung To, and E. Iwata. The microarchitecture of the synergistic processor for a cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):63–70, 2006.

[50] Geoffrey Taylor. The instability of liquid surfaces when accelerated in a direction perpendicular to their planes. i. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 201(1065):192–196, 1950.

[51] Paul R. Woodward, Guy Dimonte, Gabriel M Rockefeller, Christopher L Fryer, Guy Dimonte, W Dai, and R J Kares. Simulating rayleigh-taylor (RT) instability using ppm hydrodynamics scale on roadrunner. In *Proc. NECDC conference*, 2011.

[52] Paul R. Woodward. A complete description of the PPM compressible gas dynamics scheme. *Implicit Large Eddy Simulation: Computing Turbulent Flow Dynamics*, 2005.

[53] Paul R. Woodward, J. Jayaraj, P.-H. Lin, and William Dai. First experience of compressible gas dynamics simulation on the los alamos roadrunner machine. *Concurr. Comput. : Pract. Exper.*, 21(17):2160–2175, December 2009.

[54] Paul R. Woodward, David H. Porter, Falk Herwig, Marco Pignatari, J. Jayaraj, and P.-H. Lin. The hydrodynamic environment for the s process in the he-shell flash of agb stars. In *10th Symposium on Nuclei in the Cosmos*, 01 2009, 0901.1414.

[55] Paul R. Woodward. Numerical methods for astrophysicists. *Astrophysical Radiation Hydrodynamics*, 54:256 – 326, 1986.

[56] William Dai, Guy Dimonte, Christopher Fryer, Paul R. Woodward, Falk Herwig, David Porter, Tyler Fuchs, Tony Nowatzki, and Michael Knox. The piecewise-parabolic boltzmann advection scheme (PPB) applied to multifluid hydrodynamics. In *Conference: International Conference on Computational Science*, 2010.

[57] Fernando F Grinstein, Len G Margolin, and William J Rider. *Implicit large eddy simulation: computing turbulent fluid dynamics*. Cambridge university press, 2007.

[58] The ASCI sPPM benchmark code. https://asc.llnl.gov/computing_resources/purple/archive/benchma

[59] Ronald H. Cohen, William P. Dannevik, Andris M. Dimits, Eliason Donald E., Arthur A. Mirin, Ye Zhou, David H. Porter, and P.R. Woodward. Three-dimensional simulation of a richtmyer–meshkov instability with a two-scale initial perturbation. *Physics of Fluids*, 14(10):3692–3709, 2002.

[60] J. L. Baer. Multiprocessing systems. *IEEE Trans. Comput.*, 25(12):1271–1277, December 1976.

[61] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[62] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 147–162, New York, NY, USA, 2012. ACM.

[63] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[64] Cloud model 1 (CM1). http://www.mmm.ucar.edu/people/bryan/cm1/.

[65] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language*

design and implementation, PLDI '89, pages 246–257, New York, NY, USA, 1989. ACM.

[66] Pohua P Chang, Scott A Mahlke, William Y Chen, and Wen-Mei W Hwu. Profile-guided automatic inline expansion for c programs. *Software: Practice and Experience*, 22(5):349–369, 1992.

[67] Paul R. Woodward, J. Jayaraj, P.-H. Lin, and Pen-Chung Yew. Moving scientific codes to multicore microprocessor CPUs. *Computing in Science Engineering*, 10(6):16–25, 2008.

[68] P.-H. Lin, J. Jayaraj, Paul R. Woodward, and Pen-Chung Yew. A code transformation framework for scientific applications on structured grids. Technical Report 11-021, University of Minnesota, Twin Cities, November 2011.

[69] Paul R. Woodward, Jagan Jayaraj, Pei-Hung Lin, Pen-Chung Yew, Michael Knox, Jim Greensky, Anthony Nowatski, and Karl Stoffels. Boosting the performance of computational fluid dynamics codes for interactive supercomputing. *Procedia Computer Science*, 1(1):2055–2064, 2010.

[70] Paul R Woodward, David H Porter, Falk Herwig, Marco Pignatari, Jagan Jayaraj, and Pei-Hung Lin. The hydrodynamic environment for the s process in the he-shell flash of agb stars. *arXiv preprint arXiv:0901.1414*, 2009.

[71] P.-H. Lin, J. Jayaraj, Paul R. Woodward, and Pen-Chung Yew. A study of performance portability using piecewise-parabolic method (ppm) gas dynamics applications. *Procedia Computer Science*, 9(0):1988 – 1991, 2012.

[72] C. Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[73] Terence Parr. *The definitive ANTLR reference.* building domain-specific languages. The Pragmatic Bookshelf, 2007.

[74] ANTLR (another tool for language recognition). http://www.antlr.org/.

[75] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf, 2009.

[76] Open fortran parser (OFP). http://fortran-parser.sourceforge.net/.

[77] Qing Yi. Poet: a scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.*, 42(6):675–706, June 2012.

[78] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[79] Jaewook Shin. Introducing control flow into vectorized code. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 280–291, 2007.

[80] Uday Kumar Reddy Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model.* PhD thesis, Ohio State University, 2008.

[81] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010.

[82] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[83] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[84] P. Ramaprabhu, Guy Dimonte, Paul R. Woodward, C. Fryer, Rockefeller G., Muthuraman K., P.-H. Lin, and J. Jayaraj. The late-time dynamics of the single-mode rayleigh-taylor instability. *Physics of Fluids*, 24(7):074107–074107–21, 2012.

[85] Paul R. Woodward, J Jayayaraj, Mike Knox, David H Porter, Christopher L Fryer, Guy Dimonte, Candace C Joggerst, Gabriel M Rockefeller, William W Dai, Robert J Kares, et al. Simulating turbulent mixing from richtmyer-meshkov and rayleigh-taylor instabilities in converging geometries using moving cartesian grids. Technical report, Los Alamos National Laboratory (LANL), 2013.

[86] Livermore unstructured lagrangian explicit shock hydrodynamics (LULESH). https://codesign.llnl.gov/lulesh.php.

[87] The weather research and forecasting model (WRF). http://www.wrf-model.org/index.php.

Appendix A

Glossary and Acronyms

Care has been taken in this thesis to minimize the use of jargon and acronyms, but this cannot always be achieved. This appendix defines jargon terms in a glossary, and contains a table of acronyms and their meaning.

A.1 Glossary

- **Computational Intensity** – The ratio of floating point operations to the memory access (memory read from, or write to the distant memory).

- **Memory Wall** – The growing disparity of speed between CPU and memory outside the CPU chip.

- **Briquette** – A cubical structure that can store a group of grid cells in the scientific computation.

- **CVMGM** – Conditional vector merge functions used in Cray vector machines.

A.2 Acronyms

Table A.1: Acronyms

Acronym	Meaning
HPC	High Performance Computing
SIMD	Single Instruction Multiple Data
CFD	Computational Fluid Dynamics
PPM	Piecewise Parabolic Method
GPGPU	General Purpose Graphics Processing Unit
DSP	Digital Signal Processing
CUDA	Compute Unified Device Architecture
MIC	Many Integrated Core Architecture
QCD	Quantum Chromodynamics
WRF	Weather Research and Forecast
SMT	Simultaneous Multi-Thread
L1	First Level Cache
L2	Second Level Cache
LLC	Last Level Cache
PPU	Power Processor Unit
SPU	Synergistic Processing Unit
DMA	Direct Memory Access
SIMT	Single Instruction Multiple Threads
AVX	Advanced Vector Extension
VMX	Vector Multimedia eXtension
MPI	Message Passing Interface
TLB	Translation Lookaside Buffer
AST	Abstract Syntax Tree
SCoP	Static Control Part

Continued on next page

Table A.1 – continued from previous page

Acronym	Meaning
SPMD	Single Program Multiple Data
SM	Streaming Multiprocessor

Appendix B

PPM example code

Code B.1: PPM advection example for polyhedral model

```
      subroutine ppmmfX(xxl, yyb, zzf, rho, voxels,
     &                   smlrho, smallu, ujiggle, time,
     &                   nx, ny, nz, nbdy, ipass, itymes, ifvis, ArgsImg,
      parameter (nfluids=0)
      parameter (nvarsfld=nfluids*10)
      parameter (nvarsflddiag=nvarsfld)
      parameter (nvars=1+nvarsfld)
      dimension    xxl(-nbdy:nx+nbdy+2)
      dimension    yyb(-nbdy:ny+nbdy+2)
      dimension    zzf(-nbdy:nz+nbdy+2)
      dimension     rho(1-nbdy:nx+nbdy,1-nbdy:ny+nbdy,1-nbdy:nz+nbdy)
      dimension      ArgsImg(18)
      dimension         sintwopixl(1-nbdy:nx+nbdy)
      dimension            dm(1-nbdy:nx+nbdy, ny, nz)
      dimension        unsmth(1-nbdy:nx+nbdy, ny, nz)
      dimension          rhol(1-nbdy:nx+nbdy, ny, nz)
      dimension          rhor(1-nbdy:nx+nbdy, ny, nz)
      dimension          drho(1-nbdy:nx+nbdy, ny, nz)
      dimension          rho6(1-nbdy:nx+nbdy, ny, nz)
      dimension           dal(1-nbdy:nx+nbdy, ny, nz)
```

```
      dimension          absdal(1-nbdy:nx+nbdy,ny,nz)
      dimension          dasppm(1-nbdy:nx+nbdy,ny,nz)
      dimension          damnot(1-nbdy:nx+nbdy,ny,nz)
      dimension          alsmth(1-nbdy:nx+nbdy,ny,nz)
      dimension          alunsm(1-nbdy:nx+nbdy,ny,nz)
      dimension           uxavl(1-nbdy:nx+nbdy,ny,nz)
      dimension          duxavl(1-nbdy:nx+nbdy,ny,nz)
      dimension          sigmal(1-nbdy:nx+nbdy,ny,nz)
      dimension           dvoll(1-nbdy:nx+nbdy,ny,nz)
      dimension          dmassl(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy01(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy02(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy03(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy04(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy05(1-nbdy:nx+nbdy,ny,nz)
      dimension        thngy06(1-nbdy:nx+nbdy,ny,nz)
c
      ...
c
      do k = 1,nz
      do j = 1,ny
      do 100    i = 2-nbdy,nx+nbdy
       dal(i,j,k) = rho(i,j,k) - rho(i-1,j,k)
       absdal(i,j,k) = abs (dal(i,j,k))
      enddo
      enddo
      do k = 1,nz
      do j = 1,ny
      do 5000    i = 3-nbdy,nx+nbdy-2
      adiff = rho(i+1,j,k) - rho(i-1,j,k)
      azrdif = 3. * dal(i,j,k)  -  dal(i-1,j,k)  -  adiff
      azldif = dal(i+2,j,k)  -  3. * dal(i+1,j,k)  +  adiff
      thngy01(i,j,k) = azldif
      if (thngy01(i,j,k) .lt. 0.)    thngy01(i,j,k) = - thngy01(i,j,k)
```

```
 thngy02(i,j,k) = azrdif
 if (thngy02(i,j,k) .lt. 0.)   thngy02(i,j,k) = - thngy02(i,j,k)
 ferror = .5 * (thngy01(i,j,k) + thngy02(i,j,k))  /
1          (absdal(i,j,k) + absdal(i+1,j,k) + small)
 if (thngy06(i,j,k) .lt. 0.)   thngy06(i,j,k) = 0.
 if (thngy06(i,j,k) .gt. 1.)   thngy06(i,j,k) = 1.
 if (thngy06(i,j,k) .gt. unsmth(i,j,k)) unsmth(i,j,k)=thngy06(i,j,k)
 dasppm(i,j,k) = .5 * (dal(i,j,k) + dal(i+1,j,k))
 thngy03(i,j,k) = 1.
 if (dasppm(i,j,k) .lt. 0.)   thngy03(i,j,k) = -1.
 thngy01(i,j,k) = thngy03(i,j,k) * dal(i,j,k)
 thngy02(i,j,k) = thngy03(i,j,k) * dal(i+1,j,k)
 if (thngy02(i,j,k) .lt. thngy01(i,j,k))   thngy01(i,j,k) = thngy02(i,j,k)
 thngy01(i,j,k) = 2. * thngy01(i,j,k)
 thngy02(i,j,k) = thngy03(i,j,k) * dasppm(i,j,k)
 if (thngy02(i,j,k) .lt. thngy01(i,j,k))   thngy01(i,j,k) = thngy02(i,j,k)
 if (thngy01(i,j,k) .lt. 0.)   thngy01(i,j,k) = 0.
 damnot(i,j,k) = thngy03(i,j,k) * thngy01(i,j,k)
 enddo
 enddo
 do k = 1,nz
 do j = 1,ny
 do 6000   i = 4-nbdy,nx+nbdy-2
 alsmth(i,j,k) =  rho(i,j,k)  -  .5 * dal(i,j,k)  -
&                 sixth * (dasppm(i,j,k) - dasppm(i-1,j,k))
 alunsm(i,j,k) =  rho(i,j,k)  -  .5 * dal(i,j,k)  -
&                 sixth * (damnot(i,j,k) - damnot(i-1,j,k))
 enddo
 enddo
 do k = 1,nz
 do j = 1,ny
 do 7000   i = 4-nbdy,nx+nbdy-3
 rhol(i,j,k) = alunsm(i,j,k)
 rhor(i,j,k) = alunsm(i+1,j,k)
```

```
      thngy04(i,j,k) = 3. * rho(i,j,k)  -  2. * rhor(i,j,k)
      thngy05(i,j,k) = 3. * rho(i,j,k)  -  2. * rhol(i,j,k)
      if (((rho(i,j,k)-rhol(i,j,k)) * (rho(i,j,k)-rhor(i,j,k))) .ge. 0.)
     &then
         rhol(i,j,k) = rho(i,j,k)
         rhor(i,j,k) = rho(i,j,k)
         thngy04(i,j,k) = rho(i,j,k)
         thngy05(i,j,k) = rho(i,j,k)
      endif
      if (((rhor(i,j,k) - rhol(i,j,k)) * (thngy04(i,j,k) - rhol(i,j,k)))
     &                                    .gt. 0.)
     &        rhol(i,j,k) = thngy04(i,j,k)
      if (((rhor(i,j,k) - rhol(i,j,k)) * (thngy05(i,j,k) - rhor(i,j,k)))
     &                                    .lt. 0.)
     &        rhor(i,j,k) = thngy05(i,j,k)
      rhol(i,j,k) = alsmth(i,j,k)
     &          -  unsmth(i,j,k) * (alsmth(i,j,k) - rhol(i,j,k))
      rhor(i,j,k) = alsmth(i+1,j,k)
     &          -  unsmth(i,j,k) * (alsmth(i+1,j,k) - rhor(i,j,k))
      drho(i,j,k) = rhor(i,j,k) - rhol(i,j,k)
      rho6(i,j,k) = 6. * (rho(i,j,k)  -  .5*(rhol(i,j,k)+rhor(i,j,k)))
      enddo
      enddo
c
      ...
c
      do k = 1,nz
      do j = 1,ny
      do 3700    i = 5-nbdy,nx+nbdy-3
       xfr0 = .5 * sigmal(i,j,k)
       xfr01 = 1.  -  forthd * xfr0
       thngy01(i,j,k) = rhor(i-1,j,k)
       if (uxavl(i,j,k) .lt. 0.)    thngy01(i,j,k) = rhol(i,j,k)
       thngy02(i,j,k) = - drho(i-1,j,k)
```

```
         if (uxavl(i,j,k) .lt. 0.)    thngy02(i,j,k) = drho(i,j,k)
         thngy03(i,j,k) = rho6(i-1,j,k)
         if (uxavl(i,j,k) .lt. 0.)    thngy03(i,j,k) = rho6(i,j,k)
         rhomlr0 = thngy01(i,j,k)  + xfr0*(thngy02(i,j,k)  + xfr01*thngy03(i,j,k))
         dvoll(i,j,k) = sigmal(i,j,k) * deex
         if (uxavl(i,j,k) .lt. 0.)    dvoll(i,j,k) = - dvoll(i,j,k)
         dmassl(i,j,k) = rhomlr0 * dvoll(i,j,k)
        enddo
        enddo
        do k = 1,nz
        do j = 1,ny
        do 8000    i = -nbdy+5,nx+nbdy-4
         rho(i,j,k) = (dm(i,j,k) + dmassl(i,j,k) - dmassl(i+1,j,k))
     &    / (deex + dvoll(i,j,k) - dvoll(i+1,j,k))
        enddo
        enddo
        do k = 1,nz
        do j = 1,ny
        do 9200    i = 1,nx
        courmx = max (courmx, courno(i,j,k))
9200    continue
        enddo
        enddo
        return
        end
```