Supporting Applications Involving Dynamic Data Structures and Irregular Memory Access on Emerging Parallel Platforms

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Bin Ren, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

Dr. Gagan Agrawal, Advisor

Dr. P. Sadayappan

Dr. Radu Teodorescu

Abstract

SIMD accelerators and many-core coprocessors with *coarse-grained* and *fine-grained* level parallelism, become more and more popular. Streaming SIMD Extensions (SSE), Graphics Processing Unit (GPU), and Intel Xeon Phi (MIC) can provide orders of magnitude better performance and efficiency for parallel workloads as compared to single core CPUs. However, parallelizing irregular applications involving dynamic data structures and irregular memory access on these parallel platforms is not straightforward due to their intensive control-flow dependency and lack of memory locality. Our efforts focus on three classes of irregular applications: *Irregular Trees and Graphs Traversals, Irregular Reductions and Dynamic Allocated Arrays and Lists*, and explore the mechanism of parallelizing them on various parallel architectures from both fine-grained and coarse-grained perspectives.

We first focus on the *traversal of irregular trees and graphs*, more specifically, a class of applications involving the traversal of many pointer-intensive data structures, *e.g.* Random Forest, and Regular Expressions, on various fine-grained SIMD architectures, *e.g.* the Streaming SIMD Extension (SSE), and Graphic Processing Unit (GPU). We address this problem by developing an intermediate language for specifying such traversals, followed by a run-time scheduler that maps traversals to SIMD units. A key idea in our run-time scheme is converting branches to arithmetic operations, which then allows us to use SIMD

hardware. In order to make our approach fast, we demonstrate several optimizations including a stream compaction method that aids with control flow in SIMD, a set of layouts that reduce memory latency, and a tiling approach that enables more effective prefetching. Using our approach, we demonstrate significant increases in single-core performance over optimized baselines for two applications.

However, different SIMD architectures have different features, so a significant challenge to our previous work is to automatically optimize applications for various architectures, *i.e.*, we need to implement *performance portability*. Moreover, one of the first architectural features programmers look to when optimizing their applications is the memory hierarchy. Thus, we design a portable optimization engine for accelerating irregular data traversal applications on various SIMD architectures by emphasizing on improving the data locality and hiding memory latency. The contributions of this work are two fold: First, we develop a set of data layout optimizations that improve spatial locality for applications that traverse many irregular data structures. Unlike prior data layout optimizations, our approach incorporates a notion of both inter-thread and intra-thread spatial reuse into data layout. Second, we enable performance portability for these applications on SIMD architectures by accurately modeling the impact of inter and intra thread locality on program performance. As a consequence, our model can predict which data layout optimization to use on a wide variety of SIMD architectures.

We next explore the possibility of efficiently parallelizing two *irregular reduction* applications on Intel Xeon Phi architecture, an emerging many-core coprocessor architecture with long SIMD vectors, via data layout optimization. During this process, we also identify a general data management problem in the CPU-Coprocessor programming model, *i.e.*,

the problem of automating and optimizing *dynamic allocated data structures* transfers between CPU and Coprocessors. For dynamic multi-dimensional arrays, we design a set of compile-time solutions involving heap layout transformation, while for other irregular data structures such as linked lists, we improve the existing shared memory runtime solution to reduce the transfer costs.

Dynamic allocated data structures like List are also commonly used in high-level programming languages, such as Python to support dynamic, flexible features to increase the programming productivity. To parallelize applications in such high-level programming languages on both coarse-grained and fine-grained parallel platforms, we design a compilation system linearizing dynamic data structures into arrays, and invoking low level multi-core, many-core libraries. A critical issue of our linearization method is that it incurs extra data structure transformation overhead, especially for the irregular data structures not reused frequently. To address this challenge, we design a set of transformation optimization algorithms including an inter-procedural *Partial Redundancy Elimination (PRE)* algorithm to minimize the data transformation overhead automatically.

This is dedicated to the ones I love: my parents and my wife.

Acknowledgments

It has been a long road, and there are so many people to whom I would like to extend my sincerest thanks! It would not have been possible to write this doctoral dissertation without their help and support. To only some of them, it is possible to give particular mention here.

First, I would like to thank my Advisor Prof. Gagan Agrawal for his constant patience, support, and advice over the last six years. I came to OSU in 2008, when I had very limited knowledge and experience in both research and parallel computing area. It is impossible for me to finish this difficult, but extremely rewarding journey without his teaching, encouragement and help, and it is impossible for me to learn so much without his unsurpassed insight into compiler, parallel computing, and research methodology as well. Moreover, his enthusiasm for research were also contagious and motivational for me, especially during the toughest time. From now on, he will be an excellent example for me for my whole career.

Next, I would like to acknowledge the financial, academic, technical and all other kinds of supports from National Science Foundation, The Ohio State University, and the Computer Science and Engineering Department, especially the head of our department, Prof. Xiaodong Zhang, and all administrative staffs of our department, especially Kathryn M. Reeves and Lynn Lyons. I also want to extend my thanks to all Professors who have taught me in past six years for their vivid teaching and helpful suggestions in both my study and research, and they are: Prof. Atanas Rountev, Prof. Srinivasan Parthasarathy, Prof. D.K.

Panda, Prof. Arnab Nandi, Prof. Timothy Long, and Prof. Rephael Wenger. Especially, I would like to thank my candidacy committee members, Prof. P. Sadayappan, Prof. Radu Teodorescu, and Prof. Feng Qin for their valuable time and invaluable advice, without which, this dissertation would not have been possible.

I also would like to extend my sincerest thanks to my internship mentors, Dr. Todd Mytkowicz from Microsoft Research, and Dr. Nishkam Ravi and Dr. Yi Yang from NEC Labs. They are not only my mentors but also my friends, and they helped me a lot in both my research and my life. I would never forget the beautiful sunny days for sailing in Seattle, and the excited happy moment of championing the Volleyball Game in Princeton. At the same time, I would like to thank all collaborators in my past research, Dr. Bradford L. Chamberlain, Dr. Steve Deitz, Dr. James R. Larus, Dr. Tomi Poutanen, Dr. Wolfram Schulte, Dr. Min Feng, and Dr. Srimat Chakradhar, without whose generous efforts, it is impossible for the projects in this dissertation to come true.

I also would like to thank all my colleagues in my lab, Data-Intensive and High Performance Computing Research Group. Former and present members of our group have been very supportive and helpful both inside and outside school, and they are: David Chiu, Qian Zhu, Fan Wang, Wenjing Ma, Vignesh Ravi, Wei Jiang, Tantan Liu, Tekin Bicer, Jiedan Zhu, Xin Huo, Yu Su, Linchuan Chen, Yi Wang, Mehmet Can Kurt, Mucahid Kutlu, Jiaqi Liu, and Sameh Shohdy. Our group has been a source of friendships as well as good advice and collaboration.

Also I would like to thank all my friends in and out our department. I just want my buddies to know how much I appreciate them and their friendship. Every time when I struggled, my buddies comforted me; every time when I felt happy, my buddies were around

me. Many places have witnessed our friendship, and we have also gained a lot from it, not only in our research, but also in our life.

At last, special thanks are extended to my parents and my wife for their constant generous support! I love you forever!

Vita

December 7th, 1983	Born - Zhengding, China
2006	B.S. Software Engineering, Beihang University, Beijing, China
2008	M.S. Software Engineering, Beihang University, Beijing, China
2013	M.S. Computer Science & Engineering, Ohio State University, Columbus, OH
June, 2011 - September, 2011	Research Intern, Microsoft Research, Redmond, WA
May, 2013 - August, 2013	Research Intern, NEC Lab America, Princeton, NJ

Publications

Research Publications

Bin Ren, Nishkam Ravi, Yi Yang, Min Feng, Gagan Agrawal, and Srimat Chakradhar. "Automating and Optimizing Data Transfers for Many-core Coprocessors". In *Proceedings of The 28th ACM International Conference on Supercomputing (ICS)*, June, 2014. (Poster Paper, Full Paper is Under Submission).

Xin Huo, **Bin Ren**, and Gagan Agrawal "A Programming System for Xeon Phis with Runtime SIMD Parallelization". In *Proceedings of The 28th ACM International Conference on Supercomputing (ICS)*, June, 2014.

Bin Ren, Todd Mytkowicz, and Gagan Agrawal. "A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures". In *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. "SIMD Parallelization of Applications that Traverse Irregular Data Structures". In *Proceedings of The 2013 International Symposium on Code Generation and Optimization (CGO)*, February, 2013.

Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. "Fine-Grained Parallel Traversals of Irregular Data Structures". In *Proceedings of The 21th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October, 2012. (Poster Paper).

Bin Ren, and Gagan Agrawal. "Compiling Dynamic Data Structure in Python to Enable the Use of Multi-core and Many-core Libraries". In *Proceeding of The 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October, 2011.

Bin Ren, Gagan Agrawal, Brad Chamberlain, and Steve Deitz. "Translating Chapel to Use FREERIDE: A Case Study in Using an HPC language for Data-intensive Computing". In *Proceeding of The 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments(HIPS) held in conjunction with IPDPS*, May, 2011.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

Our overall research work has been motivated by two facts emerging recently. On one hand, starting from the last few years, it is no longer possible to improve computing speed by simply increasing clock frequencies. As a result, various SIMD accelerators and many-core architectures, like SSE, GPUs, and Intel Xeon Phi have become cost-effective means for scaling performance. These parallel platforms offer us both coarse-grained (multi-thread) level and fine-grained (instruction) level parallelism.

On the other hand, parallelizing applications involving dynamic data structures and irregular memory access on these parallel platforms is not straightforward. Our work focuses on three classes of such applications:

- Irregular Trees and Graphs Traversals like Decision Trees Traversal and Regular Expression Matching,
- Irregular Reductions like Molecular Dynamics and Euler,
- Applications with dynamic multi-dimension arrays, and linked lists written in *C/C++* or some higher level languages like *Python*.

For parallelizing the first class of applications, there are two challenges: first, it is difficult to explore parallelism, since normally such pointer-intensive data structures show

dynamic behavior, and traversal on them involves high control-flow dependency; second, their poor data locality incurring high memory latency even for uni-processor makes them barely benefit from the parallelism. The first challenge is more obvious for fine-grained SIMD architectures, like SSE and GPUs due to the more strict, lockstep behavior of fine-grained parallelism than coarse-grained parallelism; while the second one is crucial for both levels of parallelism and various parallel architectures with different memory hierarchies.

Our first two efforts, Chapter 2 and Chapter 3, target on speeding up a class of irregular applications in traditional languages like $C/C++$ that involve independent traversals of many instances of a pointer-based data structure on SIMD architectures, SSE, and many-core coprocessors, GPUs. Examples of the class of applications we target include prediction using a collection of decision trees [10], traversal of an indexing structure for a number of simultaneous (but independent) queries [28], matching with regular expressions [134], parsing XML documents [102], and frequent pattern mining [47], including finding common subgraphs in a set of graphs [138]. These applications arise in domains as diverse as machine learning, compilation, intrusion detection, web services, databases, and data mining.

These applications traverse independent data structures for two reasons. First, applications can manipulate a large number of logically independent data structures. For example, the forest of decision trees produced by a machine learner or the set of alternative patterns used in an intrusion detection system such as Snort [118] represent independent computations that can be proceeded in parallel. Second, applications can traverse a single irregular data structure, but do so with many independent inputs. The approach in our work handles both cases. While applications of this type can easily be parallelized across multiple cores, SIMD within each core can provide a multiplicative improvement in performance.

Our work develops an intermediate language for specifying such traversals, followed by a run-time scheduler that maps traversals to SIMD units. However, different SIMD architectures have different features, so a significant challenge to our previous work is to automatically optimize applications for various architectures, *i.e.*, we need to implement *performance portability*. Moreover, one of the first architectural features programmers look to when optimizing their applications is the memory hierarchy. Thus, we design a portable optimization engine for accelerating irregular data traversal applications on various SIMD architectures by emphasizing on improving the data locality and hiding memory latency.

Next, we explore parallelism for the second and the third classes of applications, irregular reductions, and dynamic allocated multi-dimension array and structures with multi-level pointers on latest Intel Xeon Phi architecture in Chapter 4 and Chapter 5. For irregular reduction applications, we emphasize on the challenges causing by indirected array access, such as random memory access, and write conflicts within the same SIMD register, and for applications with dynamic multi-dimension arrays and multi-level pointers, we focus on the challenge of automating and optimizing dynamic data transfers between CPU and Xeon Phi.

Both of these efforts are to reduce the programmers' burden of working with Xeon Phi, while maintain the high performance of the code by a set of optimizations. The former designs a set of APIs to leverage the *hierarchical parallelism* to explore both shared memory MIMD and SIMD parallelization, while the latter designs and implements a compile-time and runtime integrated source-code to source-code transformation framework to generate the optimized data transfer related codes automatically.

Moreover, still within the scope of the third class of applications, dynamic allocated data structures are also commonly used in very high-level languages to support many dynamic, flexible features to offer programmers high programming productivity. However, usually, the performance of these dynamic data structures is very low due to their irregular behavior, and it is impossible for them to support the use of coarse-grained and fine-grained parallelism from existing multi-core and many-core libraries, like data-intensive applications library [63], and linear algebra application libraries [90, 104]. Because these libraries expect parameters to be multi-dimensional arrays, and cannot be directly invoked when the application is based on dynamic data structures.

To bridge the gap between performance and productivity for high level languages, our last effort, Chapter 6, picks up Python, and develops a Python based compilation system that can replace dynamic data structures with arrays, and invoke libraries for multi-core and many-core architectures for specific types of computations. This work also generalizes our idea of layout optimization used in previous works to a broader area, and designs an *Inter-Procedural PRE* algorithm to reduce the data transformation overhead during the optimization process.

In this Chapter, we first introduce the SIMD accelerator and many-core coprocessors, especially the emerging Intel Xeon Phi many-core coprocessor architecture. Next, we give an overview of our existing work including fine-grained SIMD traversals on irregular data structures on SSE and GPUs, efficiently parallelizing two irregular applications on emerging Intel Xeon Phi architectures, automating and optimizing data transfers for Xeon Phi, and compiling dynamic data structures in Python to use both coarse-grained and fine-grained parallelism libraries.

1.1 Introduction to SIMD accelerators and Many-core coprocessors

This part introduces the parallel platforms used in this dissertation, SSE (Streaming SIMD Extension), a typical SIMD accelerator, and GPUs and Intel Xeon Phi, two emerging many-core coprocessors. Especially, we describe the Intel Xeon Phi in more detail, since people are less familiar with this new many-core architecture.

1.1.1 SSE

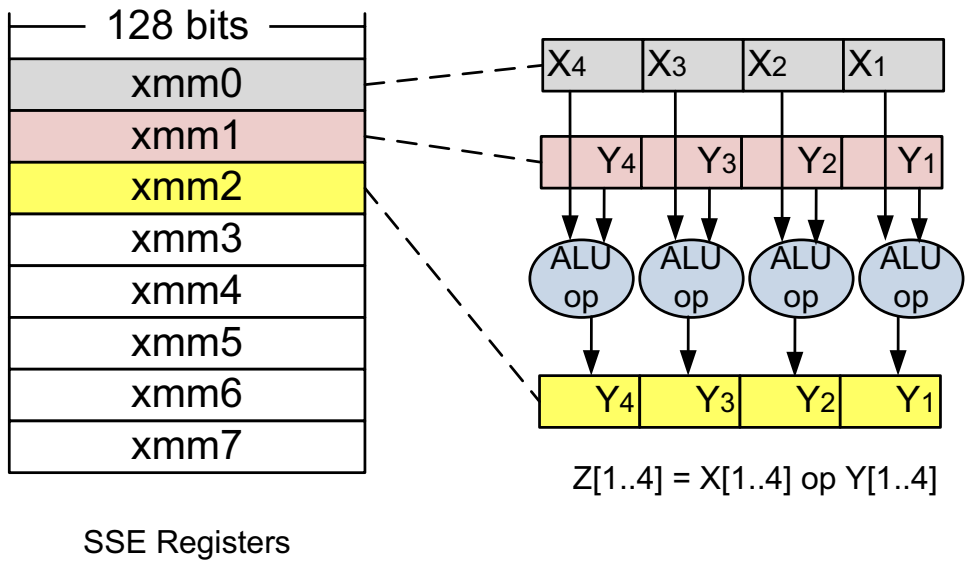


Figure 1.1: An Illustration of Using SSE Registers

SSE has been part of the x86 since 1999 and is widely used in a large variety of computation fields [58, 38, 40, 119]. The latest version of SSE is 4.2. As shown in Figure 1.1, SSE essentially includes a set of (normally 8) multi-lane registers that can support 4 32-bit

(total 128 bits) float operations at the same time [1]. SSE offers programmers a set of intrinsics or instructions to process various arithmetic, logic and memory access operations. In modern multi-core CPU architecture, each core has its own SSE unit, and both scalar unit and SSE unit share the same memory hierarchy.

1.1.2 Many-core Coprocessors: Nvidia GPUs and Intel Xeon Phi

As we all know, many-core coprocessors can provide orders of magnitude better performance and efficiency for parallel workloads as compared to multi-core CPUs, and are being widely adopted as accelerators for high performance computing. An increasing number of *top 500* supercomputers[2] are now based on accelerators, like the Nvidia GPUs or the x86-compatible Intel Xeon Phi (MIC) systems. Particularly, MIC is designed to leverage existing x86 experience and benefit from familiar parallelization tools, libraries, and programming models, including OpenMP [25], MPI [45], CilkPlus [9] and TBB [116].

GPUs

Comparing with SSE units, Nvidia GPUs architectures are more complex. Figure 1.2 and Figure 1.3 from Nvidia programming documents [105] illustrate the Nvidia Tesla GPU architecture, and the basic idea of CUDA programming model designed for GPU programming. The basic structure of Nvidia Fermi GPUs architecture is similar to Tesla, and the most significant difference is that it has a L1/L2 cache hierarchy for tolerating irregular accesses and improving the memory performance. In GPUs architecture, multiple low frequency processors are organized into a few of SMs (Streaming Multiprocessor). Each SM has its own small shared memory, constant, texture cache (and L1 cache for Fermi),

[1] To increase the computation power, the Intel Sandybridge processor doubles its vector length to 256 bits, and the new instruction set is called AVX (Advanced Vector Extensions).

[2] Top 500 Supercomputers. http://www.top500.org.

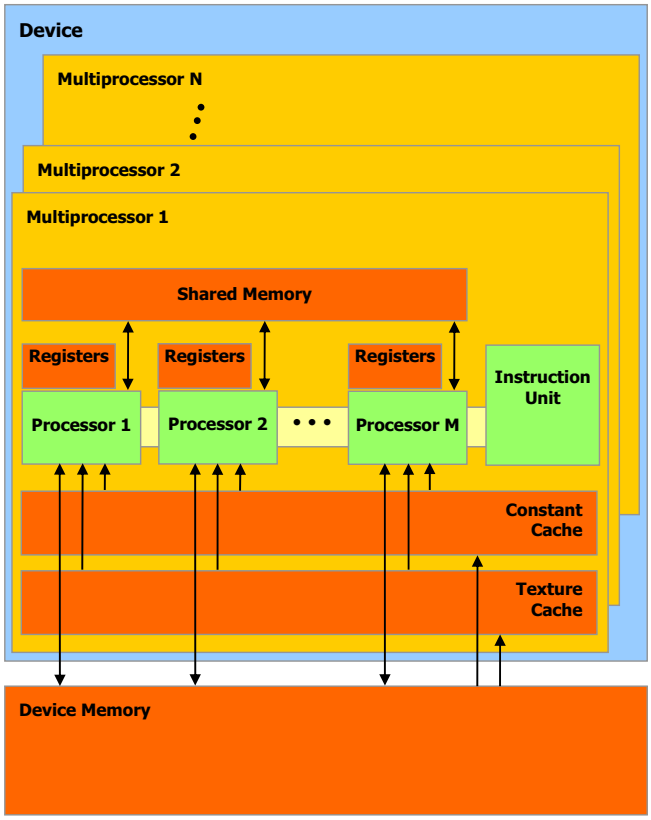


Figure 1.2: Nvidia Tesla GPUs Architecture

and device memory (and larger L2 cache for Fermi) is shared among all SMs. In CUDA programming, tasks of each kernel are organized into a grid-block-thread hierarchy: each kernel is a grid, which is partitioned into multiple blocks, and each block contains multiple threads. The GPUs CUDA kernels and the CPU host code sections are executed interleaved, so we call it a heterogeneous programming model. The mapping relationship of CUDA programming and GPU device is as following: each kernel is corresponding to the whole device, and each block is processed on a specific SM, and each thread is executed by a processor.

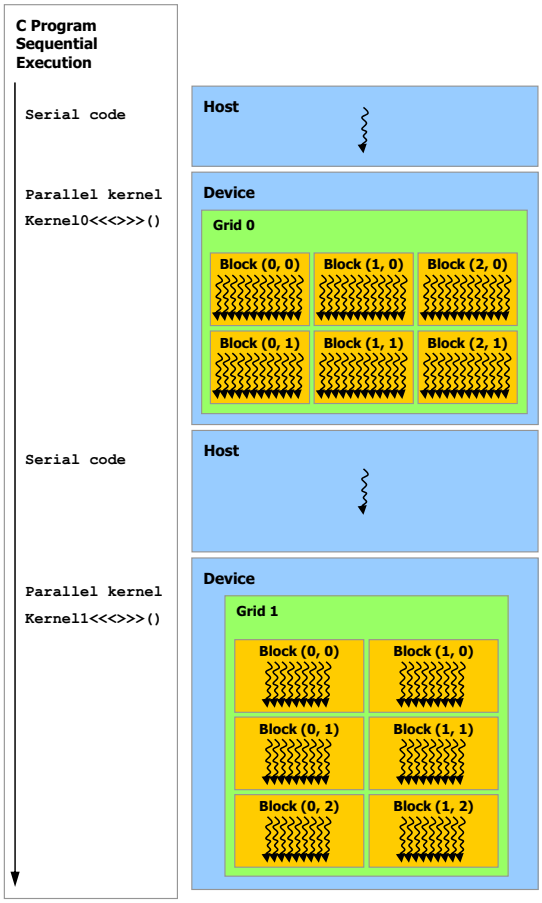


Figure 1.3: CUDA Programming Model

Xeon Phi

In the available MIC systems, there are 60 or 61 x86 cores organized with shared memory. These cores are low frequency in-order ones, and each supports as many as 4 hardware threads. Additionally, there are 32 512-bit vector registers on each core for SIMD operations. The main memory sizes vary from 8 GB to 16 GB, and the memory is shared by all cores. The L1 cache is 32 KB, entirely local to each core, whereas each core has a coherent L2 cache, 512 KB, where cache for different cores are interconnected in a ring.

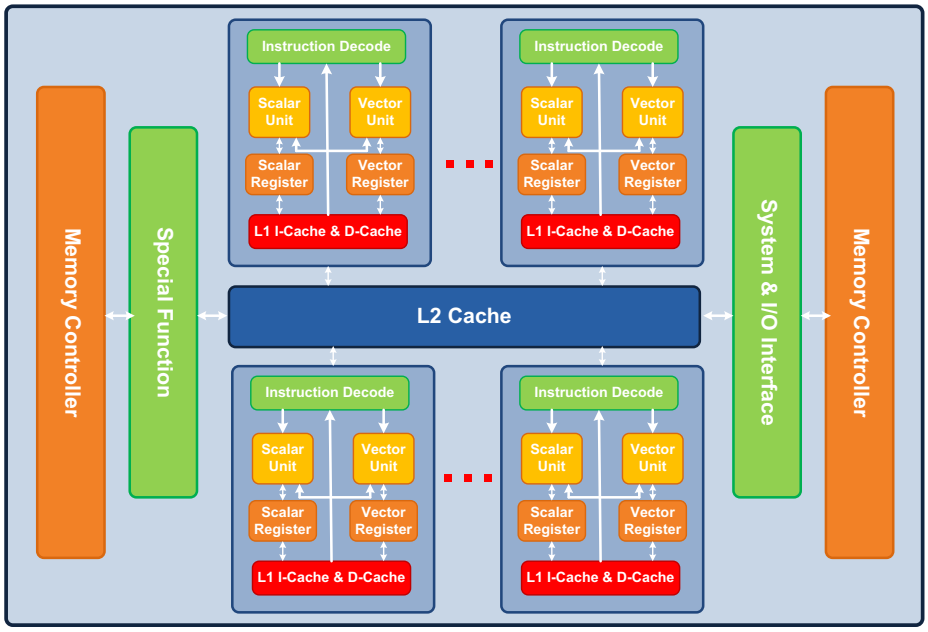


Figure 1.4: The Illustration of Intel MIC Architecture

Our work focuses on three important features of Intel MIC architecture, which need to be exploited for obtaining high performance:

Wide SIMD Registers and Vector Processing Units (VPU): VPU has been treated as the most significant feature of Xeon Phi by many previous studies [84, 126, 110, 34]. The reason is that the Intel Xeon Phi coprocessor has doubled the SIMD lane width compared to Intel Xeon processor, i.e., 256-bit to 512-bit, which means that it is possible to process 16 (8) identical floating point (double precision) operations at the same time. In addition, we have a new 512-bit SIMD instruction set called Intel Initial Many Core Instructions (Intel IMCI), which has built-in *gather* and *scatter* operations that allow irregular memory accesses, a hardware supported *mask* data type, and *write-mask* operations that allow operating on some specific elements within the same SIMD register. Even though all of these

new instructions could potentially be simulated by the programmers in the SIMD Streaming Extension (SSE) model, explicit new instructions allow easier implementation of more irregular parallelism. Note that SIMD instructions can be generated by the ICC compiler through the *auto-vectorization* option, or the programmers could use IMCI instruction set directly. The former needs low programming effort, though current compilation systems have several limitations and do not always obtain high performance. In comparison, the latter option can achieve the best performance, however, is tedious and error prone, and creates non-portable code.

Large Number of Hyper-threads: Each Xeon Phi core allows up to 4 hyper-threads, in another word, we can have as many as 240/244 hardware threads sharing the same memory on Xeon Phi. This provides us with massive Multiple Instruction Multiple Data (MIMD) parallelism with shared memory, which has not been common in the past.

Coherent L2 Cache: Intel Xeon Phi architecture uses coherent L2 Cache with ring interconnection. When a L2 cache miss occurs for a specific core, an address request is sent to the ring. If the address is found in another core's L2 cache, the corresponding data is forwarded back along the ring. In the worst case, the entire process may take hundreds of clock cycles. Thus, Xeon Phi reduces the number of L2 cache misses, but even an L2 cache hit can be very expensive. Thus, data locality is crucial for the overall performance.

1.2 Dissertation Contributions

This section introduces the contributions of our research, including five components.

1.2.1 Fine-Grained Parallel Traversals of Irregular Data Structures on SIMD Architectures

The irregular data structures targeted by our work have significant amount of *fine-grained* parallelism, which makes it possible to parallelize them on SIMD architectures like SSE and GPUs. However, effective parallelization of independent traversals of irregular data structures on a SIMD unit requires addressing multiple challenges. One such challenge is related to the uneven amount of work each SIMD unit might have to perform while traversing different structures. Another challenge is that these applications involve branch operations, which cannot be parallelized on SIMD units. Memory latency while traversing pointer-based data structures is another issue.

This work develops techniques to address these problems. Moreover, we offer a solution to programmers interested in developing SIMD parallelized implementations of these applications, by developing an intermediate language and a run-time scheduler. The intermediate language exposes several types of operations, which can be used to specify the traversal involved in the application. Several optimizations are implemented in the run-time scheduler, including a *stream compaction* method, several *layouts* that reduce memory latency (while also allowing branch operations to be replaced by arithmetic operations), and a *tiling* scheme.

Overall, the contributions of this work are:

- Identification of an opportunity to exploit fine-grain data parallelism in important, latency critical algorithms widely used in production software.

- An approach to exploiting fine-grained parallelism in the traversal of pointer-based data structures, with a specific emphasis on trees and graphs.

- An illustration of the practicality of our approach by demonstrating significant single-core speedups of two applications that use irregular data structures. Applying it to two real random forest applications results in a single-core speedup of 9-17X on a variety of data-sets. We also apply our technique to produce a single-core speedup of nearly 5X in an existing regular expression engine.

The overall objective of this work is to handle the first challenge—*fine-grained parallelism* further explained in Section 2.1.4. We provide a comprehensive study on the second one—*memory locality* in next section.

1.2.2 A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures

For irregular data-traversal applications, memory locality is a critical factor for improving the performance, which is further explained in Section 2.1.4. Previous work preliminarily proposed a layout optimization strategy for SSE environment without any further discussion. This work puts this topic into a more general background—*performance portability*, and tries to establish a theoretical analysis model to guide the programmers' practice on various SIMD architectures with distinct memory hierarchies.

Performance Portability: When developing performance-sensitive applications, programmers have to optimize for a diverse set of architectures and architectural features. These choices make the development of portable high performance applications challenging. Optimizing an application on any one particular architecture is a challenging task—optimizing that application for several architectures is a daunting, if not impossible, task. Because of this complexity, programmers are unable to guarantee the performance of their applications

on the wide range of architectures on which their application will run. In other words, it is difficult for programmers to guarantee an application's *performance portability*.

Apparently, one of the first architectural features programmers look to when optimizing their applications is the memory hierarchy, specifically, for our irregular applications. In sequential applications, a programmer need only consider locality from the view-point of a single thread (e.g. *intra-thread* locality). However, for modern SSE based processors, multi-core, or SIMD architectures, the memory hierarchy is shared by multiple threads, so locality, shared among different threads (*inter-thread* locality) can dominate an application's performance [94, 131]. Clearly, it would be desirable for a code generator to automatically optimize an application's locality for a given architecture, thereby removing the burden of performance portability from the programmer. Unfortunately, in decades of research on restructuring compilers, we as a community have not developed such a solution.

Rather than seeking a general solution to this hard problem, we demonstrate how to enable performance portability across a wide variety of SIMD architectures for our irregular data traversal applications. We present an application-class specific optimization framework that targets a variety of architectures that use SIMD parallelism. This includes an x86 system with SSE instructions and two generations of NVIDIA GPUs. Each of these architectures differs significantly in their memory hierarchies. Our optimization framework includes a family of data layout optimizations that help achieve inter-thread spatial locality, intra-thread spatial, or a combination of the two.

We find that the relative performance among different layouts depends upon an application's characteristics (e.g. whether all trees are processed with equal likelihood, or if probability of taking all branches is the same or not), as well as architectural characteristics (e.g. features of memory hierarchy and degree of SIMD parallelism). Thus, we develop

a detailed performance model that can help select an appropriate layout for a given application/architecture combination. The key insight of our model—and why we are able to accurately model real world applications running on diverse machines—is that, for the applications in our restricted domain, the number of L2 cache misses turns out to be an effective predictor of performance.

Overall, the contributions of this work are as follows:

- First, we describe three novel data layout optimizations that are designed to extract intra and/or inter thread locality from applications that traverse a large number of irregular data structures on SIMD hardware.

- Second, we demonstrate the efficacy of our data layout optimizations by showing significant speedups of two real world applications on three diverse SIMD architectures.

- Third, we introduce an analytic model that removes the burden of performance portability from the programmer by accurately modeling which of our data layout optimizations to use on a particular architecture.

1.2.3 Efficiently Parallelizing Irregular Applications on Xeon Phi by a Programming System

Use of SSE-like instruction sets has always been a hard problem, and it turns out that such parallelism has not been consistently used for applications outside dense matrix or imaging kernels. Moreover, there are significant programming differences between CUDA and SSE-like instruction sets, since they target SIMT and SIMD models, respectively. Specifically, while coalesced memory accesses are important for performance in SIMT programming, parallelism is still available, whereas programmers need to explicitly create

aligned and contiguous accesses in the case of SSE or IMCI. Similarly, while branches are automatically managed in SIMT, with masks internally implemented, programmers or compilers must identify instructions executed by all threads with SSE/IMCI.

Effectively exploiting the power of a coprocessor like Xeon Phi requires that we exploit both MIMD and SIMD parallelism. While the former can be done through Pthreads or OpenMP, it is much harder to extract SIMD performance. This is because the restrictions on the model make hand-parallelization very hard. At the same time, productions compilers are unable to exploit SIMD parallelism for many of the cases.

This work focuses on the problem of application development on any system that supports both shared memory parallelism and SSE-like SIMD parallelism, with a specific emphasis on the Intel Xeon Phi system. We describe an API and a runtime system that helps extract both shared memory and SIMD parallelism. One of the key ideas in our approach is to exploit the information about underlying communication patterns, to both partition and schedule the computation for MIMD parallelism, and reorganize the data for achieving better SIMD parallelism. While our approach is general, we currently focus on irregular reductions.

Overall, the contributions of this work are as follows:

- First, we provide an *end-to-end* application development system for the Xeon Phi architecture, or more broadly, any system with both shared memory and SIMD parallelism.

- Second, our work can be viewed as providing a CUDA or OpenCL-like programming API for SSE-like instructions, where the responsibility for optimizing irregular accesses patterns or managing control flows is the responsibility of the underlying library.

- Third, we evaluate the efficacy and efficiency of our system by two applications involving irregular reduction communication patterns, Molecular Dynamic and Euler, and achieve good speedup.

1.2.4 Automating and Optimizing Data Transfers for Many-core Coprocessors

Accelerating parallel computation using many-core coprocessors requires specification of code regions that can be profitably offloaded to the coprocessor and executed as independent tasks. These code regions have been specified by the developer using low-level APIs (e.g., CUDA [106] and OpenCL [100]) till recently. The software available with Xeon Phi, as well as the emerging *directive-based* models for GPU programming, are providing much higher-level APIs for using accelerators. However, even with such high-level APIs, there are many challenging issues. Particularly, orchestrating data transfers for multi-level pointers using `in/out` or equivalent clauses is cumbersome and error-prone.

With the goal of further improving productivity of HPC programmers while also maintaining performance, we focus on easing data transfer related efforts, considering both compile-time and runtime solutions. While such data transfers for static arrays can be handled by ICC compiler[3] today, and solutions proposed in the literature [115, 46, 81, 62] can handle dynamically allocated one-dimensional arrays, the open problem is handling dynamically allocated multi-dimensional arrays or other structures with multi-level pointers.

It turns out that the problem is quite complex, particularly because the choice of the mechanism used for automatically inserting data transfer clauses impacts memory layouts and access functions (subscripts) on the coprocessor. Because of nature of the accelerators and complex interactions between the resulting source-code and the native compiler on the

[3]Intel C++ Compiler. http://www.intel.com/Compilers.

accelerator, the performance can be impacted in multiple ways. Overall, in order for the solution to perform well:

- Redundant data transfers between the CPU and the accelerator should be eliminated or minimized,
- Data transfer times should be reduced by utilizing Direct Memory Accesses (DMA),
- Memory allocation overheads on the accelerator (or even the host) should be kept low, and
- Last but not the least, memory layout and access should allow for aggressive memory-related compiler optimizations (e.g., vectorization and prefetching) from the native compiler, as they are critical for obtaining performance from the accelerator.

We observe that the prior solutions [60, 81, 115] do not consider these factors together, as they focus primarily on data transfer reduction. In particular, the effect of memory layout [143, 19] on DMA, cache, and compiler optimizations have been largely overlooked.

This work describes an automated framework that uses both compile-time and runtime solutions to address this problem. This system includes a simple but effective compile-time solution, where we linearize the heap without having to modify the memory accesses (subscripts), by using a *pointer reset* approach. The idea is to identify and parse all the malloc statements for a given multi-dimensional array and generate code for obtaining the total memory size (say s) for that multi-dimensional array. The malloc statements for the given array are then replaced by a single malloc statement that allocates a memory chunk of size s. Code is generated to correctly reset all the pointers of the array into this large chunk of memory. This allows the memory accesses to stay unmodified. This method scores well on

all the four metrics mentioned above and maintains code readability. One underlying observation that motivates this approach is that most scientific, big data and HPC applications (that can potentially benefit from a coprocessor) read data and allocate memory towards the beginning of the program and then proceed to process it. Memory allocation statements as well as memory access statements can be tracked, parsed and modified (if certain legality conditions are satisfied).

For the cases where our compile-time approach cannot apply, we also explore runtime solutions. The background is that a system like Xeon Phi also has shared memory implementations available between the main processor and accelerator. We also investigate and optimize the performance of the runtime memory management approach, by providing certain improvements to the existing coherence protocol. The best compile-time solution consistently performs better than the optimized runtime scheme, but is not as generally applicable. In order to combine performance with generality, we describe a mechanism for integrating the two disjoint approaches using a simple source-to-source transformation. The idea is to simultaneously and selectively insert implicit and explicit data transfer clauses in the application at compile time.

We have implemented our compile-time solution as a transformation using the Apricot framework [115], and evaluated it within the context of application execution on Xeon Phi coprocessor. We use a test suite comprising benchmarks from different sources, which involve dynamically allocated multi-level pointers. We show that our proposed compile-time solution can perform 2.5x-5x faster than original runtime solution, and the CPU-MIC code with our compile-time solution can achieve 1.5x-2.5x speedup comparing to the 16-thread CPU version.

1.2.5 Compiling Dynamic Data Structures in Python to Enable the Use of Multi-core and Many-core Libraries

Dynamic allocated data structures are not only used for traditional high performance languages, but also used in many high level, high productivity languages like Python to provide users high-level abstract utilities and relative flexible dynamic features. As programmer productivity is extremely important, there is a growing trend of applying these high-level languages for high performance computing, since they are often simpler to learn (especially, for programmers in certain domains), and result in much more concise code. However, because of their interpreted nature and the use of high-level constructs, they also often result in poor performance, besides being not able to exploit parallelism on multi-cores and GPUs.

Clearly, it will be very desirable if translators can be built to automatically or semi-automatically translate programs written in high-level languages for scalable execution on multi-cores and/or GPUs. This work describes one such system. Our work is driven by the growing popularity of Python, and the need for scaling numerical computations on multi-cores and GPUs, using the existing libraries.

Overall, the contributions of this work are as follows:

- First, we present a Python based compilation system that invokes libraries for multi-core and many-core architectures for specific types of computations. Because usually these libraries only accept dense memory buffer rather than dynamic irregular data structures like list in Python, our linearization and memory optimization strategies can be applied for the data structure transformation process, which demonstrates more generality of our previous work.

- Second, to enable such optimizations, we have developed a demand-driven *inter-procedural PRE algorithm*, and a novel *Homogeneity Checking* algorithm to reduce the layout optimization overhead.

- Third, we have evaluated our framework using two data mining and linear algebra applications written in pure Python, and it demonstrates that by our translation and optimization framework, we are able to generate code which is only $10-20\%$ slower than the hand-written C code that uses the same libraries.

1.3 Outline

The rest of this dissertation is organized as following: Chapter 2 introduces the details of mapping fine-grained parallel traversals of irregular data structures on SIMD architectures. Chapter 3 presents the portable data locality optimization engine for traversing irregular data structures on a variety of SIMD architectures like NVIDIA GPUs with Tesla and Fermi architectures, and Intel SSE architecture. Chapter 4 describes our solution of efficiently parallelizing irregular reduction applications on Xeon Phi architecture. Chapter 5 presents the integrated compile-time, runtime framework of automating and optimizing data transfers for many-core coprocessors, such as Xeon Phi. Chapter 6 introduces the framework of compiling irregular data structures in Python down to multi-core and many-core architectures, especially the inter-procedural PRE algorithm to reduce the data structure transformation overhead. Chapter 7 describes some potential future work, and Chapter 8 concludes this dissertation.

Chapter 2: Fine-Grained Parallel Traversals of Irregular Data Structures on SIMD Architectures

Fine-grained data parallelism is becoming increasingly prevalent in mainstream processors, such as the x86 and ARM, as the length of vector instructions is increasing[4]. The most common fine-grained data-parallel hardware, the Steaming SIMD Extensions (SSE), has been part of the x86 since 1999 and is widely used in graphics [58], image, video, and signal processing [38], and scientific and engineering applications [40, 119]. As such fine-grained data parallelism becomes a ubiquitous processor feature with increasing performance, it is desirable to exploit this feature for irregular computations as well.

However, programs that rely on irregular, pointer-based data structures benefit little from SIMD execution because of the mismatch between the strict, lockstep behavior of SIMD parallelism and the dynamic, data-driven behavior of programs that manipulate irregular data structures.

This Chapter develops techniques to address the challenges of mapping a class of non-numeric, non-graphic applications, which perform computations while traversing many independent, irregular data structures, to SIMD units. Such kind of mapping not addressed in prior work is non-trivial and is the first challenge of fine-grained parallel irregular data structure traversals on SIMD architectures.

[4]For example, the Intel Sandy-bridge processor doubled its vector length to 256 bits.

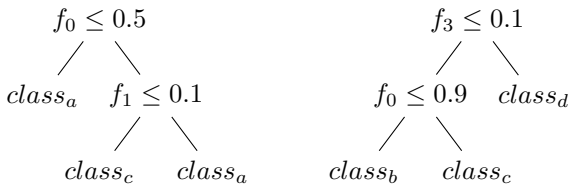
2.1 Anatomy of Three Irregular Programs

This section introduces three common algorithms that manipulate irregular data structures—Random Forest, B+-Tree Forest and Regular Expressions that are used for evaluating our existing work on SIMD architectures.

2.1.1 Random Forests

Random (decision) forests are a data mining technique used to classify an input—or a set of features—into a fixed number of categories [10]. A random forest is a collection of binary decision trees. To classify an input, each tree is traversed, comparing features of the input against threshold values, and producing a result as its categorical membership for that input.

To be more concrete, consider a random forest made of two simple trees, shown below. Each tree can classify one input, made of three features (f_0,f_1, and f_3), into one of four classes ($class_a$,$class_b$,$class_c$, and $class_d$).



Each node in the trees performs one of two actions. If a node is an internal node, it compares one feature of the input against a constant threshold and branches accordingly to a left or right child depending on the result of the comparison. If the node is a leaf node, it simply stores the class label for that tree into a global counter.

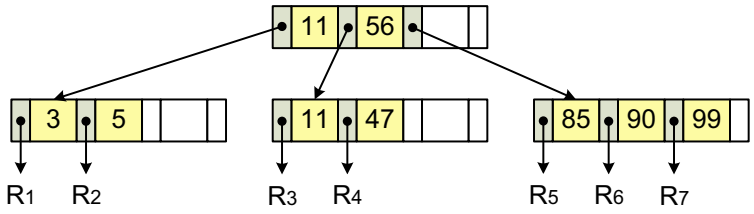


Figure 2.1: An Example of B+-Tree Structure

2.1.2 B+-Tree

B+-Tree is a popular indexing structure used in databases (e.g. for indexing) and data servers (e.g. for file-systems) [23]. The basic B+-Tree operations are: for the internal node, a comparison operation is performed between the input value and the key value; and for the leaf node, a output operation is performed to generate the searched record.

Consider the simple B+-Tree example in **Figure 2.1**. For an input query with an integer number as its key, we compare its key against the key values in an internal node and perform a branch operation. This process is repeated until we arrive at the leaf node, where the information about the searched record is output.

A single B+-Tree can be represented as a hashing function in conjunction to large a number of independent sub-trees[5]. The hashing function represents the logic of the first n levels of the tree, i.e. it is a mechanism by which we can quickly reference any one of the 2^n sub-trees.

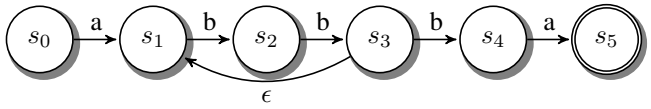
In a database or data server where a B+-tree is deployed, it is common to have a heavy query work-load, with several independent inputs at any given time, which makes the application suitable for exploiting parallelism. After the hashing function is applied to each

[5]http://publib.boulder.ibm.com/infocenter/idshelp/v117/index.jsp?topic=%2Fcom.ibm.perf.doc%2Fids_prf_763.htm

input, there can be independent queries on different sub-trees. These are analogous to independent traversal of different weak classifiers for a single input in random forests, with one key difference - not all sub-trees might be concurrently processed at any given time. We refer to this traversal pattern as *sparse buckets accesses*. Overall, this application is distinct from random forest because parallelism is arising from processing of different inputs, and yet, the issues in efficient SIMD execution are almost the same, as we elaborate below.

2.1.3 Regular Expression Matching

Regular expressions are a common way to match patterns against large bodies of text or binary data. In our work, we use a non-deterministic finite automaton (NFA) to simulate a regular expression, similar to Thompson's original regular expression compiler [129]. To simulate an NFA, we walk–or traverse–a graph, moving node to node in the graph depending on the type of action required at a node. Consider the NFA, or graph, for the regular expression noted above:



In order to evaluate this regular expression, we traverse the NFA, starting at node s_0. If the traversal ever reaches node s_5, the regular expression matches the input string. To traverse from one node to another, we compare the input character to the character on that node's outgoing edge. If the input character matches the edge character, we follow that edge to the next node and move forward one character in the input. If it is not a match, we stop the traversal as the input string is not a match. If the edge character is ϵ, we traverse the edge without advancing the input.

Traversing the graph has one added complication–nodes may have two edges. When walking the graph, if our traversal comes to a node with two outgoing edges, we must follow both edges. That is we try both paths at the same time, reading the input only once. The result of the traversal is the union of both traversals. For example, suppose we are in node s_3, then due to the edge labeled ϵ, we start our multi-node traversal in node s_1 and s_3 simultaneously. If the next input is a 'b', both traversals advance. If the next input is an 'a', the set of active traversals narrows down to a single traversal.

2.1.4 Challenges to Efficient Execution

These applications—and many like them that traverse irregular data structures—face two significant challenges that limit efficient execution:

Fine Grained Parallelism: The programs we described above clearly have a large amount of *coarse-grained* parallelism. For example, simultaneous traversals of a large number of trees can be easily divided between several cores of a multi-core machine. However, these programs also have a significant amount of *fine-grained* parallelism; e.g. every internal node executes a compare. Exploiting fine-grained parallelism (e.g. via SIMD in SSE or GPUs) is complementary to coarse-grained parallelism and can speed each core's processing of its portion of the set of nodes. However, mapping this fine-grained parallelism to SIMD hardware is non-trivial.

Memory Locality: Programs that traverse a large number of irregular data structures likely have no temporal reuse, and poor spatial locality. For example, if nodes of a tree are allocated on the heap, i.e. using memory allocators like *malloc*, there is no assurance as to where a node's children are allocated with respect to the parent. Even if a programmer is

smart and linearizes their irregular data structure into a dense block of memory, high spatial reuse may not occur. Assuming a balanced tree, there are 2^n nodes at the level n of any given tree, among which, only one node will be accessed during any traversal. Thus, once we reach a level where the nodes at the level occupy more than one cache line, we will not see any spatial reuse. As a consequence, it is common for a program that uses irregular data structures to be stalled on memory, which, in turn, negates any possible benefits from parallelism.

2.2 SIMD Traversal of Fine-Grained Tasks

We now focus on the problem of executing irregular applications, like regular expressions or decision trees, on SIMD hardware. When these programs visit an internal node, they optionally perform an *operation* on it, might also including evaluating an *expression* to select the next node to visit. On visiting a leaf node, a different operation is performed (e.g., storing the result), since there is no new node to be visited. We denote the operation(s) performed at a leaf node n as $\mathcal{W}(n)$ and the computations at the non-leaf node as $\mathcal{T}(n)$.

As a concrete example, consider the decision trees in Section 2.1.1 and a typical sequential single decision tree traversal. Each node of this tree is either an internal node or a leaf node. When we traverse to an internal node n, we compare feature values, and branch to the left or right child, depending on the result. When we reach a leaf node n, we update the ranking and terminate the traversal. Thus, for this application, the former is $\mathcal{T}(n)$ and the latter is $\mathcal{W}(n)$.

2.2.1 High Level Approach to SIMD Execution

Now, suppose we want to traverse a *set* of trees on SIMD hardware. We can first execute $\mathcal{T}(n)$ for all root nodes n, which gives us a set of successor nodes BS. Next, assuming all tree traversals are of the same length, we can evaluate $\mathcal{T}(n)$ for all n in ns, and so on, until we reach a leaf node. At a leaf node n, we evaluate $\mathcal{W}(n)$ for all trees.

However, in practice, and unlike a typical array based computation, the different traversals likely have different lengths. Therefore, at a certain level, a mixture of $\mathcal{T}$ and $\mathcal{W}$ computations will be needed. Thus, we must execute each operation type for each stage, and mask the results of operations who's types are not represented by the current operation. What we are doing is essentially emulating MIMD with SIMD, a topic that has been studied in the past [48, 31, 8, 49]. However, none of this work has considered pointer-based traversals.

The second problem is that SIMD execution requires that addressing children is branchless, otherwise we are unable to parallelize the $\mathcal{T}(n)$ expressions. In order to address this problem, we design a layout generation process to organize data structure elements in the memory in a systematic way. We can provide a uniform interface, so that the details of the memory layout are *transparent*, but we must be able to address left and right children of a node with arithmetic operations. Specifically, suppose node has zero, one, or two children. If a node has two children, we store the left child contiguous to the right in memory. This organization works well for SIMD addressing as we can use a simple arithmetic operation to address the left and right children of a node. We require all $\mathcal{T}(n)$ expressions return a 0 to branch to the left child and a -1 to branch right. Thus, for a given node n, if the ns field stores the location of the *left* child, the next node to visit is $ns - \mathcal{T}(n)$. In effect, this turns the addressing of children from a control dependence into a data dependence.

Formalizing this, we can put our approach together as an general method (Algorithm 1).

Algorithm 1 Interpreter ($byte\_codes$, $task\_queue$)

```
1: result = 0
2: ▷ Initialize task_queue by adding root level tasks
3: task_queue = Initialize(roots)
4: for n ∈ task_queue by SIMD-Width do
5:     ▷ Process traversal operations in SIMD
6:     ns = T(bytecodes, n)
7:     ▷ Identify finished traversals
8:     isLeaf = findIsLeaf(ns)
9:     ▷ Strip out finished traversals
10:    ns = streamcompact(ns,isLeaf)
11:    task_queue.push_back(ns)
12:    ▷ Process W operations according to isLeaf
13:    result = W(bytecodes, isLeaf)
14: end for
15: return result
```

Besides the solutions to the two key problems we listed earlier, there are a couple of additional issues that we addressed in this algorithm. In some applications, it is necessary to dynamically *fork*, or start, new fine-grained tasks at a particular step. This operation needs to be (i) efficient and (ii) parallelizable in SIMD. For example, in our regular expression engine, we create a new fine-grained task whenever we traverse an ϵ node in a NFA. To fork a task, we introduce a fork instruction that starts a child fine-grained task at the location of its left child and continues the parent task at the location of the right child.

Further, because not all tasks finish at the same time, we need to remove tasks from processing when they reach a leaf node. Like above, removing tasks must be efficient and not require complicated control-flow. To efficiently remove tasks from processing, we use a data parallel technique called *stream compaction* [18]. We discuss details of this optimization in the next section.

Bytecode	Arg	Type	Description
match	None	$\mathcal{W}(n)$	Found a match; record position; terminate task
nomatch	None	$\mathcal{W}(n)$	Found no match; terminate task
store	`float`	$\mathcal{W}(n)$	Store the arg part of current bytecode to results
cmp	`char/float`	$\mathcal{T}(n)$	Advance PC according to the comparison result
dot	None	$\mathcal{T}(n)$	Advance PC by 1 on any input; if input is null, set PC to nomatch
jmp	`char`	$\mathcal{T}(n)$	Set PC to argument.
fork	`char`	$\mathcal{T}(n)$	Fork a thread: advance parent PC by 1 and set child PC to arg.

Table 2.1: Bytecodes Supported by our Interpreter and their Semantics

2.2.2 A General Solution for Multiple Applications

SIMD parallelization of each individual application following the methodology we described above can be extremely hard. The programmers need to pay attention to a number of details, and can easily write unoptimized and/or even incorrect code. To help development of applications, we have developed an intermediate language and a run-time scheduler or interpreter.

Our solution can be viewed as a *virtual machine*, where instructions from an intermediate language or *bytecodes* are executed on SIMD units. The bytecodes we currently support are listed in **Table 2.1**. Each bytecode is one of the two types: $\mathcal{T}(n)$ and $\mathcal{W}(n)$, representing non-leaf and leaf operations, respectively. Any application that can be implemented using this operation can be mapped to SIMD hardware by our interpreter.

To show the generality of our approach, we have implemented both the decision forest and regular expression matching applications using our interpreter. **Table 2.2** shows the

Nodes Type	Bytecode Sequence
Internal	cmp a;
Leaf	store

Table 2.2: Random Forest Using the Bytecodes

Regular Expression	Bytecode Sequence
$\mathcal{C}(a)$	cmp a;
$\mathcal{C}(.)$	dot;
e_1e_2	$\mathcal{C}(e_1)$; $\mathcal{C}(e_2)$;
$e_1\|e_2$	fork L2; $\mathcal{C}(e_1)$; jmp L3; L2: $\mathcal{C}(e_2)$; L3: ...;
$e?$	fork L2; $\mathcal{C}(e)$; L2: ...;
$e*$	L1: fork L2; $\mathcal{C}(e)$; jmp L1; L2: ...;
$e+$	L1: $\mathcal{C}(e)$; fork L1;

Table 2.3: NFA Regex Using the Bytecode

translation from a tree structure to our Bytecodes from a subset of bytecodes we lised in **Table 2.1**. For SIMD execution for NFA regular expressions, the specific method we use is along the lines of Cox's NFA engine [24], which in turn is based on Thompson's work [129]. This approach has an asymptotic complexity of $O(nm)$ where n is the number of fine-grained tasks and m is the size of the input string. This is far better than a naive NFA interpreter, which can at worst run in $O(n^2)$. **Table 2.3** shows how the implementation handles different cases, using the bytecode from **Table 2.1**.

Now, returning to how our interpreter works, we summarize the sequential and SIMD implementations of our virtual machine in Algorithms 2 and 3, respectively. Algorithm 2 interprets the bytecodes of all trees/graphs level by level sequentially. In each level, it fetches bytecodes indexed by the task queue and processes either a $\mathcal{T}(n)$ operation or a

Algorithm 2 SeqInterpreter ($byte\_codes$, $input$)

```
1: ▷ Initialize the result accumulator and task queue
2: result = 0 {*Accumulator for results*}
3: vector<> clist = Initialize(roots) {*Current list of PCs*}
4: while input != NULL do
5:    ▷ If necessary, advance the input pointer
6:    input += AppShift
7:    vector<> nlist = Initialize(NULL) {*Next list of PCs*}
8:    while (!clist.empty()) do
9:       ▷ Get the bytecode indexed by clist
10:      pc = clist.pop_back() {*Pop a PC to execute*}
11:      op = byte_codes[pc]
12:      ▷ Process T(n) and W(n) operations
13:      if op.type == Bytecode :: cmp then
14:         nextPC = cmp(input, op.arg)
15:         nlist.push_back(nextPC)
16:      else if op.type == Bytecode :: dot then
17:         nlist.push_back(pc + 1)
18:      else if op.type == Bytecode :: jmp then
19:         clist.push_back(op.arg)
20:      else if op.type == Bytecode :: fork then
21:         clist.push_back(pc + 1)
22:         clist.push_back(op.arg)
23:      else if op.type == Bytecode :: match then
24:         result += 1
25:      else if op.type == Bytecode :: nomatch then
26:         result += 0
27:      else if op.type == Bytecode :: store then
28:         result += op.arg
29:         ▷ Jump to return statement
30:      end if
31:   end while
32:   swap(clist, nlist)
33: end while
34: return result
```

$\mathcal{W}(n)$ operation for each tree/graph according to the type of the bytecode. Considering a more general situation that different portions of input may be required for different bytecodes dynamically, and the input pointer may be advanced as line 6 of Algorithm 2, such as Regular Expression application, we maintain two task queues (lists), i.e. $clist$ and $nlist$, in which, $clist$ is to handle the current portion of input, and $nlist$ is to handle the next portion. Especially for applications like Random Forest, the required input index is pre-decided by bytecodes, and we do not need to move the input pointer, so $clist$ and $nlist$ can be simply merged as one task queue. After each iteration, we update either $clist$ or $nlist$ according to the bytecodes, especially, $\mathcal{T}(n)$ operations generate either one or two (task expansion) new tasks, and $\mathcal{W}(n)$ operations generate $zero$ tasks.

The SIMD interpreter described in Algorithm 3 is a SIMD parallel version of Algorithms 2, and a more detailed implementation of the overall method introduced in Algorithm 1. The basic logic of the SIMD execution part (line 14 to line 40) is as follows. We fetch multiple bytecodes indexed by the task queue elements according to the width of SIMD lanes, and then load identical parts of multiple bytecodes into the same SIMD register, such as $highBits$ part identifying the type of bytecodes, and $args$ part storing the address of next PC or output value. We next calculate various flags from types of bytecodes according to the highest bits, to be able to mask invalid results. Finally, we process both $\mathcal{T}(n)$ and $\mathcal{W}(n)$ operations for all SIMD tasks, and strip out the invalid results by the bytecode type flags calculated before. In the last stage, a *stream compaction* operation is used to remove the finished tasks, and, thus, to compact the task queues.

Algorithm 3 SIMDInterpreter ($byte\_codes$, $input$)

```
1: ▷ Initialize the result accumulator and task queue
2: __m128 results = _mm_setzero_ps()
3: results_index = 0
4: clist[] = Initialize(roots)
5: clist_index = Initialize(roots)
6: while (input != NULL) do
7:    input += AppShift
8:    nlist[] = Initialize(NULL)
9:    nlist_index = 0
10:   while (clist_index > 0) do
11:      ▷ Copy clist for task creations
12:      tmplist = Initialize(clist)
13:      tmplist_index = clist
14:      for (i = 0; i < clist_length; i += SIMDWidth) do
15:         ▷ Get bytecodes indexed by clist in parallel
16:         __m128i PCIndexes = SIMDLoadPCIndexes(clist, i)
17:         __m128i ops = SIMDLoadCodes(byte_codes, PCIndexes)
18:         ▷ Get different parts of bytecodes parallel
19:         __m128i args = SIMDLoadArgs(ops) {*Args part*}
20:         __m128i highBits = SIMDLoadHiBits(ops){*High bits*}
21:         ▷ Decide types of ops in current SIMD lane by High bits
22:         __m128i isDot = SIMDcmp(ops, _mm_set1_epix(1))
23:         __m128i isJmp = SIMDcmp(highBits, jmpFlag)
24:         __m128i isFork = SIMDcmp(highBits, forkFlag)
25:         __m128i isMatch = SIMDcmp(ops, _mm_setzero_si128())
26:         __m128i isStore = SIMDcmp(highBits, storeFlag)
27:         ▷ Process T(n) operations and prepare new task queues
28:         ▷ 1. Execute the compare operation
29:         __m128i cmpResults = SIMDcmp(input, args)
30:         ▷ 2. Get addresses of nextPCs by types of operations
31:         __m128i address = SIMDcmp(highBits, ops)
32:         ▷ 3. Mask out the invalid nextPCs by isFork and isJmp
33:         __m128i nextAddress = SIMDcmp(isFork, isJmp, address)
34:         ▷ 4. Strip out finished nlist tasks, store rests to the proper position, advance nlist_index
35:         nlist_index += streamCompaction(cmpResults, isDot, nlist, nlist_index)
36:         ▷ 5. Similar operation on tmplist
37:         tmplist_index += streamCompaction(isJmp, isFork, nextAddress, tmplist,
            tmplist_index)
38:         ▷ Process W(n) operations in parallel
39:         results_index += streamCompaction(isMatch/isStore, results, results_index)
40:      end for
41:      swap(tmplist, clist)
42:      clist_index = tmplist_index
43:   end while
44:   swap(clist, nlist)
45:   clist_index = nlist_index
46: end while
47: return results
```

2.3 Optimizations for Execution Efficiency

We now describe several optimizations that turn out to be critical for achieving efficient execution.

2.3.1 Light-Weight Stream Compaction

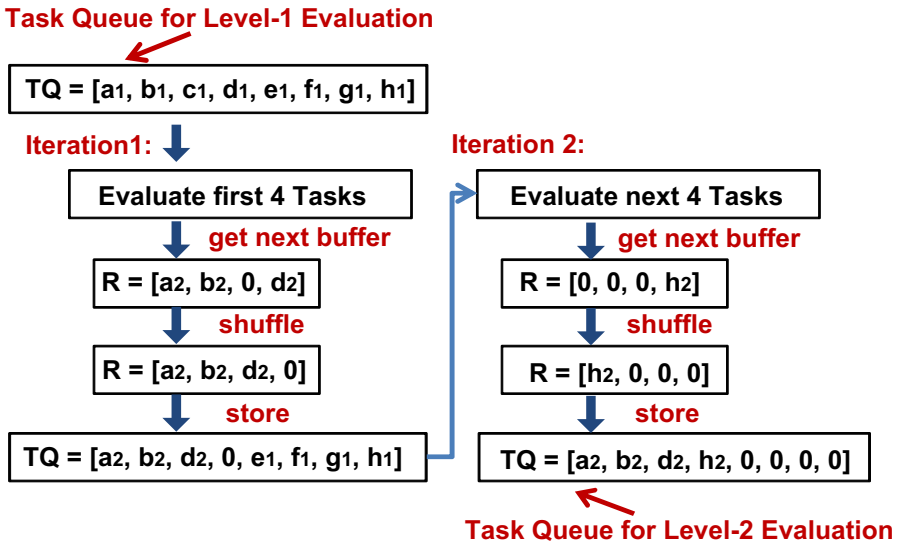


Figure 2.2: An Example of Stream Compaction for SIMD Efficiency

When different SIMD units are processing paths of different length, stream compaction is an important optimization to ensure SIMD efficiency. The basic idea is as follows. Suppose, we start off by needing to process 16 tree traversals concurrently. If the SIMD width is 4, processing the root level requires 4 iterations. Following that, suppose the list of nodes to be processed next is stored in an array of length 16, with a value of 0 denoting that the traversal is over. As an example, assuming that 5 traversals have been completed, whereas

the other 11 traversals are still active, a simple execution mechanism may require 4 iterations to process these 11 traversals, as 4 consecutive values from the array are scheduled in one iteration. A more advanced strategy might be to *compact* the non-zeroes in the array, and use only 3 iterations to process the 11 non-zero entries. This is the idea of *stream compaction* that has been implemented in our framework.

We now explain our implementation with the help of an example, shown in **Figure 2.2**. In this example, at first we have 8 tasks in the initial task queue, and the SIMD lane width allows processing 4 tasks concurrently. Tasks c_1, e_1, f_1 and g_1 are leaf nodes, and after the first iteration of evaluation, we get one 0 in the corresponding position of c_1. Without stream compaction, a *bubble* task left in the SIMD lane undermines the utilization of parallelism. So, we utilize a *shuffle* operation to move the completed *bubble* tasks to the end of the SIMD lane, store the reordered tasks into the beginning position of the new task queue ($store\_position$ initialized as 0), and change $store\_position$ to 3. A similar operation is applied for the second iteration of evaluation, and the new generated tasks are stored into $store\_position$ (3), and the new $store\_position$ is increased to 4. If our application does not require task creation, we may use only one task queue to hold both the old and new tasks, since the number of new generated tasks is always smaller than or equal to the old ones, and it is impossible for the new generated tasks to overlap the unhandled old ones. However, for an application that involves task creation, we need to use two task queues to hold old tasks and new ones respectively, and we swap them at the end of the evaluation of the same level of tasks.

Stream compaction can clearly improve the performance of our method by reducing the number of SIMD evaluation iterations of deeper levels with finished *bubble* tasks. For

example, in **Figure 2.2**, without stream compaction, two iterations are required for level-2 evaluation, while with it, only one iteration is needed. Moreover, an interesting aspect of our implementation is that we can remove tasks from processing *without* complicated control flow (line 10 in Algorithm 1). Thus, we maintain parallel efficiency with only a small scheduling overhead.

2.3.2 Reducing Memory Latency with Intelligent Data Layouts

Effective utilization of fine-grained SIMD parallelism requires that the application not be memory-bound. The naive implementation of the two applications described earlier, as well as other pointer-based applications, can be easily limited by the latency of the memory subsystem. For example, if every node of a tree or a graph is allocated using a library like `malloc`, there is no guarantee of spatial (or temporal) reuse. To address this problem, we introduce several optimized layouts.

Depth First and Breadth First: A Depth First or Breadth First layout linearizes nodes of a tree based on the order they are visited in a depth first or breadth first traversal (e.g. *DF* or *BF* in **Figure 3.1**). However, it turns out that these layouts are not particularly helpful for improving SIMD performance. To illustrate this, we use the notion of *loading distance*. Loading distance gives an intuition as to the amount of data locality among a group of parallel tasks. Specifically, the loading distance is the average distance between the memory locations accessed at the same time, by concurrent threads. Clearly, as the loading distance increases, the possibility of benefiting from spatial reuse or prefetching is reduced. In DF or BF, the average loading distance between any two neighboring lanes is the number of nodes in the first of the two trees. Moreover, on the average, the same

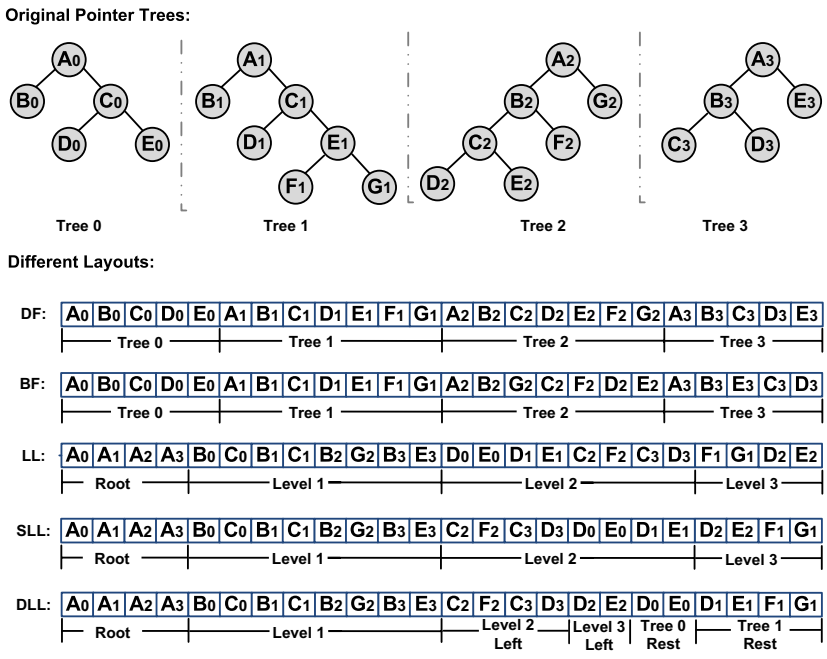


Figure 2.3: Memory Layout with Different Schemes

applies to concurrent evaluation at any level. In the worst case, it implies that each of the four concurrent accesses is a cache-miss.

Level by Level: The prior two layouts linearizes an entire tree before linearizing the next. We next introduce a layout that interleaves nodes across trees, level by level (shown as *LL* in **Figure 3.1**). We co-locate a node's left and right children next to each other in memory. This allows us to use a single pointer to reference both child nodes, thus reducing the size to store the set of irregular data structures.

If we have a complete and balanced binary tree, the loading distance is 2^l where l corresponds to the level of the tree. If the depth of each tree is k, then the loading distance varies between 1 and 2^k. In comparison, the loading distance for the breadth first and depth first layouts are nearly 2^{k+1}. Thus, on the average, the loading distance is reduced, though

more so for the initial levels of the tree than the lower levels. We expect to benefit from spatial locality or the regular strided access pattern (e.g. for prefetching) while traversing through the initial levels of the trees.

Sorted Level by Level: While the LL scheme above has several advantages, it does not utilize any possible bias in the traversal pattern. As stated above, there can be a greater likelihood of visiting one child above the other, and if this bias is known in advance, the more likely child can always be made the left child. The next layout we introduce is called Sorted Level by Level (SLL), and exploits such a bias to decrease the loading distance by a factor of up to two. An example of this layout is shown as the array *SLL* in **Figure 3.1**. In this layout, we divide the nodes of each tree at each level as *left* children and *right* children sets. We allocate the left children together, followed by all right children. By this arrangement, we are expecting a better loading distance within each level when the traversal has a bias.

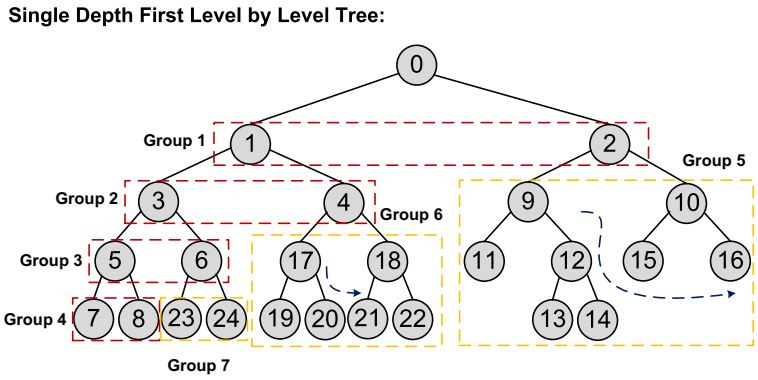


Figure 2.4: Depth First Level by Level Layout for a Single Tree

Depth First Level by Level: The idea of exploiting bias in the SLL scheme can be taken even further through another scheme, which we refer to as the Depth-first Level by Level or DLL scheme. An example of this layout is shown in **Figure 3.1** as *DLL*. In this layout, we also put left child and right child next to each other. So, for the root and the first levels, it is the same as the *LL* and *SLL* layouts. However, from the second level onwards, we focus on exploiting the left bias. At the second level, we allocate the left portions of each tree next to each other, which is the same as *SLL*. However, next, we skip the right part of the second level, and continue to linearize the left part of each tree at the third level. We repeat this process until we finish the left-most parts of all the trees, and then we move to the right part of the second level for each tree. In order to illustrate this idea more clearly, we use another example, with a single but deeper tree (**Figure 2.4**). The numbers in nodes represent the linearization order in memory, if we only have this single tree. The dotted lines are used to organize the nodes into several groups. If we have multiple trees, we will put the groups with the same id of each tree together. Within each group, the order of the nodes is in the Depth-First manner, like the order we have used within this single tree.

2.3.3 Tiling of Trees

In our discussion of the last three layouts, we have assumed that the nodes from all the trees are interleaved. Since only a small number of trees are processed at the same time, it may be more reasonable to interleave nodes from a subset of trees. This is possible through what we refer to as (tiling). If we choose a tile size of N, we interleave all nodes from the first N trees using LL, SLL, or DLL approach, and then repeat the process for each consecutive set of N trees. The tile size can be chosen to obtain the best performance. We will study the impact of tile sizes on performance later.

2.4 Experimental Evaluation

In this section, we evaluate the efficacy of our SIMD interpreter on two applications introduced in Section 2.1—Random Forest and Regular Expressions.

2.4.1 Methods

We had the following two goals in our experiments. First, we want to evaluate the overall speedups obtained on two applications using our general interpretation approach. Second, we want to quantify the benefits obtained from the different optimizations we introduced in the previous section.

Our experiments are conducted on a machine with Intel Xeon E5420 CPU (2.5GHz frequency) with Streaming SIMD Extension 4 (SSE-4). All applications are compiled with the Intel ICC (Intel Parallel Composer 2011) compiler to fully utilize the SSE unit. For all of our experiments, we run the program 30 times; speedup numbers include the mean and 95% confidence interval of the mean.

2.4.2 Overall Speedups from SIMD Parallelization

Random Forests

We compare our method of evaluation against the popular open source numerical analysis and data processing library, ALGLIB, and a random forest implementation that is used in a large Microsoft product. While comparing against ALGLIB, we use four datasets from UCI Machine Learning Repository[39]—Poker, Shuttle, Abalone, and Satellite. While comparing against the Microsoft product, we use production data for that product. In both datasets (UCI and Microsoft), we used the respective libraries to train random forests. We

DateSet	#Tree	#Ave_Node	Path_Leng	Ave_Path_Leng	Bias
Poker	1280	249	4 - 13	7.3	0.51
Shuttle	1280	217	4 - 10	7.5	0.55
Abalone	1280	333	5 - 12	8.0	0.52
Satellite	1280	353	4 - 12	8.2	0.55
Microsoft	3372	239	1 - 45	11.34	0.8

Table 2.4: Summary of Datasets for Random Forest

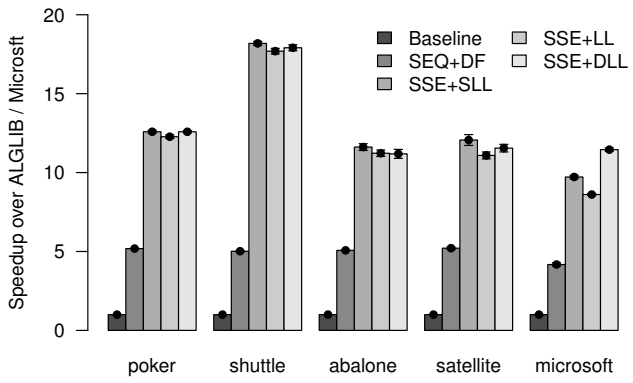


Figure 2.5: Speedup with Our Approach (over Baseline Implementations) - Random Forest

then transform those trees into our SIMD implementation. **Table 3.2** provides a set of descriptive information about the forests we built from these datasets.

The random forests created from the Poker and Abalone dataset result in traversals without any significant bias, those from Shuttle and Satellite have a slight left bias, while the Microsoft dataset has a severe left bias. Lastly, the length of the paths from roots to leaf nodes varies considerably for all datasets (i.e. the trees are not balanced).

Using the SIMD interpreter described earlier, we traverse 4 trees in parallel (one for each lane of the SIMD unit). In **Figure 2.5**, a bar gives the speedup (y-axis) of our approach

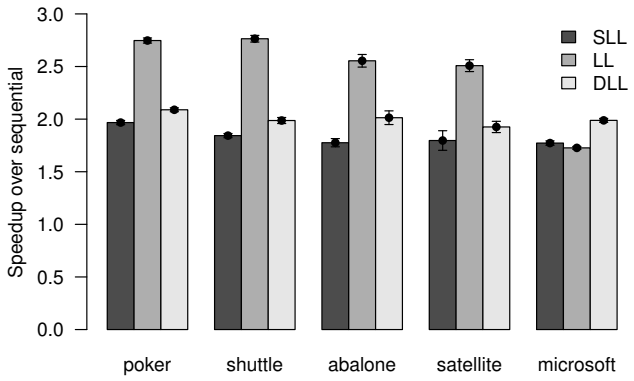


Figure 2.6: SSE Speedups with Different Data Layouts - Random Forest

over ALGLIB and Microsoft, respectively for each of our five datasets. Each dataset has five bars, one bar for each of random forest implementation. The baseline (darkest bar per dataset) is the code that ships with ALGLIB or Microsoft, respectively. The SEQ+DF bar refers to a sequential interpreter evaluated on a depth first layout of the random forest nodes. The other three bars per dataset refer to the SIMD interpreter run on different data layouts. We do not show the performance of SSE + Depth First and Breadth First versions, as they quickly become memory bound and do not allow much benefit from SSE units.

From **Figure 2.5**, we can see that by our transformed dense layouts and SIMD optimized interpreter, we can gain more than 10 times speedup over the baseline implementations on all 5 datasets. We include the SEQ+DF implementation because we are interested in showing how much speedup we get from SIMD after linearization; on the UCI datasets SIMD increases performance by a factor of 3, while on the Microsoft dataset SIMD increases performance by a factor of more than 2.

To understand the performance impact of our SIMD interpreter we compare the runtime of a sequential interpreter against a SIMD one, holding the layout constant in this

comparison. We show the results of this experiment in **Figure 2.6**. A bar on this graph shows the speedup (y-axis) of our SIMD interpreter over the sequential interpreter on the same layout (*SLL*, *LL*, and *DLL*) for each dataset (x-axis). The speedups from SIMD (with 4 SIMD lanes) range between 2 and 2.8.

Regular Expression Matching

We now investigate the performance of our SIMD interpreter on regular expression matching. For this application, we use a simple level-by-level (LL) layout because the graphs generated by our regular expressions small fit easily in L1 so memory optimizations are not as important as in the random forests application.

We compare our approach to GNU `grep`, which is chosen for two reasons. First, like our regular expression engine, it counts matches and matches regular expressions from the POSIX Extended Regular Expression syntax. Second, GNU `grep` is known to be fast.[6]

We search the King James Bible for up to 10 different regular expressions. Each regular expression follows the pattern $.*ab$, where the characters a and b are unique for each regular expression. To match N regular expressions, we combine them using the choice operator. Note that because we can pack instructions into a `byte`, our SIMD interpreter can traverse up to 16 graphs in parallel for this application.

Figure 2.7 shows the speedup of our approach. A bar on this graph (x,y) gives the speedup over GNU `grep` (y), varying the number of regular expressions, or fine-grained tasks, executed. GNU `grep` at 1.0 is the baseline. It is very fast for the first two regular expressions, as it uses Boyer-Moore to perform a sub-linear search over the input string.

[6]In a recent post to the freebsd mailing list, entitled "Why GNU grep is Fast", the author of GNU `grep` describes why his implementation is fast; GNU `grep` uses the Boyer-Moore[98] algorithm for sub-linear search. It also uses a DFA based graph traversal once it finds a position in the input string to match against text.

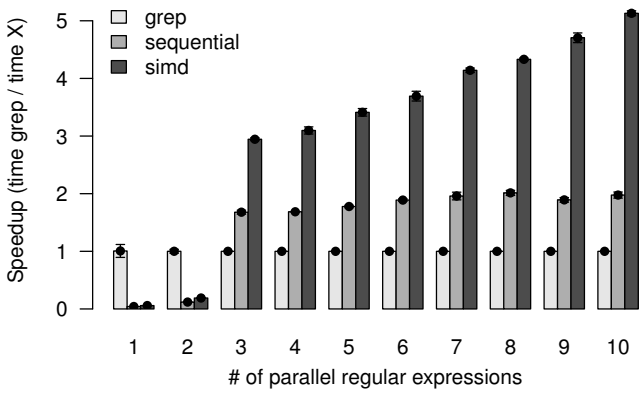


Figure 2.7: Speedup of the SIMD interpreter over GNU `grep` - Regular Expressions

However, after 3 or more regular expressions, GNU `grep` cannot use Boyer-Moore as the resulting regular expression gets too complicated. After 3 parallel regular expressions, the sequential interpreter is 1.7X faster than GNU `grep`. This is due to the regular level-by-level access pattern of our interpreter. However, the speedup for the SIMD interpreter linearly increases as we add fine-grained tasks. The SIMD interpreter is anywhere from 3X to 5X faster when searching for three or more parallel regular expressions.

2.4.3 Benefits from Optimizations

The speedups we reported above are made possible due to a number of optimizations we have implemented. Using one of the two applications (random forest), we now quantify the gains from each of the optimizations.

Improvements from Stream Compaction

Figure 2.8 shows the comparison of execution times among the SIMD code with and without stream compaction, for each of the five datasets. The results show that for datasets

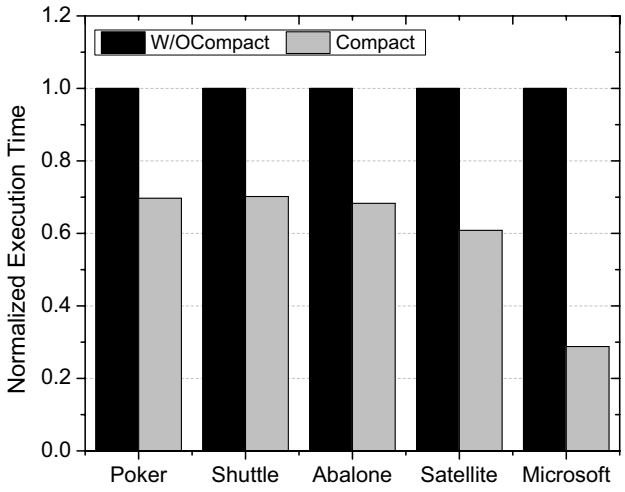


Figure 2.8: Speedup Improvements from Stream Compaction - Random Forest Using 5 Datasets

with a smaller variation in path lengths, such as Poker and Shuttle, the stream compaction method gives around 30% speedup over the unoptimized version. For the dataset that has a larger variation in path lengths, i.e. the Microsoft dataset, stream compaction gives more than 70% speedup.

To further study the reasons for these speedup, in **Figure 2.9**, we show the workload reduction by the stream compaction method, using two representative datasets, Poker and Microsoft. Specifically, the Poker represents the case with a smaller variation in path lengths, whereas, Microsoft involves a much larger variation in path lengths. x-axis here is the evaluation level, and the y-axis is the cumulative number of SIMD evaluation iterations, i.e., the workload on the SIMD lanes. We can see that for Poker dataset, our stream compaction method is able to reduce around 40% of the workload, with most gains seen from levels 8 through 12. For the Microsoft dataset, the benefits are seen even at earlier levels of the tree, and overall, add up to 80% of the number of iterations needed. By comparing the workload

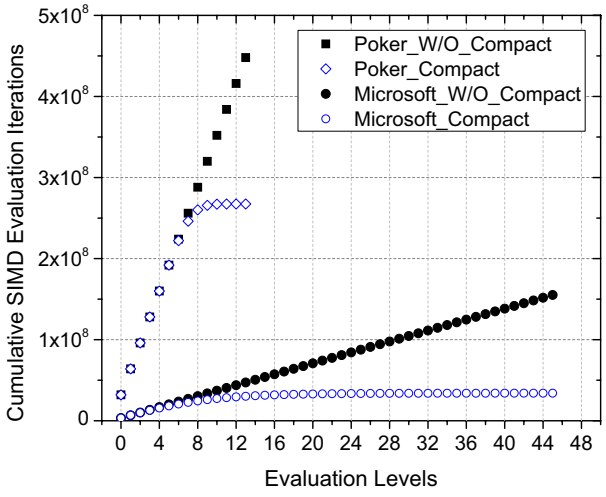


Figure 2.9: Reduction in Workload from Stream Compaction: Poker and Microsoft Datasets

reduction (80% and 40%) and the execution time reduction (70% and 30%), it appears that nearly a 10% scheduling overhead is introduced by our stream compaction method.

Detailed Examination of Benefits from Optimized Layouts

In this section, we further study the performance impact of our intelligent layouts. As shown in **Figure 2.6**, by our intelligent layouts, we can gain 2.0-2.8 times speedup. However, the speedups from depth-first layout were only between 1.2 and 1.5. This shows that our layouts provide better locality, and therefore, reduce the impact of memory latency.

In order to study the underlying reason as to how our intelligent layouts hide memory latency, we conduct a profiling experiment. We create 1000 balanced, unbiased trees with a varying number of nodes per tree (changing the depth of each tree from 1 up to 13). Now, using different layouts, **Figure 2.10** examines the percent of time the backend of the microprocessor is stalled. We can see that the microprocessors are often stalled. The plot has five lines, one for each of our layouts; a point on this graph (x,y) gives the amount of

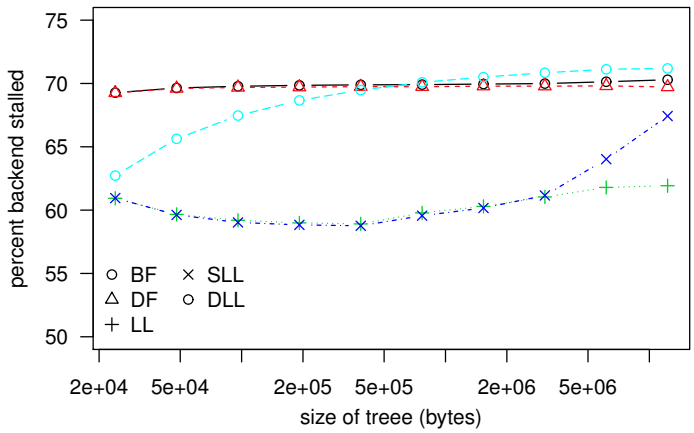


Figure 2.10: Percent of Time Backend is Stalled (Function of the Tree Size, for Different Layouts)

time the backend is stalled (y) as we change the size of the tree (reported as the number of bytes it takes in memory (x)). The *DF* and *BF* layouts spend about 70% of the time stalled on memory references, irrespective of tree size, since these layouts lack spatial locality. In contrast, even when the size of trees is larger than the L2 cache size on our processor (8MB), the *LL* layout is able to keep the processor working about 40% of the time, since the *LL* layout has predictable memory system pattern, and as a result, the hardware prefetchers are able to predict memory access so that there are fewer L2 data cache misses.

In both benchmarks, the *SLL* and *DLL* layouts do not perform as well as the *LL* layout; this is expected as *SLL* and *DLL* are optimized for *biased* layouts. So we redid these experiments but with 80% left bias (not shown due to space). As a result we see *SLL* and *DLL* are significantly better than *LL* (percent the backend is stalled drops to 54%). Especially, with severe biased and imbalanced tree access, *DLL* shows much better performance than both *LL* and *SLL*.

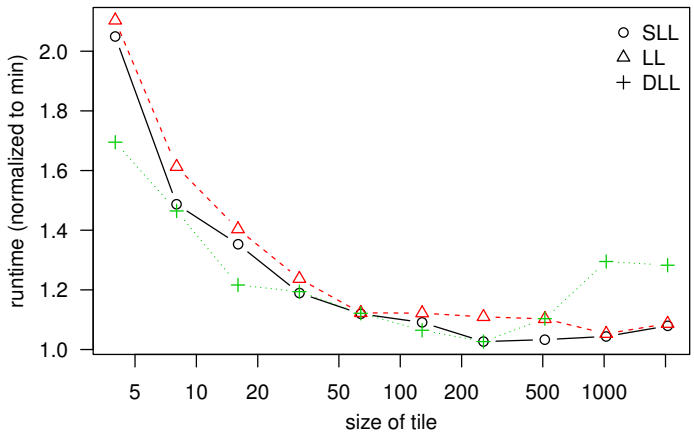


Figure 2.11: Benefits of Tiling (Poker Dataset)

If we compare the three level-by-level layouts in **Figure 2.5** we see that in biased datasets (i.e. Shuttle, Satellite, and Microsoft), SLL and DLL both show better performance than the LL layout. The improvement in performance with SLL and DLL is consistent with what we discussed above. The benefit of DLL is further confirmed by the results from the Microsoft test case, where there is most imbalance and bias. The DLL version now has the best performance, outperforming SLL by about 15%, and LL by nearly 25%. For the cases with only a moderate bias, i.e. Shuttle and Satellite, SLL and DLL are both 5% better than LL.

Impact of Tiling

In this section we evaluate the impact of the tile size on performance. A point (x,y) on **Figure 2.11** shows the execution time (y-axis), normalized to the minimum execution time for all tile sizes, as we change the number of trees per tile (x-axis). We show three lines, one line per level-by-level layout. We use only the Poker dataset as the results generalize to other datasets. We can see that when there is no tiling (i.e. 1 tree per *tile*) or when the

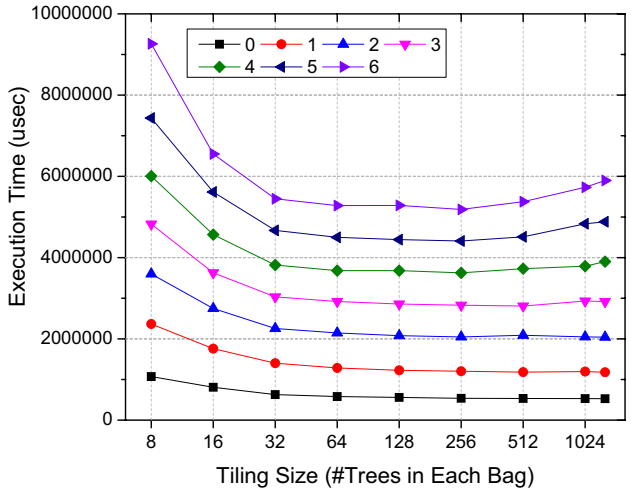


Figure 2.12: Execution Time with Changing Tree Levels and Tiling Sizes - SSE + DLL on Poker

tile has only a small number of trees, the performance is up to a factor of 2.0 worse over the cases when between 50 and 500 trees are put in a single bag or a tile.

In order to explain this behavior, we carefully observed the relationship between the evaluation time for the different tree levels, as a function of tile sizes. We investigated the run time as a function of tile size for the *DLL* layout on the Poker dataset (**Figure 2.12**). A point on this graph (x,y) shows execution time in milliseconds (y-axis) as a function of tile size (x-axis). Each line provides the amount of time required to execute all trees up to a certain depth (inclusive running time). For example, the line corresponding to 0 implies that we only evaluate the root nodes, 1 implies that we evaluate the root nodes and the nodes in the first level, and so on.

At the lower levels, the performance improves slightly as we increase the number of trees in the tile. This is because the set of memory addresses accessed follow a regular pattern when we process the same level for a larger number of distinct trees. The same

regularity is not seen when we start processing other levels of the trees. Such regularity, achieved with a larger tile size, helps achieve better prefetching, and hence, better performance. At the initial levels of the tree traversal, there is no loss of performance as we continue to increase the tile size, though there are not too many gains either after a size of 64.

With lower levels of the tree, and with DLL layout, there is also a reduction in performance when the tile size becomes very large. This is because the possibility of exploiting spatial locality across consecutive accesses to nodes of the same tree decreases with increasing tile size. Recall from our earlier discussion that main advantage of DLL is exploiting such locality, for biased traversals on imbalanced trees. This advantage is lost with a very large tile size. In comparison, there is hardly any change in performance for SLL and LL layouts, as we continue to increase the tile size.

2.5 Related Work

Earlier work had used very sophisticated compiler analysis to automatically determine parallelism in pointer-based programs [44]. More recently, the Galois project has extensively considered parallelization of irregular applications [77, 93]. Their focus is coarse-grained or MIMD parallelism, while our focus is SIMD execution.

There are also many efforts focusing on manual optimization of this class of applications on SIMD and vector units. Key recent efforts include the work by Sewall *et al.* [124] and Kim *et al.* [70]. This work considers simultaneously processing multiple inputs on a single data structure. We are focusing on processing one input point across multiple pointer-based data structures, and focus on a more general interpretation system.

Execution of tree and graph traversal algorithms on SIMD (mostly GPU) has been a popular topic in recent years. As GPGPUs were emerging, Harish and Narayannan [50] designed a set of algorithms to map graph algorithms to the GPU architecture. More recently, many others have worked on this problem with many efforts focusing on breadth-first traversals [88, 1, 55, 96], which is a key kernel in many applications, and others focusing on single source shortest path or other interesting graph algorithms [29, 127]. Our work is distinct in several ways. First, we consider traversals over a collection of trees, which leads to a different set of challenges for memory locality. Second, our focus is on performance portability, which hasn't been the topic of prior studies.

More closely related to the class of applications we study in our work, tree forest applications like decision trees and suffix trees have also been studied on GPUs [125, 123, 130]. Especially, Sharp [125] has parallelized decision tree and forest traversal on GPUs. The work is based on using a GPU's texture memory and does not apply to the SSE units we have considered. Moreover, none of them have carefully studied the effect of different memory organizations architecture, i.e. they usually implement one, specific layout. In addition, this prior work does not use any form of analytical modeling to achieve performance portability. Similarly, regular expression traversal has been implemented on GPUs [134] and Cell processor [122]. Cascarano *et al.* [13] also designed an NFA based regular expression engine focusing on GPUs architecture, which has been further improved by Zu *et al.* [146]. Our work is distinct in considering SSE parallelism and locality issues related to modern uni-processors. Prior to the interest in SIMD or many-core execution, many efforts focused on vectorization of pointer-based applications. Lars *et al.* [79] and Junichiro *et al.* [69] vectorized tree traversals, but considered only a single tree.

Another class of irregular applications involve sparse matrices and/or indirection arrays. SIMD and GPU parallelization and optimization of these applications has been studied in recent years. For example, Kim and Han [72] propose a code generation method to vectorize indirection of array-based loops and Zhang *et al.* [140] design a set of strategies to optimize such irregular applications on the GPUs architecture.

Processing of MIMD tasks on SIMD machines has received considerable attention in the past. For example, Hanxleden and Kennedy [48] developed loop transformation techniques (focused on array based programs) to achieve this goal. Prins and Palmer [112] had a similar focus, but targeted vectorization. Dietz and Cohen described a more general scheme [31]. Blelloch *et al.*[8] and Hardwick[49] have focused on exploiting nested data parallelism, similar in spirit to our use of data parallelism to handle irregular applications. Our work has considered specific challenges arising for pointer-based traversals, which have not been considered in the past. We have also developed optimizations that are critical for performance on today's processors (e.g. locality, as more applications have become memory bound over time).

2.6 Summary

This work shows how to extract SIMD parallelism from applications that traverse irregular data structures such as trees and graphs. As SIMD execution units become more common and capable in the near future, it becomes increasingly pressing to find general techniques to exploit the power of this hardware in new and broader contexts. This work describes one such approach, which is to traverse and compute on multiple, independent, irregular data structures in parallel using a targeted virtual machine running on a SIMD

vector processor. By scheduling operations from the virtual machine and implementing a number of optimizations, we have shown substantial speedups from two applications.

Chapter 3: A Portable Data Locality Optimization Engine to Accelerate Irregular Data Traversals on Various SIMD Architectures

The second challenge of fine-grained parallel irregular data structures traversals on SIMD architectures is to improve the memory locality. Our work not only considers the memory locality of irregular traversals on a specific SIMD architecture, but also puts this topic into the overall background of *performance portability*, and considers the automatic ways to improve the memory locality across multiple SIMD architectures with distinct memory hierarchies.

In this Chapter, we first comprehensively study the intelligent data layouts presented in Chapter 2 from another perspective—*inter-thread* data locality and *intra-thread* data locality; and based on this study, we next propose a cache analysis model for automatically selecting memory layout on various SIMD architectures according to their respective configurations; and in the end, we evaluate our model by two irregular traversal applications mentioned in Section 2.1.

3.1 Intelligent Data Layouts

A naive implementation of the two applications, as well as other pointer-based applications, can be easily limited by the latency of the memory subsystem. For example, if every node of a tree or a graph is allocated using a library like `malloc`, there will likely be no

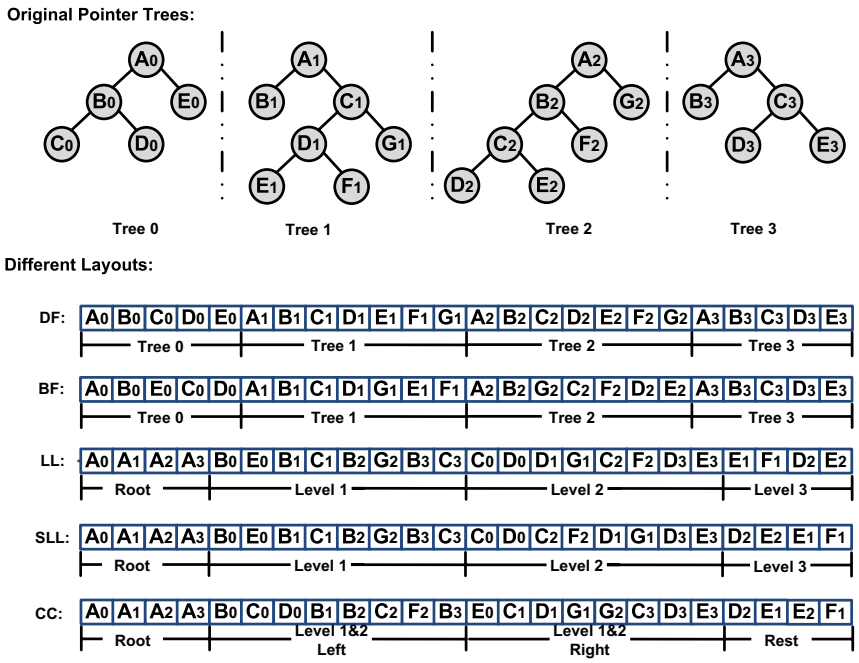


Figure 3.1: Memory Layout with Different Schemes

spatial reuse, in addition to the fact that there is no temporal reuse in the application. Thus, improving memory efficiency is critical toward obtaining benefits from SIMD parallelize.

In this section we present a number of layouts that not only *linearize* pointer-based irregular data structures into dense arrays, but improve spatial locality as well. The goals here include co-locating nodes (in the same cache line) that are likely to be accessed concurrently by different threads (*inter-thread locality*) and/or co-locating threads that are accessed in successive steps by the same thread (*intra-thread spatial locality*).

First, we note that linearization also compresses a data structure. For example, a simple depth first or breadth first layout linearizes nodes of a tree based on the order they are visited in a depth first or breadth first traversal (e.g. *DF* or *BF* in **Figure 3.1**). By linearizing in

DF or BF layout, we can reduce the size of the irregular data structures; instead of two pointers, we only require a single pointer for each node in both linearized arrays.

However, both DF and BF linearize each tree independently of other trees and cannot improve inter-thread locality. While discussing various layouts, we use the notion of *loading distance* to measure inter-thread data locality. Specifically, the loading distance is the average distance between memory locations accessed at the same time, by consecutive threads. Clearly, as the loading distance *increases*, inter-thread data locality *decreases*. In DF or BF, the average loading distance between any two neighboring lanes is the number of nodes in the first of the two trees.

3.1.1 Improving Inter-Thread Locality

To improve inter-thread locality, we introduce a layout that interleaves nodes across trees (level by level, shown as *LL* in **Figure 3.1**). To be concrete, we first lay out the root nodes—the first tree's root node is followed by the second tree's root node, followed by the third, and so on. Then, we move on the the next level. We co-locate a node's left and right children next to each other in memory. This allows us to use a single pointer to reference both child nodes, thus reducing the size to store the set of irregular data structures.

Inter-thread locality improvements through the use of this method is easy to see. For example, at the root level, nodes have large spatial locality; when we access the first tree's root node, we pull in the second tree's root node. If we have a complete and balanced binary tree, the loading distance is 2^l where l corresponds to the depth of the tree. If the depth of each tree is k, then the loading distance varies between 1 and 2^k. In comparison, the loading distance for the breadth first and depth first layouts are nearly 2^{k+1}. Thus, on

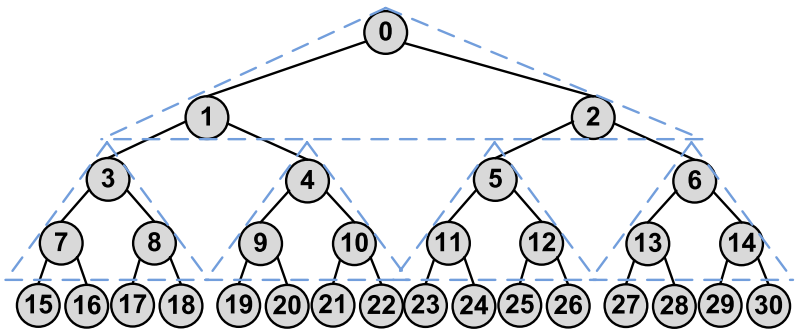


Figure 3.2: Cache Conscious Layout for a Single Tree Structure

the average, the loading distance is reduced, though more so for the initial levels of the tree than the lower levels.

While the LL scheme above has several advantages, it does not utilize any possible bias in the traversal pattern. In many applications, there can be a greater likelihood of visiting one child above the other, and if this bias is known in advance, the more likely child can always be made the left child.

The next layout we introduce is called Sorted Level by Level (SLL), and exploits such a bias to decrease the loading distance by a factor of up to two. An example of this layout is shown as the array *SLL* in **Figure 3.1**. In this layout, we divide the nodes of each tree at each level as *left* children and *right* children sets. We allocate the left children together, followed by all right children. When a tree is biased, we realize a reduced loading distance for each level of the tree.

3.1.2 Improving Intra-Thread Locality

Improve cache performance of pointer-traversing applications has been studied in the context of single-threaded programs [21, 114], with the main outcome being a Cache Conscious (CC) data layout. In the CC layout, we partition a single tree into blocks according to the size of L1 or L2 cache line. For example, in **Figure 3.2**, if the cache line size is 16 Bytes, and each tree node takes 4 Bytes, we can put 3 nodes together into a single block(e.g. the dotted triangles). In each block, when the parent node is fetched into the cache, a cache hit occurs whether the left or the right child is accessed next, leading to spatial reuse.

However, for multi-threaded or SIMD memory accesses, traditional CC layout does not work well, as it completely ignores inter-thread spatial reuse. If we have multiple trees organized with the CC layout, the loading distance for each level is still the entire tree. In our work, we slightly modify the traditional CC layout by organizing all root nodes next to each other by LL layout, since it is obvious to improve the memory performance on SIMD architecture. We also apply SLL strategy to organize the cache conscious blocks into separated left and right groups. We show our CC layout in **Figure 3.1** and use this version in our experiments.

3.1.3 Hybrid Layout

From our discussion above, we can observe that while processing of the earlier levels of the tree gives opportunity for significant inter-thread spatial reuse, the loading distance increases beyond the size of a cache line after a certain level, and only intra-thread locality can be exploited. Based on this observation, we design a hybrid layout schema to combine the benefit of inter-thread data locality of SLL for top levels, and the benefit of

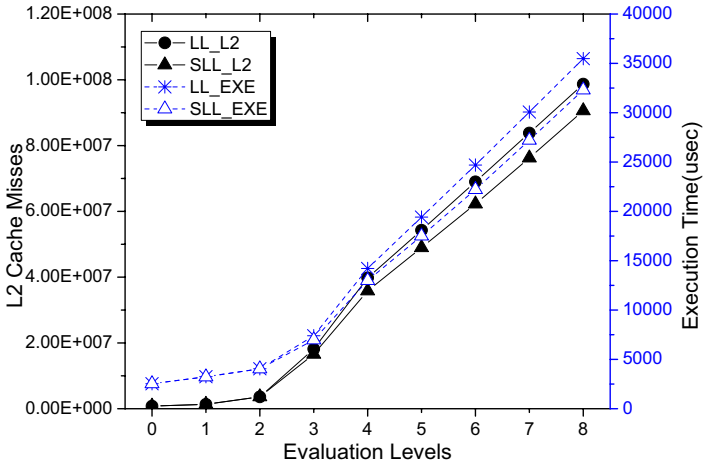


Figure 3.3: Comparison of Last Level (L2) Cache Misses and Execution Time for LL and SLL layouts

intra-thread data locality of CC layout for deeper levels. The hybrid layout is parameterized by a *switching parameter*, which denotes the level at we shift from SLL layout to CC layout.

To further validate our reasoning about relative performance of difference layouts and the motivation behind the hybrid layout, we conducted several experiments. We take the the B+ Tree application on Fermi architecture as an example to examine the cache behavior of different layouts. The last level (L2) cache misses and real execution times of different layouts are shown in **Figure 3.3** and **Figure 3.4**. Profiling data is collected using CUDA Visual Profiler 4.1. Solid lines show the L2 cache misses, and dotted lines show the actual execution time of various versions. By comparing **Figure 3.3** and **Figure 3.4**, we can see that for top levels, the last level cache misses of LL and SLL layouts are lower than those with the CC layout. However, they increase rapidly, and starting from a certain level, the CC layout outperforms LL and SLL layouts. Further, we show the L2 cache misses and

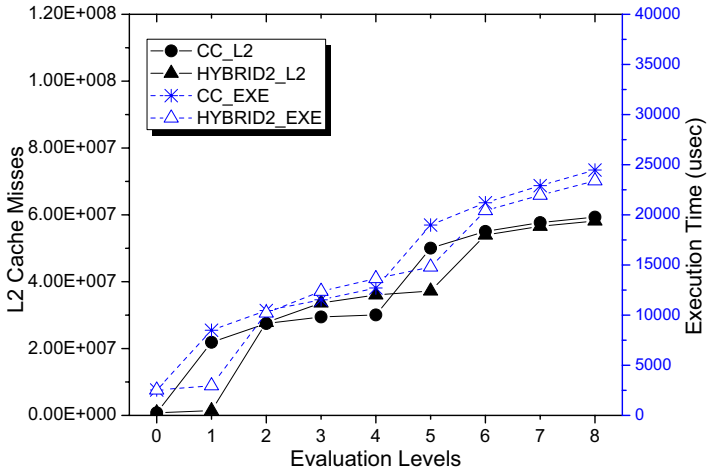


Figure 3.4: Comparison of Last Level (L2) Cache Misses and Execution Time for CC and HYBRID Layouts

execution time of the hybrid layout in **Figure 3.4**. We can see that the overall performance of HYBRID layout is better than both SLL and CC layouts.

3.2 Cache Analysis Model for Automatic Selection of Layout

For a given application and architecture, selecting the best layout from among the ones we introduced in the previous section is a challenging problem, yet automating the selection is critical for *performance portability*. Application parameters like the number of bytes needed for one node of the tree, possible bias in traversing one child of the tree over others, and whether all trees are accessed with equal probability or not, can all impact the choice of the layout to use. Similarly, architectural characteristics, like the size of a cache line, cache miss penalties, and degree of SIMD parallelism can impact how one layout may result in better performance over another.

In general, modeling computer systems and predicting performance of a given application on a given architecture is very hard. However, by focusing on a restricted domain, we

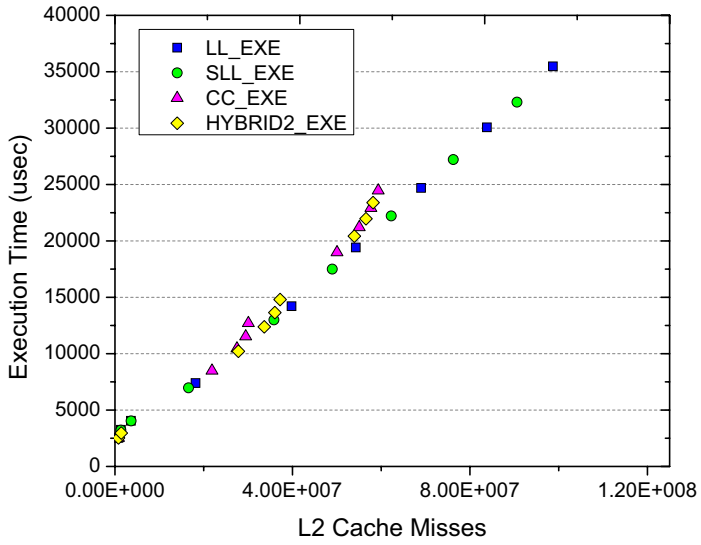


Figure 3.5: Correlation between Last Level (L2) Cache Misses and Execution Time (Different Layouts)

can simplify the problem. For our target class of applications, the number of L2 (and L1) cache misses is an effective predictor of execution time. **Figure 3.5** shows the relationship between L2 cache misses and execution times, with the use of different layouts. We can see a very high correlation, even across different layouts. More detailed profiling data (not included here) further shows that L1 cache misses can also play an important role. Based on these observations, our model captures L1 and L2 cache misses.

To understand the key insight behind our model, suppose that the tree nodes are organized by the LL layout. As we traverse through a number of trees, for the root (zeroth) level, one load from main memory is sufficient to bring in many trees (as many as fit in a cache line). Thus, the *loading efficiency* is 1. For the first level of the tree, we will use only 50% of the elements we have loaded, i.e. the loading efficiency is $1/2$. Similarly, for the second level, the efficiency is $1/4$. If only 4 tree nodes can be stored in one cache line,

Param	Type	Explanation	Value
N	App	No. of Levels of Each Tree	B+ Tree $N = 9$
B	App	# of trees to evaluate	32
T_1	Arch	L1 cache access latency	Assume 1 clock cycle
T_2	Arch	L2 cache access latency (L1 cache miss penalty)	Assume $T_2 = 4T_1$
T_M	Arch	Memory access latency (L2 cache miss penalty)	Assume $T_M = 8T_1$
L	Both	L1 and L2 cache line size, in # of tree nodes;	$Node\_size$ = 8 Bytes, GPU: L = $128/8$ = 16, SSE: L = $64/8$ = 8
G	Both	In CC layout, the # of levels of nodes can be held by one cache block. $G = \log_2 L$	GPU: $G = \log_2 16 = 4$, SSE: $G = \log_2 8 = 3$
x	User	*Switch Level*, i.e., from this level, we start to use the CC layout	

Table 3.1: Model Parameters

the loading efficiency remains $1/4$ for subsequent levels. However, a different choice of layout, any bias (e.g. to left or right children), can all complicate the calculations of cache misses.

3.2.1 Parameters and Assumptions

Parameters: The parameters used in this model are explained in Table 3.1. The GPU here implies Fermi, the Tesla-10 series architecture which does not have L1 or L2 cache will be explained separately later. L is the cache line size in terms of the number of tree nodes, and we load L nodes into L1/L2 cache in each access. G is a parameter used for the CC layout. Recall that in the CC layout, we partition a single tree into blocks with L nodes per block in a triangular fashion. G, which is also $\log_2 L$, indicates the number of levels of nodes that fit in each block. For example, if $L = 16$, we group $G = \log_2 (16) = 4$ levels of nodes into a block. The parameter G has another significance for the LL layout. Considering root

as the level 0, G is the level at which the loading efficiency decreases to $1/L$, and cannot reduce any further.

Assumptions: All data is in main memory initially and both L1 and L2 cache are empty. The tree data is too big to fit into either L1 or L2 cache, or even a combination of shared memory (on GPUs) and the cache. There is no temporal reuse while executing application once. We also assume that the tree is perfectly balanced. Furthermore, the detailed calculations assume a binary tree, though only a trivial modification is needed to capture a general k-ary tree.

Initially, we focus on the hybrid layout, i.e. we use either the LL or SLL layouts to exploit inter-thread data locality for upper levels of the tree, and use CC layout to explore intra-thread data locality for the lower levels, switching at level x. Assuming that there is no bias in accessing left/right child and all trees are accessed with the same probability, our objective will be to find the proper switch level, x.

3.2.2 Basic Model for Balanced Accesses

Our first observation is that while using the hybrid layout with LL at initial levels, the switch to the CC layout must be made latest by the level G. This is because there will not be any spatial reuse with LL layout level G onward, i.e. only one node from a cache line will be read.

The memory access times for processing B trees, as a function of the switch level x, is:

$$T(x) = (T_1 + T_2 + T_M) \times \frac{B}{L} \times (2^x - 1) + [T_1 \times (N - x) + \lceil \frac{(N - x)}{G} \rceil \times (T_2 + T_M)] \times B \quad (3.1)$$

The two terms above capture the memory access times for up to the level $x - 1$ and levels x through N, respectively. For the part of the trees that are organized using LL

layout, we have a total of $2^x - 1$ nodes for each tree, or $B \times (2^x - 1)$ nodes for B trees we are processing, which correspond to $B/L \times (2^x - 1)$ cache lines. While using the CC layout, recall that G levels fit in one cache line, which means that there is a G times reuse of a single cache line. Thus, the number of cache misses for processing the last $N - x$ levels is $\lceil \frac{(N-x)}{G} \rceil$.

In order to get the value of x where this term is minimized, we take a derivative, resulting in

$$T'(x) = (T_1 + T_2 + T_M) \times \frac{2^x \ln 2 \times B}{L} - [T_1 + \frac{1}{G} \times (T_2 + T_M)] \times B \tag{3.2}$$

For $T'(x)$ in Equation (3.2) to be zero, we need

$$x = \log_2 \frac{L(T_1 + \frac{1}{G} \times (T_2 + T_M))}{\ln 2 \times \ (T_1 + T_2 + T_M)} \tag{3.3}$$

Using SLL for Upper Levels We now show how the analysis above can be extended to the SLL layout. Recall that in SLL layout, we store the left and right parts of the trees separately. Thus, depending upon the traversal, it may be possible to better exploit inter-thread data locality for more levels. Our model formally captures this through a *deferring factor*, which we empirically determine in an architecture-independent fashion. Using λ to denote such an empirically determined deferring factor, we modify the execution time expression for LL as follows:

$$T(x) = (T_1 + T_2 + T_M) \times \frac{B}{L} \times (2^{x-\lambda} - 1 + \lambda) + [T_1 \times (N - x) + \lceil \frac{(N - x)}{G} \rceil \times (T_2 + T_M)] \times B \tag{3.4}$$

Similarly, by taking the derivative of T in terms of x, and checking when it becomes zero, we have

$$x = \log_2 \frac{L(T_1 + \frac{1}{G} \times (T_2 + T_M))}{\ln 2 \times (T_1 + T_2 + T_M)} + \lambda \tag{3.5}$$

Equation (3.5) shows λ also affects the split level. By analysis of performance with datasets that have bias, we have found that $\lambda = 1$ is quite effective so we use this value in all of our experiments.

3.2.3 Capturing Biased Accesses

In many cases, probability of accessing different children of any node is different. Without loss of generality, we assume that the child that is more likely to be accessed is organized to the left.

We introduce a bias metric β to indicate the percentage of inputs falling into the left most path of all trees. Then, if $T1(x)$ (defined shortly), is the time spent when an input involves all left-most paths and if T2(x) is the time spent for all other inputs, we have

$$T(x) \ = \ T2(x) \times (1 - \beta) + T1(x) \times \beta \tag{3.6}$$

$T2$ can be calculated using Equation (3.4). For calculating $T1$, we perform the following analysis. If all queries fall into the left most path of each tree, the SLL layout is preferred and then the loading time of a bag of trees is:

$$T(x) = (T_1 + T_2 + T_M) \times \frac{B}{L} \times (1 + 2(x - 1)) + [T_1 \times (N - x) + \lceil \frac{(N - x)}{G} \rceil \times (T_2 + T_M)] \times B \tag{3.7}$$

An analysis of the above expression shows that when $x = N$, $T1(x)$ is the minimum. We choose x to find the minimum value, where the weighted sum of $T1(x)$ and $T2(x)$ is

minimized. Intuitively, we can see if there is more bias, we can better exploit inter thread locality. Thus, either a SLL layout or a hybrid layout with SLL at the top several levels is optimal.

β is an architecture independent parameter, which depends on the application, and more particularly, the dataset. To obtain the value of β, we employed the following strategy. First, we randomly sample 5% of the input data. We process these inputs and calculate the probability of reaching each leaf node (denoted as P_i for the leaf node i). We then estimate β as the sum of the probability of reaching the K left-most leaf nodes, i.e, $\beta = \sum_{i=1}^{K} P_i$.

3.2.4 Impact of Sparse Buckets Accesses

As we discussed previously, in an application like B+-Tree Forest, each tree can have an imbalanced number of inputs in that tree's bucket. This situation, which we refer to as sparse buckets accesses, can change the relative performance if we use different layouts.

Let us revisit Equation (3.1) and see how to modify it to handle this particular situation. If the probability of accessing any particular tree is high, each of the blocks at the upper level still need to be loaded into memory. However, at the lower levels, when we are using the CC layout, not all blocks will be loaded. Thus, we can simply apply a scaling factor θ, $\theta < 1$, to the second part of Equation (3.1). Analyzing when this expression is minimized, we now get the value of x as $\log_2 \frac{L(T_1+\frac{1}{G}\times(T_2+T_M))\times\theta}{\ln 2\times\ (T_1+T_2+T_M)}$. Thus, it is preferable to focus on intra-thread spatial reuse starting with even earlier levels.

To summarize, θ is an dataset dependent parameter, which we can estimate by randomly sampling from said dataset.

3.2.5 Modeling a System Without L1/L2 Cache

Our discussion so far has assumed presence of an L1 and L2 cache. An architecture like that of the Tesla 10-series GPU does not have L1 or L2 cache, though it does have support for *coalesced accesses*, i.e. simultaneous access to consecutive memory locations by different threads are faster than random accesses.

Our model can also be applied to such an architecture with small modifications. Particularly, we set $G = 1$, and L is used to show the coalesced access block size. By applying similar calculations to Equation (3.1) and Equation (3.2), we get $T'(x) = 0$ when $x > \log_2 L$ ($\log_L$ is the last level where it is possible to utilize coalesced accesses for LL layout). This implies that we can only utilize inter-thread data locality for such an architecture, and our hybrid layout does not provide any further benefits for deeper levels. Furthermore, with SLL layout, it is possible to further exploit the inter-thread data locality for a few additional levels.

3.3 Experimental Results

In this section we describe our experimental results. We had the following goals in our experiments: 1) examining the speedups obtained using SIMD parallelism for our target class of applications, 2) understanding the relative performance with different layouts, and 3) validating the analytical model we have developed.

Platforms We conduct our experiments on the following three machines: 1) C2050 GPU with the Fermi architecture, connected to an Intel Xeon E5630 CPU (2.53GHz frequency), 2) Quadro FX 5800 GPU with the Tesla architecture, connected to Quad-Core AMD Opteron(tm) Processor 2380 (2.49GHz frequency), and 3) Intel Xeon E5420 CPU

(2.5GHz frequency) with Streaming SIMD Extension 4 (SSE-4). Sequential and CUDA codes are compiled by g++ and nvcc, respectively, with O3 optimizations, whereas SSE codes are compiled with Intel ICC (Intel Parallel Composer 2011) compiler to fully utilize the SSE unit. We run all programs for 30 times, and speedup numbers include the mean and 95% confidence interval of the mean. For CUDA versions, the execution time difference across different runs is very small, and therefore, we omit the error bars.

Methods For the Random Forest application, we used trees and datasets from two different sources. The first is a popular open source numerical analysis and data processing library, ALGLIB, with which we used four datasets distributed by the UCI Machine Learning Repository — Poker, Shuttle, Abalone, and Satellite. The second is an internal random forest created for a large Microsoft product which has its own associated datasets. For the B+-Tree application, our evaluation was based on the experiments reported in the literature [137]. Based on this study, we establish a tree forest with different degrees of left bias input datasets: unbiased (50% bias), 62.5% bias, 75% bias and 87.5% bias. **Table 3.2** summarizes the basic information of tree forests used for both our B+-Tree and Random Forest applications.

For both Tesla 10 and Fermi architectures, we used the available shared memory as a buffer to hold input features and evaluation task queues. For the Fermi architecture, we used 48 KB shared memory (and thus, a 16 KB L1 cache), to hold both the evaluation buffer and also some of the top level nodes of the trees.

Baselines In both applications, we use a sequential, pointer-based CPU implementation as a baseline. We wrote our own sequential B+-tree baseline and in the Random Forest

DateSet	#Tree	#Ave_Node	Path_Leng	Ave_Path_Leng	Bias
B+-Tree	3584	513	8 - 11	9.0	various
Poker	3584	249	4 - 10	7.3	0.51
Shuttle	3584	217	4 - 10	7.5	0.55
Abalone	3584	333	5 - 12	8.0	0.52
Satellite	3584	353	4 - 12	8.2	0.55
Microsoft	3372	239	1 - 45	11.34	0.8

Table 3.2: Characteristics of Datasets Used in Our Experiments

application, we use either the ALGLIB original sequential implementation or the the Microsoft sequential implementation. To focus on speedups obtained using SIMD parallelism, we also created locality optimized sequential versions. This version uses a linearized layout, with a depth-first traversal, and use of *bagging* or *tiling* to organize B trees together into a single *bag*. This version is referred to as *DF_Seq* in our description. The value of B that leads to the best performance was empirically determined and used in our sequential version.

3.3.1 Speedups and Performance with Different Layouts

In this section we demonstrate the efficacy of our different layouts; in particular we demonstrate that our layouts are able (i) significantly increase the performance of these irregular applications and (ii) enable the use of SIMD architecture for this class of application.

B+-Tree

We evaluated the B+-Tree application on both the Fermi GPU and the SSE architecture. Results from two datasets, the unbiased traversal and the 87.5% are reported here. In **Figure 3.6**, we show the speedups of different versions over the sequential baseline on Fermi GPU architecture. We see that using SIMD execution we gain around 25X to 36X

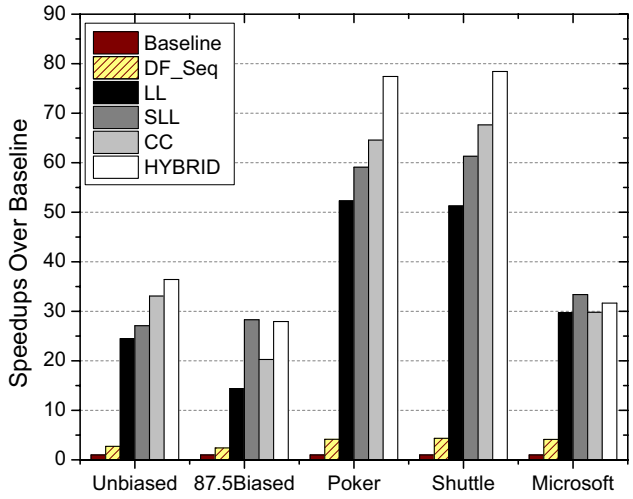


Figure 3.6: B+-Tree (2 datasets) and Random Forest (3 datasets) on FERMI GPU: Speedups of Different Versions Over Sequential Baselines

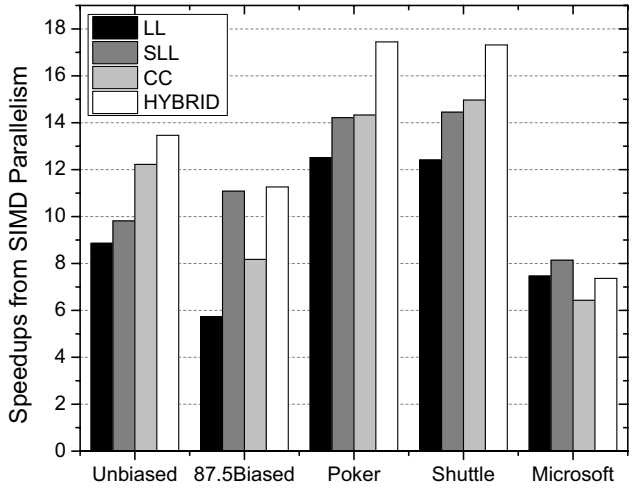


Figure 3.7: B+-Tree (2 datasets) and Random Forest (3 datasets) on FERMI GPU: Speedups from SIMD Parallelization (Same Layout Used for Sequential Execution)

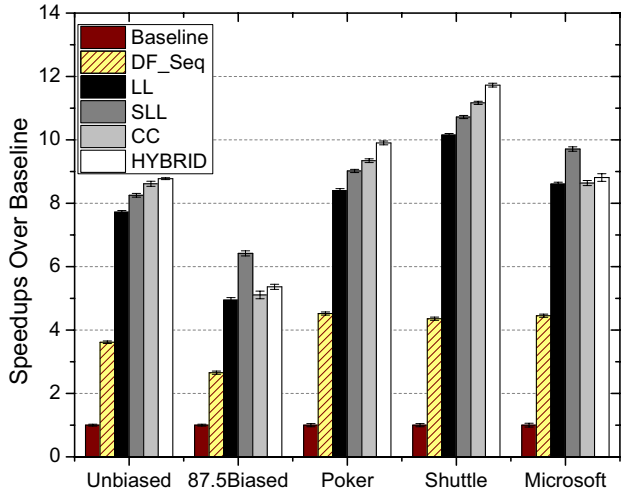


Figure 3.8: B+-Tree (2 datasets) and Random Forest (3 datasets) on SSE: Speedups of Different Versions over Sequential Baselines

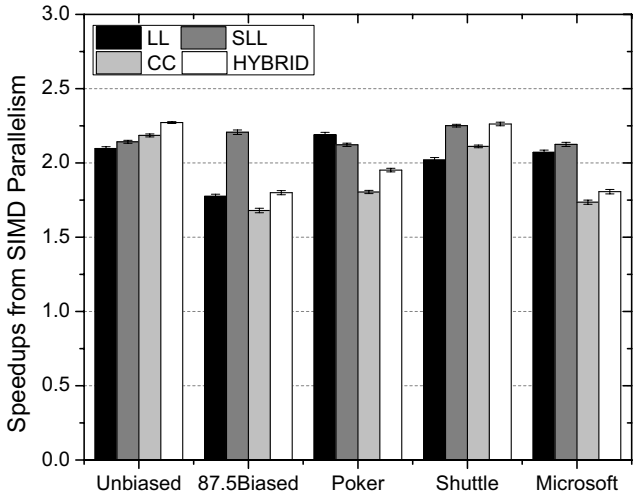


Figure 3.9: B+-Tree (2 datasets) and Random Forest (3 datasets) on SSE: Speedups from SIMD Parallelization on (Same Layout Used for Sequential Execution)

speedup over the sequential baseline for the unbiased traversal, and 15X to 29X speedup for the biased traversal. In order to separate the speedup from GPU's parallelism from linearizion of the structures, in **Figure 3.7**, we show the speedup of CUDA versions over the sequential version with the same layout. For unbiased traversal, the Fermi GPU architecture enables 9X to 13X speedup, while for biased traversal, 6X to 11X speedups are seen. Considering the irregular nature of these applications, we consider these speedups to be substantial, and the use of GPUs toward accelerating this application is justified.

From **Figure 3.6**, we also see that SLL and HYBRID are better than LL and CC. For unbiased traversal, HYBRID layout shows the best performance, since it allows benefits from inter-thread spatial reuse at earlier levels, and intra-thread spatial reuse at later levels. For biased traversal, SLL layout allows benefits from inter-thread reuse for a deeper levels as well, and has the best speedups. It should all be noted that for the 87.5% bias case, the switch level used is not optimized for bias, which cause the HYBRID layout to perform worse. Overall, up to a factor 2X difference in performance can be seen from the choice of layout from among the four layouts we have developed.

The same experiments were repeated on the SSE architecture, and the results are shown in **Figure 3.8** and **Figure 3.9**. Similar trends can be seen, and the only difference is that the optimization of memory layouts brings relatively smaller benefits on SSE than on the Fermi GPU. There are two reasons for this: 1) the SIMD lane width of SSE is much narrower, i.e., only 4-way parallelism is possible, and thus, the data requirements in each cycle are modest, 2) the modern CPU memory hierarchy is more advanced than GPU, with better prefetching as well as prediction strategies, which make it possible to avoid some of the cache misses.

Random Forest

We also conducted similar experiments on the ALGLIB and Microsoft Random Forests. Due to the space limitation, we only show the results from two datasets for ALGLIB Random Forests: Poker and Shuttle, as the trends from Abalone and Satellite almost exactly match the trends for these two.

In **Figure 3.6** and **Figure 3.8**, we show the speedups of different versions over the baseline on GPU and SSE, whereas in **Figure 3.7** and **Figure 3.9**, we show the speedup of GPU and SSE versions over corresponding best sequential versions. The speedups are very similar to those we obtained from B+-tree. One notable difference is that the Microsoft Random Forest shows only 8X speedup from the GPU. This is because in the dataset used, the input feature vector is very large (2648 float numbers), and shared memory cannot be used in the same fashion as in the UCI datasets. Among different layouts, the best performance is obtained from HYBRID for Poker and Shuttle, and SLL for Microsoft. Considering that Poker and Shuttle almost unbiased and Microsoft has a significant bias, these results are consistent with what we saw with the two datasets of B+-tree.

3.3.2 Model Validation

Our model is accurate and thus enables performance portability for this class of application on SIMD hardware. In this section we demonstrate our model's accuracy on a wide variety of SIMD architectures. Our model, as described in the previous section, only calculates the memory access times. For a more direct comparison with real execution times, the predicted values were normalized to execution times by using linear regression with a small number of sample values.

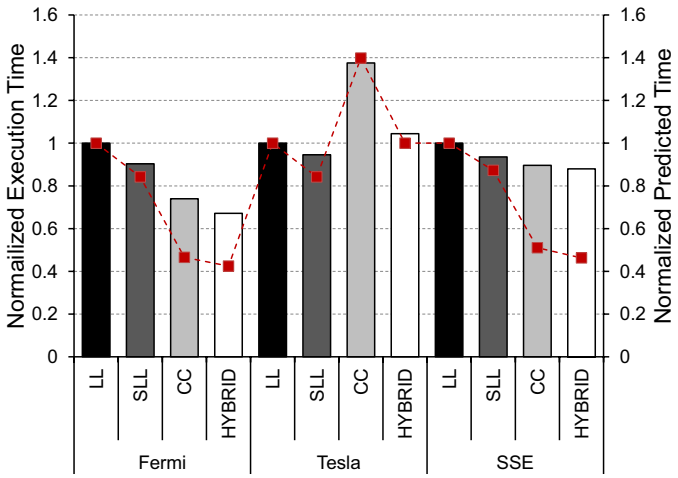


Figure 3.10: Real and Predicted Execution Times with Different Layouts and Architectures: B+ Tree Forest with Unbiased Traversal

Choosing Layouts on Different Architectures

Our analytical model is designed to support automatic optimization, with the goal of performance portability across different architectures. Thus, to evaluate its effectiveness, we first examine its ability to choose best layout for 2 different applications on three different architectures. In **Figure 3.10** and **Figure 3.11**, we compare observed and predicted execution time with four different layouts on two GPU architectures — Fermi (Tesla C2050) (with L1/L2 caches) and Tesla (Quadro FX5800) (without caches), and the SSE architecture with a sophisticated cache hierarchy. The left and the right y-axes correspond to measured and model predicted execution times. The (solid) bars report measured performance, whereas the dotted line shows the model predicted times.

Figure 3.10 reports results from the B+-Tree application, using a dataset where there is no bias. We can see that for all three architectures, our model is able to predict the layout that will result in the lowest execution time. Moreover, we can even predict the

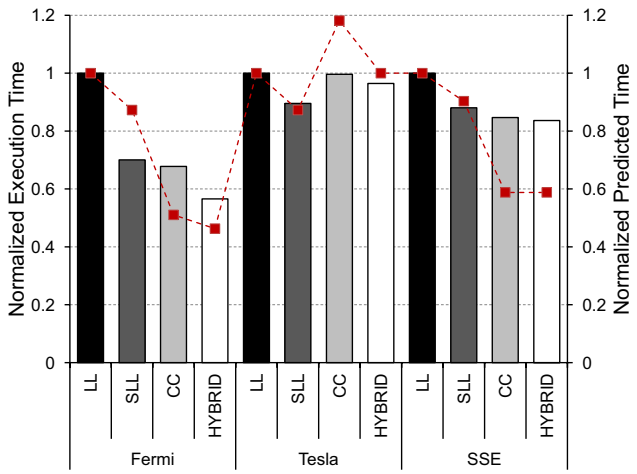


Figure 3.11: Real and Predicted Execution Times with Different Layouts and Architectures: Random Forest with Satellite Data set

relative execution times for the four layouts on all architectures. Particularly noteworthy is that the relative performance trends are very different for Fermi and Tesla, yet, they can be captured by our model. On Fermi architecture, the HYBRID layout results in the best performance, around 35%, 25%, and 10% faster than LL, SLL, and CC, respectively. On the Tesla architecture without cache hierarchy, SLL layout shows the best performance, while CC shows the worst, since we cannot exploit any intra-thread locality here. The trends on SSE are quite similar to those on Fermi, though the relative differences between different layouts are much smaller. This is because of support for aggressive prefetching and limited degree of parallelism, both of which reduce the performance impact due to spatial reuse.

We can see that our model does not always predict the precise execution time. This is because we are using a simple model, which just captures memory access times. Factors related to degree of parallelism in the application are not captured. However, our simple

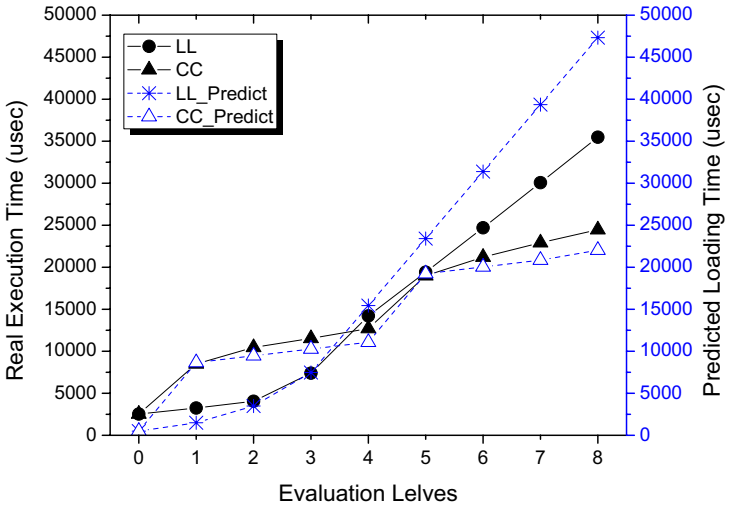


Figure 3.12: Comparing Real Execution and Model Predicted Times for Each Level of the Tree: LL and CC Layouts, B+-Tree on FERMI GPU

model is able to achieve our goal of predicting the relative performance with different layouts.

We repeat the same experiment on the Random Forest application, using the ALGLIB library tree with the Satellite dataset from the UCI repository. The results are shown in **Figure 3.10**, which show similar trends. Again, our model is able to predict which layout will result in the best performance in each case.

To further examine the efficacy of the analytical model, we carefully studied how it predicts the evaluation time for different levels of trees. The results are shown in **Figure 3.12**. The x-axis is the evaluation level of the tree, and the left and the right y-axes are the measured and the predicted times, respectively. Note that in all double-y-axis figures of this section, the solid lines correspond to the left y-axis, and the dotted lines correspond to the right y-axis. We have compared two contrasting layouts: LL and CC, For LL, both measured and the model predicted times show an exponential increased, followed by a linear

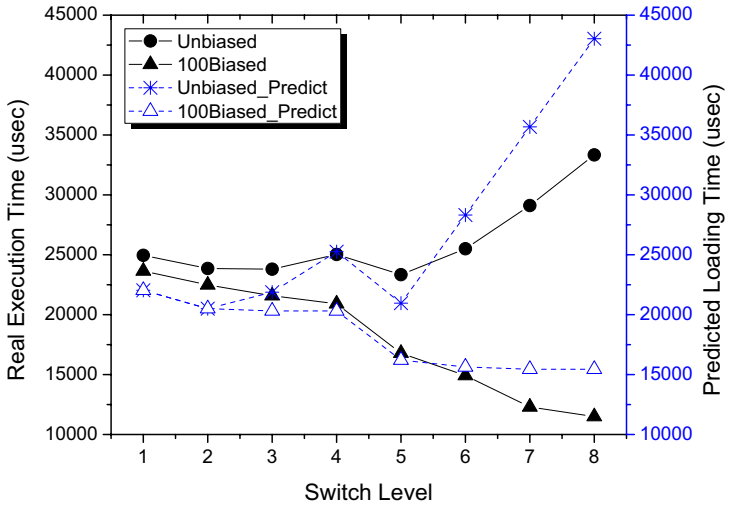


Figure 3.13: Comparing Real and Predicted Execution Times with Different Bias Levels: B+-Tree on FERMI GPU

behavior. The model predicted times follow the shape of the curve of the measured times, even though there are differences in the absolute values. Again, with the CC layout, the predicted execution time curve matches the shape of the curve of the measured execution times, and both show stage-increasing behavior. Similarly, the level at which the CC starts outperforming LL (level 4) can be correctly predicted by our model.

Handling Application Characteristics

Besides performance portability across different architectures, another goal of our analytical model is to be able to choose appropriate layout for applications that have different characteristics. We now show how the model is able to predict performance when there can be bias in traversal or sparse accesses.

An important factor that impacts the relative performance of different layouts is the bias degree of the traversal in each tree. Particularly, SLL layout can be more effective when

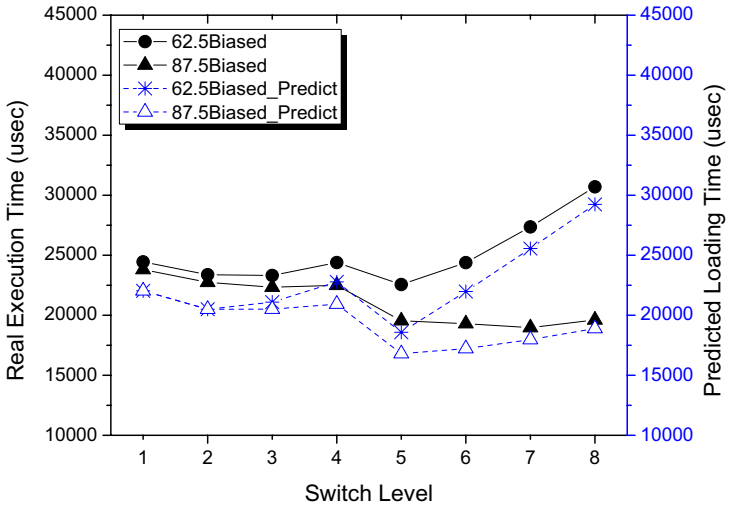


Figure 3.14: Comparing Real and Predicted Execution Times with Different Bias Levels: B+-Tree on FERMI GPU

there is a bias, and similarly, in using the HYBRID layout, it helps to *switch* at a deeper level when there is a bias. We now examine how our prediction model can help choose the appropriate layout, reporting the results in **Figure 3.13** and **Figure 3.14**, looking at unbiased case and three different levels of bias. We consider the HYBRID layout, and vary the *switch level*, i.e. the level at which we start using the CC layout. Again, we can see that the shape of curves for the real and predicted times match well. The performance obtained at the switch level predicted by the model is either the best, or very close to the best performance observed experimentally.

Another important application factor is the *sparsity level*, i.e. the probability that any given tree will not be accessed during one execution. We have again compared the observed and predicted times. In **Figure 3.15**, we show the cases with sparsity levels varying from 0% to 75%. Again, we use the HYBRID layout and vary the switch level. Again, by

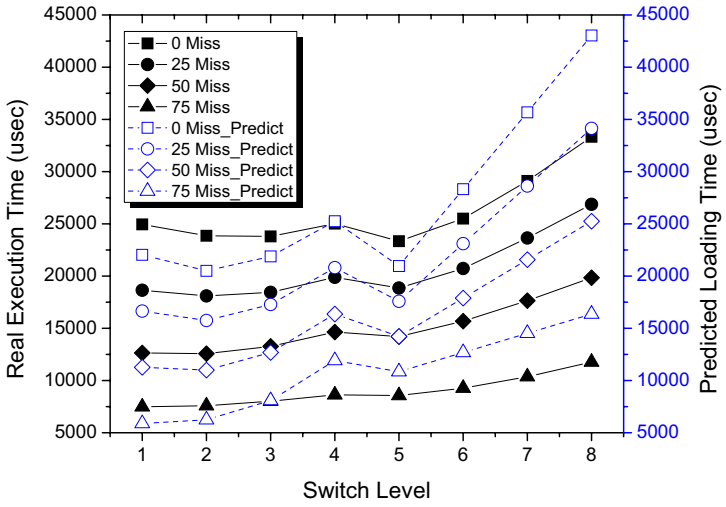


Figure 3.15: Comparing Real and Predicted Execution Times with Sparse Accesses

comparing the measured and predicted times, we see that the corresponding curves match very well, and the switch level leading to the best performance can be correctly predicted.

3.4 Related Work

This section compares our work with related research efforts from other groups.

3.4.1 Improving Data Locality of Irregular Data Structures

Improving memory locality of irregular data structures has also been studied in the past. Lattner *et al.* [80] proposed an *automatic pool allocation* method to manage the data structure layout in the heap to optimize the pointer intensive programs. Spek *et al.* [133, 132] developed a way to transform the recursive pointer-based data structure and related loops to the array-based data structure and counted loop structure that can be optimized by traditional methods.

There is significant research about linearization and reorganization of data and operations to reduce the cache misses or memory latency. Luk and Mowry [87] invented a *Linearization* method somewhat similar to our linearization method to support data prefetching. Ding and Kennedy [32] proposed a set of algorithms, including locality grouping and dynamic data packing, to improve the cache performance. Strout *et al.* [128] designed a compile-time framework to compose run-time data and iteration reordering transformation. Zhong *et al.* [144] proposed a structure splitting and array regrouping strategy based on the concept of *Whole Program Reference Affinity*. Mannarswamy *et al.* [91] presented a *Region Based Structure Layout* transformation method to reorganize the linked list-based data structures to increase the cache line utilization.

Chilimbi *et al.* [21] developed a set of *cache conscious* structures, and Rao and Ross [114] proposed a cache conscious B-tree structure. This early work considered sequential (single thread) execution, but does form the basis for the one of the layouts we will consider while optimizing for SIMD execution. Kim *et al.* [70] have designed an architecture-sensitive binary search tree for both CPUs and GPUs. They do not consider the processing of multiple trees concurrently, and thus face a distinct set of data locality issues. In a recent effort, Jo and Kulkarni [64, 65] design a tiling-like transformation to improve the performance of irregular data structure traversal. Their primary focus is on temporal reuse, which is complimentary to our work on spatial locality.

3.4.2 Exploring Inter-thread Data Locality in Multi-threaded Environment and Cache Modeling

There have also been many efforts on exploiting inter-thread data locality in multi-threaded environment. In this area, Meng *et al.* [94] have designed a symbiotic affinity scheduling (SAS) algorithm to maximize the cache locality of the threads on the same

core, Che *et al.* [20] propose an API, Dymaxion, to help programmers to reorganize the data to achieve better data locality in a heterogeneous environment, Zhang *et al*. [141] propose a method to transform programs in a cache-sharing-aware manner to improve the performance, whereas Jang *et al.* [61] provide a set of techniques to transform the data according to different memory access patterns to improve the performance on both AMD and NVIDIA GPUs. Considering performance fairness in multiprocessor environment, Zhou *et al.* [145] design a mechanism to share the cache among concurrent applications. Ding *et al.* [33] propose a runtime library, *User Level Cache Control*, for programmers to explicitly manage and optimize the last level cache for data sets in multi-threaded programs. More recently, Unkule *et al.* [131] present a software framework to analyze and restructure the GPU kernels to explore inter-thread data locality. The distinct aspect of our work are: 1) we are focusing on improving inter-thread spatial reuse across concurrent threads from the same application, whereas most of the above work considers capacity and conflict misses from different applications, and 2) our focus is on detailed analytical modeling of one application with the goal of performance portability.

Cache behavior modeling and analysis is widely used as part of restructuring compilers that focus on scientific (array-based) programs. Earlier work in this area includes those from Porterfield [111] and McKinley [92]. More recently, Cascaval and Padua [14] proposed a machine independent model to estimate cache misses during compile time based on stack algorithms. Zhong *et al.* [144] developed a model based on *Whole Program Reference Affinity*. Our work is distinct in considering a different class of applications, and combining intra-thread reuse with inter-thread reuse for SIMD.

3.5 Summary

Optimizing an application on any one particular architecture is a challenging task—optimizing that application for several architectures is a daunting, if not impossible, task. In other words, it is difficult for programmers to guarantee an application's *performance portability*.

For SIMD architectures, the memory hierarchy is often the bottleneck to peak performance. In this work we introduce data layout optimizations for a common class of memory bound applications designed to balance intra thread and inter thread spatial locality. Further, to remove the burden on a programmer from deciding which data layout to choose for which SIMD architecture, we develop an accurate model that enables performance portability for these applications, and extensively validate it across different applications and architectures. Though our work has been in context of a specific class of applications, the main underlying idea of analytically choosing and/or combining intra-thread and inter-thread locality is broadly applicable, especially, as multi-core and many-core architectures become more popular.

Chapter 4: Efficiently Parallelizing Irregular Applications on Xeon Phi by a Programming System

As we mentioned before, Intel Xeon Phi is an emerging many-core coprocessor architecture providing us both MIMD and SIMD parallelism, while parallelizing applications involving irregular memory access on such kind of architecture efficiently is still an open problem.

In this Chapter, we present a programming system to address this problem by exploring both MIMD and SIMD parallelism. Specially, we focus on applications involving irregular reduction communication patterns, such as Molecular Dynamics and Euler, in which, indexed array access causes many challenges, such as non-continuous memory access, and writing conflicts. Our framework leverages a novel runtime data reordering scheme to address these challenges.

4.1 Overview of our Approach

This section presents some background information about irregular reduction application, and provides an overview of our overall solution.

```
/*Indexed Array Accesses – Molecular Dynamics*/
P1(m), P2(m), L(n)
for ( t = 0; t < time; ++t ) {
    for (i = 0; i < n; ++i) {
        d = P1(L(i)) – P2(L(i));
        force = power(d, -5) – power(d, -6);
        P1(L(i)) += force;
        P2(L(i)) += -force;
    }
}
```

```
/*Pointer-Chasing Accesses – Random Forest*/
float Search(float features[]) {
    float feature = features[this->featureIndex];
    ...
    if (feature <= this->threshold)
        return this->left->Search(features);
    else
        return this->right->Search(features);
    …
}
```

Figure 4.1: The Code Examples for Two Classes of Irregular Applications

4.1.1 Irregular Reduction Application

A typical application involving irregular reduction communication pattern, Molecular Dynamics, is shown in the left-hand side of **Figure 4.1**. In this example, the two arrays P1 and P2 are not indexed directly by the iterator variables, *i.e.*, they are not affine array access. Different from the pointer-chasing irregularity in Random Forest as shown in the right-hand side of **Figure 4.1**, the irregularity of this application is coming from the fact that these indexes cannot be known during the compilation time, and are normally not continuous values. Constraint by this kind of irregularity, both spatial and temporal locality for such kind of applications is poor, even for uni-processor machines.

In irregular reduction applications, indexed array accesses commonly arise, to represent more complicated relationships between data, which are more efficient than pointers. For example, in Molecular Dynamics applications in **Figure 4.1**, the indexed array stores the edge information, and the in-directed reference indicates the relationship between the points and edges, *i.e.*, iterating on the edges to update on the information of associated points. Another typical irregular reduction application is Euler [26], which is a Computational Fluid Dynamics application similar to Molecular Dynamics.

As we mentioned above, for these applications, the indexes cannot be obtained only according to the compilation information as affine array access, while a run-time reorder algorithm should be developed instead to improve their memory performance.

4.1.2 Challenges and Opportunities

There are two levels of parallelism that one can seek on the Xeon Phi: MIMD parallelism supported by large number of hyper-threads, and SIMD parallelism provided by the wide VPU. There are challenges associated with each of them, as well as opportunities to exploit information from specific communication patterns.

MIMD Parallelization Issues

A Xeon Phi can be viewed as a SMP machine, in which all the cores not only share the same memory address, but also a coherent cache space. Thus, the traditional MIMD parallelization methods, like OpenMP, can also be applied with the support of the Intel compiler. Yet, there are many opportunities for exploiting information about specific communication patterns.

Particularly, applications with different communication patterns usually have different requirements on task partitioning and scheduling. Specially, for irregular reductions, a technique like the *reduction space partitioning* [56] can be used to avoid conflicts between the threads. Moreover, dynamic, fine-grained, scheduling could achieve better performance over static scheduling by achieving better load balance.

Communication pattern specific information can also help in other ways. Data reorganization is one of the optimizations to support vectorization, but data reordering can also provide better cache locality for irregular reductions. These optimizations are normally not performed by a more general framework, such as an OpenMP implementation.

SIMD Parallelization Issues

In SIMD execution, one memory access operation can load (store) multiple data elements simultaneously from (to) the memory. However, there are strict restrictions on how and when such operations can be applied.

Unaligned/Non-unit Stride Accesses: For using SIMD parallelism, the start of the read or write memory address has to be 64 bytes aligned on Xeon Phi. But, it is very difficult to satisfy this requirement for irregular reduction application. In addition, different SIMD lanes can only access continuous memory address. Thus, accesses of elements from an array of structures or data accessed through indirection arrays cannot exploit SIMD parallelism directly.

Control Flow Dependencies: At any time, all the SIMD lanes have to execute the same instructions on different data elements. However, in the different branches of an *if-else* clause, different lanes may execute different instructions, which is not supported by SIMD. This kind of control flow arises very commonly in generalized reduction and irregular reductions.

Data Dependencies and Conflicts: When different SIMD lanes try to write to the same location, the behavior is undefined, as there is no locking operation. In the case of both generalized reductions and irregular reductions, such write conflicts arise. Thus, how to solve the data dependencies and conflicts for SIMD effectively and efficiently is another challenge.

4.2 API and Runtime Support on Xeon Phi

To parallelize irregular reduction applications on Xeon Phi, we design and implement a set of APIs and a runtime support carefully described in Article [57]. As we mentioned

above, the runtime support focuses on two aspects, MIMD parallelization, and SIMD parallelization. In this section, we present a detailed example written in such kind of API, and explain the basic idea of data reorganization, a key optimization, for irregular reduction applications.

4.2.1 Sample Kernel

```
void kernel (int *edge[2], float *edgeData, float *velocity, float *update, int index)
    {
        //step 1 Load node data according to edge data
        vint v_n0.load(edge[0]+index);
        vint v_n1.load(edge[1]+index);

        vfloat v_edgeData_0.load(edgeData, index, 4);
        vfloat v_edgeData_1.load(edgeData+1, index, 4);
        vfloat v_edgeData_2.load(edgeData+2, index, 4);

        vfloat v_velocity_n0_0.load(velocity, n0, 4);
        vfloat v_velocity_n0_1.load(velocity+1, n0, 4);
        vfloat v_velocity_n0_2.load(velocity+2, n0, 4);

        vfloat v_velocity_n1_0.load(velocity, n1, 4);
        vfloat v_velocity_n1_1.load(velocity+1, n1, 4);
        vfloat v_velocity_n1_2.load(velocity+2, n1, 4);
        //step 2 Compute the force
        vfloat v_a0 = (v_edgeData_0 * v_velocity_n0_0 + v_edgeData_1 * v_velocity_n0_1
             + v_edgeData_2 * v_velocity_n0_2) / 3.0;
        vfloat v_a1 = (v_edgeData_0 * v_velocity_n1_0 + v_edgeData_1 * v_velocity_n1_1
             + v_edgeData_2 * v_velocity_n1_2) / 3.0;

        vfloat v_r0 = v_a0 * v_velocity_n0_0 + v_a1 * v_velocity_n1_0 + v_edgeData_0;
        vfloat v_r1 = v_a0 * v_velocity_n0_1 + v_a1 * v_velocity_n1_1 + v_edgeData_1;
        vfloat v_r2 = v_a0 * v_velocity_n0_2 + v_a1 * v_velocity_n1_2 + v_edgeData_2;

        //step 3 Reduction if the node is within current partition
        mask m0 = (v_n0 >= v_part_low) && (v_n0 < v_part_high);
        update.reduction(4, v_n0, 0, v_r0, m0);
        update.reduction(4, v_n0, 1, v_r1, m0);
        update.reduction(4, v_n0, 2, v_r2, m0);

        mask m1 = (v_n1 >= v_part_low) && (v_n1 < v_part_high);
        update.reduction(4, v_n1, 0, v_r0, m1);
        update.reduction(4, v_n1, 1, v_r1, m1);
        update.reduction(4, v_n1, 2, v_r2, m1);
}
```

In the sample kernel above, we show the code snippet of a classic irregular reduction application, *Euler*. There are three steps in this application kernels. In step 1, we need to load the nodes pairs according to the edges, which involves indirection array based accesses. In the original MIC SIMD intrinsics, it can only be implemented by the built-in

gather operations. With our proposed API, the load operation can also support indirect accesses by providing the index and the scale information. In the step 2, the vectorized force computation with our API is almost the same to the original sequential one. Step 3 is the irregular reduction stage, when we reduce (add operation) the computed force to the nodes pair within the current partition to avoid duplicated updates. Step 3 involves two important features of our API, involving handling of control flow and reduction, i.e., doing the reduction if some conditions are satisfied. To support such kind of reduction, we provide our own *mask reduction* function call as shown in the sample kernel above. In the *mask reduction* functions, we need to resolve the write conflict issue, which is addressed by our data reordering mechanism we will introduce in next section.

To summarize, in our API, the code with arithmetic operations is almost as same as the original (serial) code. The reduction in our API is provided through a function interface, which allows us to vectorize these codes, whereas most compile-time solutions fail to do this. The most complicated part of our API is handling of control flow, where branches are replaced by mask operations. However, we note that existing vectorizing compilers do not handle control flow at all (as we will show through experimental results), and manual vectorization in presence of control flow is very complicated (please see an example in **Figure 4.2**).

4.2.2 Data Reorganization

SIMD operations on Xeon Phis (or any SSE-like instruction set) can only be applied if there are continuous and aligned memory access. Many applications have non-unit stride and unaligned or even random memory accesses. Such kind of accesses impede compiler

```
if(a < b) a += b;
else a -= b;
```
(a) A Statement with Control Flow

```
__mm128i mask1 = _mm_cmplt_epi32(a, b);
__mm128i mask0 = _mm_andnot_si128(mask1,
_mm_set1_epi32(0xffffffff));
__mm128i res = _mm_and_si128(_mm_add_epi32(a, b), mask1);
__mm128i oldval = _mm_and_si128(a, mask0);
a = _mm_or_si128(res, oldval);
res = _mm_and_si128(_mm_sub_epi32(a, b), mask0);
oldval = _mm_and_si128(a, mask1);
a = _mm_or_si128(res, oldval);
```
(b) SIMD Parallel Code for Example in (a)

```
__mmask16 mask1 = _mm512_cmplt_epi32(a, b);
__mmask16 mask0 = _mm512_cmpge_epi32(a, b);
a = _mm512_mask_add_epi32(a, mask1, a, b);
a = _mm512_mask_sub_epi32(a, mask0, a, b);
```
(c) SIMD Code with Mask Type

Figure 4.2: Example with Control Flow (a) sequential code (b) SIMD code (c) SIMD code with mask type

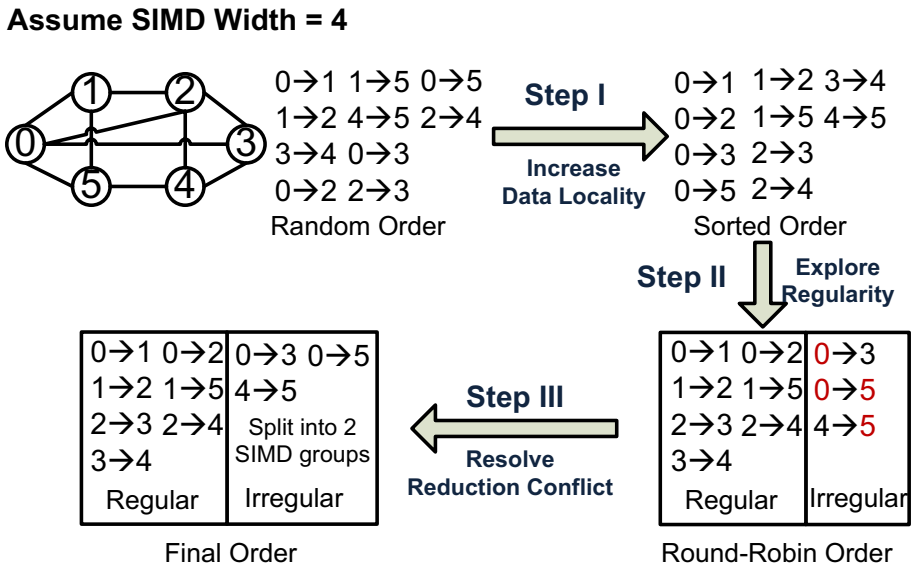


Figure 4.3: Irregular Reduction Edges Reorder

vectorization. In our framework, we exploit the knowledge about underlying communication patterns to reorganize the data and facilitate SIMD parallelization.

Irregular Reductions: In an irregular reduction kernel, indirect data references can cause very random memory accesses. If we want to vectorize these operations, a large number of *gather* and *scatter* operations must be invoked. There are many existing efforts trying to solve or alleviate this problem from different perspectives. Kim and Han [72] design an algorithm to replace unnecessary gather and scatter operations by scalar operations. Wu *et al.* [136] try to resolve a very similar problem, coalesced memory access, within the context of the GPU architecture. Focusing on inter-iteration parallelism on an irregular reduction for a SSE-like instruction set, we address this problem by a novel computation (edges data) reordering method, which we describe below.

Our method is explained with the help of an example shown in **Figure 4.3**. First, the motivation for our method is as follows. The gather and scatter operations incur a very long latency when the data locality is poor, because each gather and scatter operation works at the unit of the entire cache line. For example, when the required data is split across multiple cache lines, we need multiple gather operations to load them. So, the first objective of the our data reorganization method is to reorder the edge data, and increase data locality. To achieve this objective, based on the partitioning algorithm that is used for task partitioning at the MIMD level, we further reorder the *edge* data according to their first nodes (Step I in **Figure 4.3**). As a result, at least for one of the end-points of the edge, data is likely to be in the same cache line.

The second objective is to replace *gather* and *scatter* operations by normal SIMD *load* and *store* operations to the extent possible. To achieve this objective, we partition the edges into *regular partitions* and *irregular partitions*, as explained below. First, we further reorder the *edges* data (Step II in **Figure 4.3**), so that the edges are ordered in a round-robin manner according to their first nodes, and we have a consecutive set of first node for the set of edges that will be processed in one SIMD step (a *regular partition*). Now, clearly, given a set of edges, we cannot ensure that we can simply reorganize them as a set of regular partitions. A set of edges that will be processed in one SIMD step but whose first nodes do not form a consecutive set is an *irregular partition*. Thus, we will likely have a set of regular partitions and irregular partitions. After this, we can further apply AoS to SoA to duplicate all the first nodes of edges in the regular partition. In such case, we can apply normal SIMD *load* and *store* operations for the first nodes of edges in the regular partition, and only apply *gather* and *scatter* operations for the remaining nodes.

The third objective is to resolve write conflicts within the same SIMD register for the second nodes of edges in a regular partition and for all the nodes in an irregular partition. Note that this issue arises for generalized reductions as well. The problem is that unless we are careful, different SIMD lanes may update the same element of the SIMD register, causing a race condition. A larger SIMD width increases this possibility, and moreover, indirect accesses can make it hard to avoid such situations. In order to resolve this problem, we have two options: a) serialized reduction; and b) further data/computation reorder. For serialized reduction, we provides a way to automatically serialize all the reduction operations to eliminate the possible conflicts. Alternatively, we can further reorder the elements into blocks according to the SIMD width, even introducing *bubble* elements. For irregular reductions, we can further reorder the edges (computation order) as shown in Step III of **Figure 4.3**, by which, we can make sure there is no write conflict within the same SIMD register.

4.3 Evaluation

In this section, we evaluate our framework using two irregular reduction applications. The objectives of our experiments were: 1) Comparing the performance of applications developed using framework, over hand-written parallel versions (using Pthreads), and evaluating the SIMD parallelization in our framework, over the ICC compiler generated SIMD code, 2) Quantifying the overheads of our runtime framework, by comparing performance against the hand-written SIMD code for SIMD parallelization, 3) Comparing the performance of MIMD parallelization from our framework against OpenMP, another high-level framework, and further evaluating the SIMD parallelization by our framework against what is achieved by ICC compiler with OpenMP directives. All experiments were conducted on

a Xeon Phi SE10P card, which has 61 cores each running at 1.1 GHz, with four hyper-threads per core, along with a 32 MB L2 cache and 8 GB GDDR5 memory. The compiler that we used is Intel ICC compiler 13.1.0. All benchmarks are compiled with *-O3* optimization. Compiler vectorization is turned on and off by *-vec* and *-no-vec* options, respectively. All experiments were running in the *Native Model* with the *-mmic* option.

4.3.1 Benchmarks

Molecular Dynamics (MD) is an irregular reduction kernel used to study the structural, equilibrium, and dynamic properties of molecules. The simulation iterates over all the edges, and updates the attributes associate with the two end nodes. The small dataset used in the experiments has 16K nodes and 2M edges, while the large one has 256K nodes and 32M edges. **Euler** is another irregular reduction kernel based on Computational Fluid Dynamics (CFD) that takes description of the connectivity of a mesh and calculates quantities like velocities ate each mesh point. The small dataset used in our experiments has 182K nodes and 1.13M edges, while the large one has 1.4M nodes and 8.9M edges.

4.3.2 Speedups from Our Framework

Our first set of experiments focused on comparing the SIMD parallelization with our framework against compiler generated SIMD code (auto-vectorization), and hand-written SIMD code. Compiler SIMD parallelization was applied on Pthreads code, so as to also allow shared memory parallelization. Pthreads-based shared memory parallel versions used similar style (and thus obtain similar performance) as the shared memory parallelization supported by our framework, though the programmer effort is much smaller with our framework. In **Figure 4.4**, we compared the best performance between the pthread versions with and without compiler vectorization, and the vectorization versions with hand-coding and

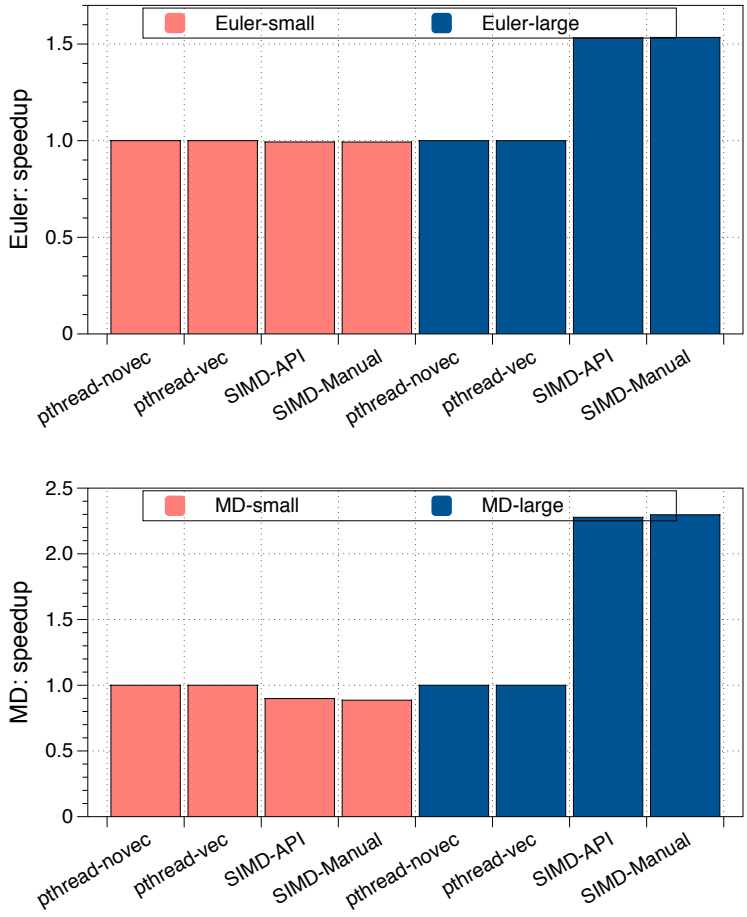


Figure 4.4: Speedup of Pthread without SIMD (`Pthread-novec`), Pthread with auto-SIMD (`Pthread-vec`), MIC SIMD with our framework (`SIMD-API`), and hand-written SIMD (`SIMD-manual`): Euler, and MD with `small` and `large` datasets each

our API, for *small* and *large* datasets described earlier. The performance, shown in **Figure 4.4**, is the one with the number of threads that leads to the best performance (which maybe different across versions). The numbers reported are relative speedups, with baseline of Pthreads version without vectorization.

For irregular reductions, the production compiler cannot vectorize a loop with indirection-based memory access at all. In our framework, we use data reordering together with a reduction in the use of gather and scatter operations to vectorize such kind of loops, which turns out to be effective when the datasets are large. We achieve 1.5 and 2.5 times speedup over the pthread versions for Euler and MD, respectively. For small datasets, the performance of the best `SIMD-API` version is comparable to the pthread versions. However, the best configuration with `SIMD-API` involves fewer threads (60 instead of 244). In other words, for the smaller datasets, enough parallelism is not available to exploit both MIMD and SIMD features. Comparing to the best `SIMD-manual` versions, `SIMD-API` incurs neglectable overheads.

4.3.3 Overall Scalability

In **Figure 4.5**, we compare the scalability of `pthread-novec`, `pthread-vec`, `SIMD-API`, and `SIMD-Manual` with an increasing number of threads. Execution with a single thread and no vectorization on Xeon Phi is used as the baseline, and thus, we are evaluating the combined benefits of shared memory parallelization (61 cores), hardware multi-threading (4 threads per core) and SIMD units. The performance scales well for all the versions. `SIMD-API` outperforms both `pthread-vec` and `pthread-novec` in most cases, consistent with what we reported earlier. `SIMD-API` achieves better relative performance when the number of threads is small. For instance, when the number of threads

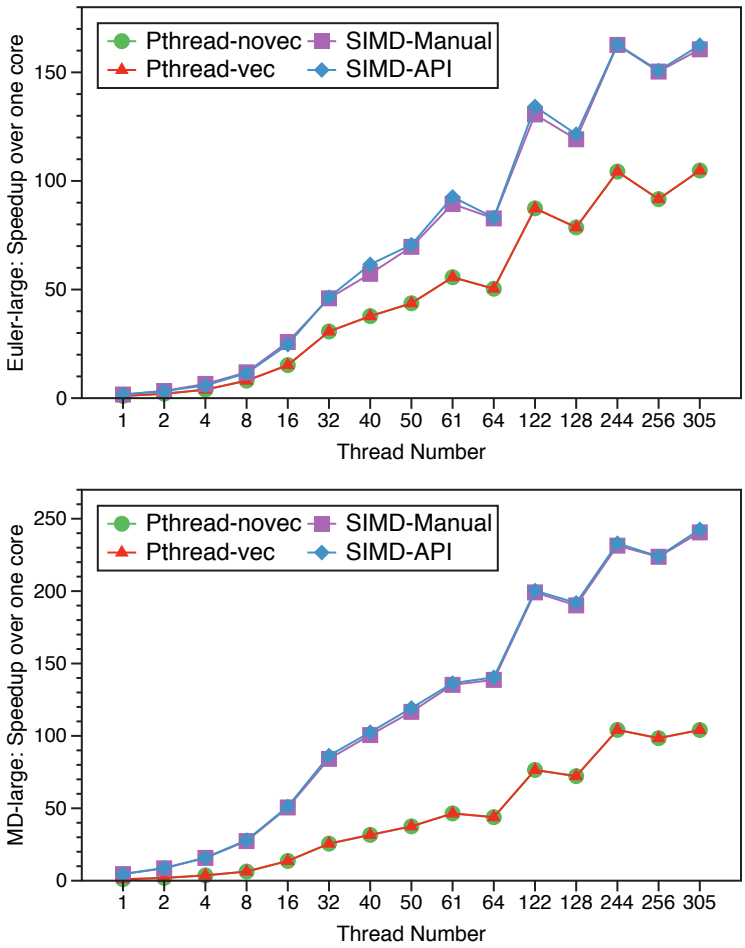


Figure 4.5: Scalability with Increasing Number of Threads: Pthread without vectorization (`Pthread-novec`), Pthread with auto-vectorization (`Pthread-vec`), SIMD with API (`SIMD-API`), and hand-written SIMD (`SIMD-manual`) with Euler, and MD (large datasets) - Relative Speedups Over 1 Thread Execution on Xeon Phi with no Vectorization

is one, `SIMD-API` is 20 times better than the `Pthreads-novec` version. With small datasets, as the number of threads increases, the vectorization advantage with `SIMD-API` becomes restricted due to limited amount of overall work. The overall speedups obtained range between 160 and 250, depending upon the application. Thus, we can see that our framework is effective in allowing application developers to exploit the Xeon Phi chip.

4.3.4 Comparison with OpenMP

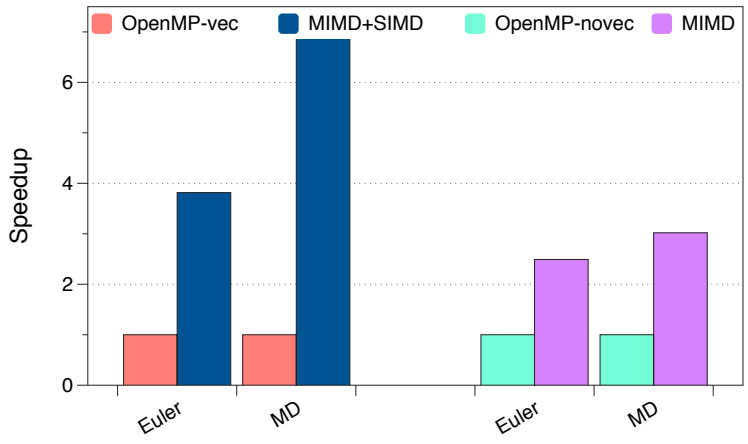


Figure 4.6: Benefits of MIMD+SIMD Execution in our Framework and MIMD-only execution

Our last set of experiments had two distinct goals. First, we wanted to examine how SIMD parallelization with our framework compares against SIMD parallelization performed by the ICC compiler with OpenMP directives. Second, because both OpenMP and the MIMD API in our framework provide a high-level model for developing shared memory applications, we wanted to examine if our framework offers any performance advantages, possibly because it exploits the knowledge of the underlying communication patterns.

In **Figure 4.6**, we compared our MIMD parallel framework with and without SIMD parallelization to the OpenMP MIMD parallelization with and without the compiler vectorization. Comparing `MIMD+SIMD` to `OpenMP-vec`, more than 3 and 6 times speedup is achieved in Euler and MD respectively, due to the better SIMD parallelization and efficient MIMD parallelism. Now, focusing just on MIMD parallelization, our parallel framework still obtains better performance compared to OpenMP. Overall, combining both MIMD and SIMD parallelization, our framework is better for both applications. As discussed throughout this Chapter, these advantages come from a number of factors, e.g., our framework can vectorize an irregular kernel with indirection-based memory accesses, while OpenMP compiler cannot, and pattern-aware MIMD partitioning and scheduling can avoid locking overheads.

4.4 Related Work

Intel SSE has been a part of the x86 since 1999, and there have been many efforts to automatically accelerate various applications using these instructions. For vectorizing stencils, memory alignment is a key problem, which was addressed by Eichenberger *et al.* [36] and Nuzman *et al.* [103] with data reorganization methods. More Recently, Henretty *et al.* [51] propose a system that involved improving data locality and utilizing short-vector SIMD optimizations, and Kong *et al.* [76] designed a Polyhedral compiler to perform loop transformation, optimization and vectorization for imperfectly nested loops.

Vectorizing irregular applications on SSE has also gained considerable interest in recent years. Kim and Han [72] propose a compiler method to generate efficient SIMD code for irregular kernels containing indirection based memory accesses. However, their work is on Cell SPU, with much shorter SIMD unit compared to the Xeon Phi and their method

primarily focuses on intra-iteration vectorization. We focus on aggressive inter-iteration parallelism, consistent with presence of wide SIMD lanes. There are also efforts on hand-optimizing irregular applications on SSE and other vector units [124, 70].

Some of the GPU compilation efforts have a similar favor, because SIMT is closely related to SIMD. This includes work on parallelizing stencil applications on GPUs [27, 95, 101, 54]. For irregular applications on GPU, the coalesced memory access problem has also been addressed [136, 140]. However, because of the differences in the architectures (e.g. lack of atomic stores), our data reorganization methods are different. Overall, as compared to the existing work on SIMD compilation, the key distinctive aspects of our work are: 1) handling branches in a general way, 2) exploiting features in the IMCI instruction set, 3) using knowledge of communication patterns for runtime data reorganization, and 4) use of an overloaded function approach, which is unlike all previous efforts on SIMD parallelization, and can simplify the compiler code generation in the future.

There are also many efforts to parallelize various applications (classes) on Xeon Phi, which includes the work of Liu *et al.* [84] on Sparse Matrix-Vector Multiplication, Pennycook *et al.* [110] on parallelizing a Molecular Dynamic application, and Lu *et al.* [86] on optimizing the MapReduce framework. We have, to the best of our knowledge, offered the first general and end-to-end system for exploiting both MIMD and SIMD parallelism on the Xeon Phis.

4.5 Summary

This work has presented and evaluated a framework for irregular reduction application parallelization on the Xeon Phi coprocessors. Two distinct aspects of our work are 1) use of the knowledge of underlying patterns to perform job partitioning and scheduling in

MIMD setting and data reorganization for SIMD parallelization, and 2) a very different approach for SIMD code execution, based on the implementation of overloaded functions, with runtime management of masks. Overall, we perform SIMD parallelization in presence of control flow, irregular accesses, and reductions, unlike previous work with SSE-like instruction sets. Moreover, our work can also be seen as providing a CUDA-like language (and its implementation) for using SSE-like instruction sets.

Chapter 5: Automating and Optimizing Data Transfers for Many-core Coprocessors

An important issue for using Xeon Phi as coprocessors is to transfer data between CPU host and Xeon Phi efficiently. For bit-wise copyable data structures such as one dimensional array, we can simply use some existing high level APIs, such as LEO (Language Extension for Offload). However, for dynamic multi-dimensional arrays, and other irregular, multi-level pointer data structures, such as link lists, trees, and graphs, there is no existing easy-programming solution with high performance.

This Chapter presents an automated framework that uses both compile-time and runtime solutions to address this problem. Specially, our compile-time solutions are for dynamic multi-dimensional arrays, while our optimized runtime solution is for other more complicated irregular data structures, such as linked list, trees, graphs, and so on.

5.1 Motivation and Problem Definition

As we stated previously, our target is the high-level APIs for exploiting architectures. **Table 5.1** summarizes these APIs, which allow the developer to mark code regions in the application from which offloadable tasks can be generated by the compiler. These APIs are all intended to improve developer productivity and simplify code maintenance, by hiding

	offload	synchronization	data transfer	data reuse
LEO	`#pragma offload`	`<signal,wait>`	`<in,out,inout>`	`nocopy`
OpenAcc	`#pragma acc kernels`	`<async,wait>`	`<copy,copyin,copyout>`	`present`
OpenHMPP	`#pragma hmpp codelet`	`<asynchronous,synchronize>`	`args[item].io=<in,out,inout>`	`args[item].noupdate=true`
OmpSs	`#pragma omp task`	`<input,output,taskwait>`	`<copy_in,copy_out>`	*by default*
OpenMPC	`#pragma cuda gpurun`	OpenMP `<nowait>`	`<c2gmemtr,g2cmemtr>`	`<nog2cmemtr,noc2gmemtr>`

Table 5.1: Key Directives in Common Directive-based Languages for Accelerator Programming

many details of data transfers and data allocation on the accelerators. As a specific example we consider LEO (Language Extension for Offload), which supports the *coprocessor offload interface* (COI), and is the primary annotation language to be used on the Xeon Phi systems. COI provides #pragma offload directive for marking offloadable code regions. This is similar to OpenAcc's #pragma acc kernels[7]. The compiler generates invocations to runtime libraries that support data and computation offload. By default, execution on the CPU is suspended when such a code region is encountered, continued on the coprocessor and then resumed on the CPU after the offloaded code region has executed to completion. Special synchronization primitives (e.g., `signal/wait`) can be used for enabling asynchronous offload (**Table 5.1**).

Our focus is on data transfers between the CPU and the accelerator. Even while using one of the high-level APIs, the developer has to orchestrate data transfers between the CPU and coprocessor. This can be done using `in/out/inout` clauses for LEO or `copyin/copyout` for OpenAcc, or similar constructs from other directive-based languages, as shown in **Table 5.1**. However, it turns out that there is still a substantial complexity for the user when dynamically allocated multi-dimensional arrays or other data structures based on multi-level pointers are involved.

[7]OpenACC: Directives for Accelerators. http://www.openacc-standard.org/

```
 1: int *A, *B;
 2: A = (int *) malloc(n * sizeof(int));
 3: B = (int *) malloc(n * sizeof(int));
 4: #pragma offload target(mic) in(B:length(n)) out(A:length(n))
 5: {
 6:      #pragma omp parallel for private(i)
 7:      for (i = 0; i < n; ++i) {
 8:          A[i] = a * B[i];
 9:          ...
10:      }
11:}
```

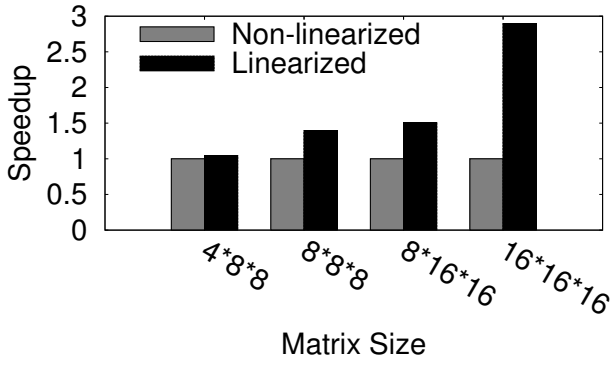
Figure 5.1: One-Dimensional Array Offload

Figure 5.1 and **Figure 5.2** show examples for handling one-dimensional and multi-dimensional arrays, respectively, with one of the high-level languages. In the case of a single-dimensional array, only one contiguous memory region needs to be transferred. For multi-dimensional arrays, on the other hand, numerous array components scattered over memory may have to be handled. Note that this complexity arises because of the way that C-versions of most existing scientific applications today allocate memory. Since the goal of the programmer is to simplify array accesses for main computational loops in the code, an N dimensional array is allocated by allocating one-dimensional arrays inside an $N-1$ dimensional loop (similar to the example in **Figure 5.2**). In some other applications, including the Multi-Grid (MG) benchmark from NAS Parallel suite, arrays are not rectangular, which further adds to the complexity.

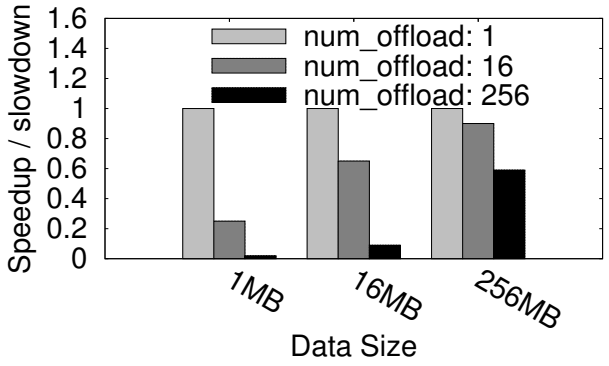
Recall that in Section 1, we had stated four requirements while addressing the problem, which included needs for fully utilizing DMA, and reducing memory allocation overheads. To motivate their impact, we present certain experimental observations. Consider the code

```
1: #define ALLOC alloc_if(1) free_if(0) //allocate data
2: #define REUSE alloc_if(0) free_if(0) //keep data persistent
3: #define FREE alloc_if(0) free_if(1) //free data
4: int **A, **B;
5: int size[m]; ...
6: /* allocate pointers on CPU */
7: A = (int **) malloc(m * sizeof(int *));
8: B = (int **) malloc(m * sizeof(int *));
9: for (i = 0; i < m; ++i){
10:     A[i] = (int *) malloc(size[i] * sizeof(int));
11:     B[i] = (int *) malloc(size[i] * sizeof(int));
12:}
13:/* allocate pointers on coprocessor */
14:#pragma offload target(mic) nocopy(A:length(m) ALLOC)
15:    nocopy(B:length(m) ALLOC)
16:{}
17:/* allocate pointers and copy data in */
18:for (i = 0; i < m; ++i){
19:     #pragma offload target(mic) in(A[i]:length(size[i])
20:         ALLOC) in(B[i]:length(size[i]) ALLOC)
21:     {}
22:}
23:#pragma offload target(mic) nocopy(A:length(m) REUSE)
24:    nocopy(B:length(m) REUSE)
25:{
26:     /* computation kernel/offloadable code region */
27:     #pragma omp parallel for private(i, j)
28:     for (i = 0; i < m; ++i) {
29:         for (j = 0; j < size[i]; ++j) {
30:             A[i][j] = a * B[i][j];
31:             ...
32:         }
33:     }
34:}
35:/* copy data out */
36:for (i = 0; i < m; ++i){
37:     #pragma offload target(mic) out(A[i]:length(size[i])
38:         FREE) nocopy(B[i]:length(size[i]) FREE)
39:     {}
40:}
```

Figure 5.2: Naive Two-Dimensional Array Offload (significantly more complex than one-dimensional case)



(a)



(b)

Figure 5.3: (a) Performance of Matrix Addition with Non-Linearized vs. Linearized Data Transfers, (b) Relationship between Number of Offload Statements (for different array components) and Data Transfer Time. (For a fixed data size, using fewer offload statements is beneficial, due to better DMA utilization and smaller memory allocation and offload overhead.)

snippet in **Figure 5.2**. Each of the memory regions is allocated and transferred independently, using a separate offload statement (in a `for` loop). Automating this is not hard, once the malloc statements, memory accesses and offload code regions have been tracked. This is similar to what CGCM [60] does, which is the state-of-the-art compiler-based communication management system for GPUs. However, this approach leads to high memory allocation overheads as well as DMA suppression (since multiple small memory regions

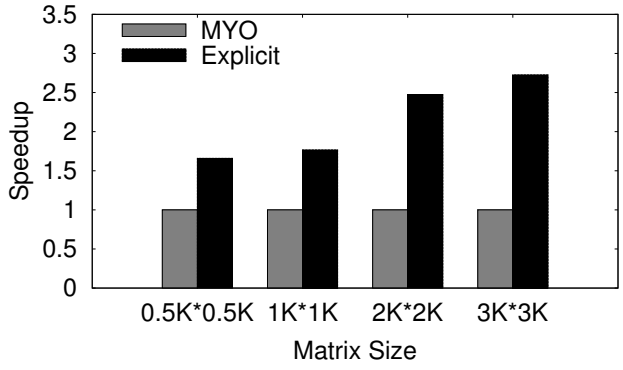
are transferred separately). Particularly, **Figure 5.3**(a) compares the performance of this approach with one where data is linearized and transferred using a single offload statement, for a matrix addition micro-benchmark. **Figure 5.3**(b) shows the impact of number of offload statements on data transfer time. The results are shown for various array sizes. For a fixed array size, using fewer offload statements results in better DMA utilization and lower offload and memory allocation overhead.

```
1: int _Cilk_shared **A;
2: int _Cilk_shared **B;
3: /* computation kernel */
4: _Cilk_shared void kernel(){
5:      #pragma omp parallel for private(i, j)
6:      for (i = 0; i < m; ++i) {
7:          for (j = 0; j < size[i]; ++j) {
8:              A[i][j] = a * B[i][j];
9:              ...
10:         }
11:     }
12:}
13:void main(){
14:     /* allocate and initialize arrays A and B */
15:     ...
16:     _Cilk_offload kernel();
17:     ...
18:}
```

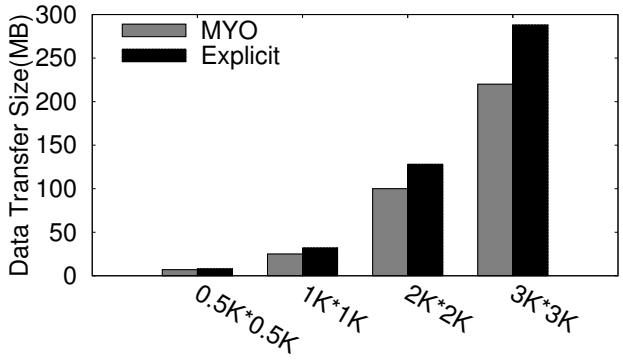
Figure 5.4: Two-Dimensional Array Offload using MYO (no explicit data transfers)

In addition to the explicit data specification model (COI), LEO also supports an implicit data transfer model and corresponding runtime mechanism (called MYO [120]) to automate data transfers between CPU and coprocessor. Any data element marked by the

`_Cilk_shared` clause is automatically synced between the two processors. In the implicit model, the offloadable code regions are marked using `_Cilk_offload`. **Figure 5.4** shows a simple example. MYO resembles state-of-the-art memory management solutions for GPU (Dymand [59] and AMM [108]), which all implement runtime data coherence mechanisms and create the illusion of virtual shared memory between the CPU and coprocessor.



(a)



(b)

Figure 5.5: (a) Performance Comparison between MYO and Explicit Data Transfers using Linearization for dgemm, (b) Total Data Transfer Size for both. (MYO transfers less data but performs worse.)

We evaluate MYO with respect to a number of benchmarks and find that explicit data transfer specification using `in/out` clauses outperforms MYO by upto 3x (**Figure 5.5** shows an example for matrix multiplication). To further understand the performance, we investigate the bottlenecks of the runtime memory management scheme and find that the mechanism that keeps tracking of dirty pages for minimizing redundant data transfers ends up imposing huge overheads. Noting that runtime mechanism can be more general and applicable for applications where all memory allocations cannot be statically tracked, it will clearly be desirable to try and optimize the existing runtime method.

5.2 Compile-time Automation of Data Transfers

Array linearization is commonly used to minimize the number of pointer indirections (and load instructions) for static arrays. For example, a two-dimensional array $A[i][j]$ would be accessed as $A[i * N + j]$ instead of $(A[i])[j]$, where N is the stride for i. The memory layout is not changed, only the memory accesses are linearized. This approach can be extended to facilitate efficient transfer of dynamically allocated multi-dimensional arrays between CPU and coprocessor, by linearizing the memory layout in addition to the memory accesses for dynamically. We introduce this approach, point out its limitations and propose efficient alternatives. We refer to the default implementation of this approach as *complete linearization* and discuss it next.

5.2.1 Complete Linearization

In *complete linearization*, all malloc statements for a given multi-dimensional array are replaced by a single malloc statement in the application source code. Instead of allocating

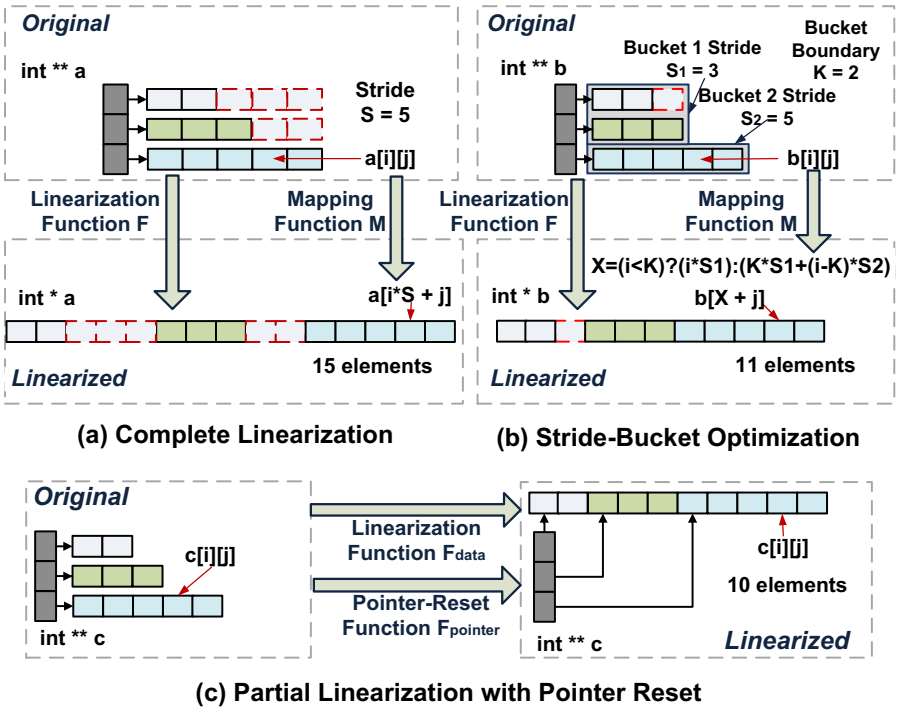


Figure 5.6: Different Linearization Schemes for Handling Data Transfers for Dynamically Allocated Multi-Dimensional Arrays

multiple small chunks of memory for the different array components, a single contiguous chunk of memory is allocated. Accordingly, the memory accesses are linearized as well, as shown in **Figure 5.7**. In essence, the complete linearization method transforms a dynamically allocated multi-dimensional array into a one-dimensional array, as shown in **Figure 5.6** (a).

Algorithm To formally state the underlying compile-time transformation: let D_m be the data layout for a multi-dimensional array in the original code, let A_m be a memory access, let D_s be the data layout for the array in the transformed code and let A_s be a memory access, our goal is to implement two functions: (i) $F : D_m \rightarrow D_s$ and (ii) $M : A_m \rightarrow A_s$.

```
1: int *A, *B;
2: int size[m];//size array for the second dimension
3: /* collect length info from malloc stmts */
4: /* and calculate stride for each dimension */
5: A_s1 = B_s1 = 0;
6: for(i = 0; i < m; ++i) {
7:      A_s1 = max(A_s1, size[i]);
8:      B_s1 = max(B_s1, size[i]);
9: }
10:A_len = A_s1 * m;
11:B_len = B_s1 * m;
12:/* allocate linearized data on CPU */
13:A = (int *) malloc(A_len * sizeof(int));
14:B = (int *) malloc(B_len * sizeof(int));
15:/* copy and allocate linearized data on coprocessor */
16:#pragma offload target(mic) in(B:length(B_len))
17:    inout(A:length(A_len))
18:{
19:     /* computation kernel */
20:     #pragma omp parallel for private(i, j)
21:     for (i = 0; i < m; ++i) {
22:         for (j = 0; j < size[i]; ++j) {
23:             /* modified memory access */
24:             A[i*A_s1 + j] = a * B[i*B_s1 + j];
25:             ...
26:         }
27:     }
28:}
```

Figure 5.7: Two-Dimensional Array Computation Offload (using complete linearization)

Let sz_0, sz_1,..,sz_k be the size of the elements in a given dimension. The malloc statement corresponding to element i in the original source code would be $malloc(sz_i * sizeof(datatype))$. For a dimension with equal-sized elements the stride value would be $s = sz_0 = sz_1,.., = sz_k$. For a dimension with variable-size elements (as in **Figure 5.6** (a)), the stride value would be chosen as $s = max(sz_0, sz_1, .., sz_k)$. For dimension d_i let the stride be s_i and the number of elements of the first dimension be m. For an n-dimensional array, the total memory size would be $total = m * s_1 * s_2.. * s_{n-1}$ and the corresponding malloc statement in the transformed source code would be $malloc(total * sizeof(datatype))$.

Let A be an n-dimensional array. Memory access $A[m_1][m_2]..[m_n]$ in the original source code is transformed into $A[m_1 * s_1 * s_2..s_{n-1} + m_2 * s_2 * s_3..s_{n-1}.. + m_n]$.

After applying functions F and M, corresponding offload statements and data transfer clauses are inserted as shown in **Figure 5.7**.

Pros and Cons As compared to allocating each row and column of the multi-dimensional structure separately, there are four distinct benefits of this approach: (i) since multiple malloc statements are replaced by a single statement, memory allocation overhead is reduced on both the CPU and coprocessor side, (ii) the overall data locality is improved because of contiguity, (iii) DMA utilization is maximized, since one large chunk of memory is transferred instead of multiple small chunks, and (iv) only one offload statement is required for data transfer.

This method has three main drawbacks. First, all memory accesses have to be identified, analyzed and modified using function M. Strong alias analysis is required. The mapping can potentially be complex and thus a source of bugs in the generated code, not to mention the loss of readability.

Second, since the subscripts are made complex, important compiler optimizations get suppressed in many cases. Optimizations like auto-vectorization and prefetching are sensitive to compiler's ability to recognize the memory access pattern. As we show later, losing important compiler optimizations (especially vectorization) can lead to significant performance loss on Intel MIC.

Third, for multi-dimensional arrays that have variable sized rows or columns, there is a trade-off between the linearized data size and the complexity of functions F and M. If we use uniform (maximum) length for each row or column, functions F and M are simplified, but redundant data transfers are introduced, as shown in **Figure 5.6** (a). If variable stride values are used for each row/column, no redundant data transfers take place, but the complexity of F and M increases substantially. The stride values need to be stored in a table, transferred to the coprocessor and looked up during memory access. For example, instead of mapping $A[i][j]$ to $A[i * s_1 + j]$, it has to be mapped to $A[i * stride\_lookup(i) + j]$. This results in increased data transfer overheads and suppression of compiler optimizations. The use of uniform (maximum) stride typically performs better than a stride-lookup approach.

To address the third drawback, we have designed an optimization to reduce the amount of redundant data transfers, without significantly increasing the complexity of functions F and M.

Stride-bucket Optimization

This optimization strives for a balance between the complexity of linearization and the amount of data transfer. The basic idea is to partition the multi-dimensional array into a finite number of *buckets* along the *first* dimension. Across these buckets, different stride values are used, whereas within each bucket, only one stride value is used. The current design uses *two* buckets, as described next.

```
1: int *A, *B;
2: int size[m];//size array for the second dimension
3: /* collect len info from malloc stmts, find bucket bound k */
4: /* A_s1:  bucket 1 stride of A; A_s2:  bucket 2 stride of A */
5: /* B_s1:  bucket 1 stride of B; B_s2:  bucket 2 stride of B */
6: A_s1 = B_s1 = A_s2 = B_s2 = 0;
7: /* determine stride values A_s1, A_s2, B_s1, B_s2 */
8: for (i = 0; i < k; ++i){
9:      A_s1 = max(A_s1, size[i]);
10:     B_s1 = max(B_s1, size[i]);
11:}
12:for (i = k; i < m; ++i){
13:     A_s2 = max(A_s2, size[i]);
14:     B_s2 = max(B_s2, size[i]);
15:}
16:A_len = A_s1*k + A_s2*(m-k);
17:B_len = B_s1*k + B_s2*(m-k);
18:/* allocate Linearized Data */
19:A = (int *) malloc(A_len * sizeof(int));
20:B = (int *) malloc(B_len * sizeof(int));
21:/* allocate and copy linearized data */
22:#pragma offload target(mic) in(B:length(B_len))
23:    inout(A:length(A_len))
24:{
25:     /* computation kernel */
26:     #pragma omp parallel for private(i, j)
27:     for (i = 0; i < m; ++i) {
28:         int A_x = (i<k) ?  (A_s1*i) :  (A_s1*k + (i-k)*A_s2);
29:         int B_x = (i<k) ?  (B_s1*i) :  (B_s1*k + (i-k)*B_s2);
30:         for (j = 0; j < size[i]; ++j) {
31:             /* modified memory access */
32:             A[A_x + j] = a * B[B_x + j];
33:             ...
34:         }
35:     }
36:}
```

Figure 5.8: Two-Dimensional Array Computation Offload (using complete linearization with stride-bucket)

Algorithm 4 CompleteLinearizationWithBucket (Multi_dim_var_set D)

1: **for** each multi-dim var $A \in D$ **do**
2: **if** A used in an offload region **and** satisfies legality checks **then**
3: D_{sub}.append(A)
4: **end if**
5: **end for**
6: **for** each multi-dim var $A \in D_{sub}$ **do**
7: /***Linearization Function F()***/
8: **for** ($i = 1$; $i <$ total dimensions n; $++i$) **do**
9: ▷ Parse malloc-sites for A, establish size table $sz_i[0..l_i]$ for each dimension i
10: **end for**
11: /*---*Apply Stride-Bucket Opt for the first dim*---*/
12: $sp_1\_table[0] = sp_2\_table[l_1] = 0$
13: /*---*Bucket boundary calculation*---*/
14: /*--- *bucket boundary: m, bucket strides: sp_1, sp_2* ---*/
15: **for** ($i = 0$; $i < l_1$; $++i$) **do**
16: $sp_1\_table[i+1]$ = max($\forall_{k=0}^{k=i} sz_1[k]$)
17: $sp_2\_table[l_1 - i]$ = max($\forall_{k=l_1-i}^{k=l_1} sz_1[k]$)
18: **end for**
19: $min\_size = MAX$
20: **for** ($i = 0$; $i <= l_1$; $++i$) **do**
21: $size = sp_1\_table[i] * i + sp_2\_table[i] * (l_1 - i)$
22: **if** $size < min\_size$ **then**
23: $m = i$, $min\_size = size$
24: $sp_1 = sp_1\_table[i]$, $sp_2 = sp_2\_table[i]$
25: **end if**
26: **end for**
27: /*---*Calculate total data size*---*/
28: $total = min\_size$
29: **for** ($i = 2$; $i <$ total dimensions n; $++i$) **do**
30: $s_i = max(\forall_{k=0}^{k=l_i} sz_i[k])$; $total = total * s_i$
31: **end for**
32: ▷ Delete old malloc-sites
33: ▷ Insert linear-alloc: A = malloc($total *$ sizeof($type$))
34: /***Mapping Function M()***/
35: **for** each memory access $A[m_1][m_2]..[m_n]$ **do**
36: A_x = $(m_1 < m)?(sp_1 * m_1) : (sp_1 * m + sp_2 * (m_1 - m))$
37: ▷ Change access-site to $A[A_x * s_2 * ..s_{n-1} + m_2 * s_2 * ..s_{n-1} + ...m_n]$
38: **end for**
39: /*---*Generate offload code for coprocessor*---*/
40: **for** each offload region R **do**
41: **if** A is used by R **then**
42: ▷ Generate data transfer and offload clauses for coprocessor
43: **end if**
44: **end for**
45: **end for**

Let sz_0, sz_1,..,sz_l be the size of elements in the first dimension. It is partitioned into two buckets P_1 and P_2, containing m and $l - m$ elements respectively. For P_1, the stride value $s_{P1} = max(sz_0, sz_1, .., sz_m)$. Similarly for P_2, the stride value $s_{P2} = max(sz_{m+1}, sz_{m+2}, .., sz_l)$. The element m serves as the bucket boundary. It is picked such that the amount of redundant data allocation (henceforth *holes*) is minimized, as described next.

Let i be the bucket boundary. Stride value for first bucket $s_{P1} = max(sz_0, sz_1, .., sz_i)$, stride value for second bucket $s_{P2} = max\ (sz_{i+1}, sz_{i+2}, .., sz_l)$. The size of the array would be $size = s_{P1} * i + s_{P2} * (l - i)$. Element m is picked as the bucket boundary, such that $size$ is minimized for m. This algorithm runs in $O(l)$ time (shown in Algo. 4).

Functions F and M are suitably tailored for the stride-bucket optimization. **Figure 5.6**(b) shows an example for a two-dimensional array– the bucket boundary is 2, the two stride values are 3 and 5 respectively. As compared to the the memory layout in **Figure 5.6**(a), the new memory layout in **Figure 5.6**(b) is around two-thirds the size. The mapping function M now contains a branch operation– the stride is determined based on which of the two buckets the element belongs to (as shown in **Figure 5.8**). If the bucket boundary is k, the stride for the first bucket is s_1 and the stride for the second bucket is s_2, element $A[i][j]$ would be accessed as $A[index + j]$, where $index = (i < k)?(i * s_1) : (k * s_1 + (i - k) * s_2)$.

5.2.2 Partial Linearization with Pointer Reset

Complete linearization method suffers from three main drawbacks, as mentioned earlier. The first and second drawbacks arise from modification of memory accesses (i.e., function M). The third drawback arises from the use of uniform strides during memory

allocation (i.e, function F), which allows simplification of M but imposes data transfer overheads, since holes are included in the memory layout.

We note that all three drawbacks can be eliminated if: (i) memory accesses do not have to be modified, and (ii) a single contiguous chunk of memory can be allocated for the entire multi-dimensional array without any holes in it.

Algorithm Our partial linearization approach is based on two simple observations. First, only the last dimension of a multi-dimensional array contains actual data, all the other dimensions only contain pointer addresses to get to this data. Therefore, if the data in the last dimension is linearized, i.e, allocated as a single contiguous chunk of memory and transferred to the coprocessor, the memory allocation and setting up of pointers can be done separately on both CPU and the coprocessor. Second, pointer structure of the multi-dimensional array can be reconstructed on the coprocessor side by simply replicating CPU-side code. The pointer sizes do not have to be transferred.

There is no mapping function M in this approach, since memory accesses are not modified. Data layout transformation function F is split into two component functions F_{data} and $F_{pointer}$, as described next.

The method (shown in Algo. 5) comprises three main steps. In the first step, the function F_{data} is applied, i.e, malloc statements for a given multi-dimensional array A are parsed and code is generated for computing the total data size ($total\_sz$) of the array by adding up the size of each element in the last dimension. A malloc statement is generated to allocate a memory chunk $data_A$ of $total\_sz$. **Figure 5.9** shows an example for a two-dimensional array– the original code structure for malloc statements is leveraged for generating the code for calculating $total\_sz$.

Algorithm 5 PartialLinearizationPointerReset(Mul_dim_var_set D)

for each multi-dim var $A \in D$ **do**
  if A used by an offload region **and** satisfies legality checks **then**
    D_{sub}.append(A)
  end if
end for
for each multi-dim var $A \in D_{sub}$ **do**
  /*__Linearization Function $\mathbf{F_{data}}$()__*/
  ▷ Parse malloc stmts of A
  /*---*Calculate total data size*---*/
  ▷ Replicate the malloc stmts for last dimension
  $total\_sz = 0$
  for each replicated malloc stmt:
    $A[m_1]..[m_n]$ = malloc($size_i$*sizeof($type$)) **do**
    ▷ Replace it by: $total\_sz$ += $size_i$
  end for
  ▷ Insert linear-alloc: $data_A$ = malloc($total\_sz$ * sizeof($type$))
  /*__Pointer-Reset Function $\mathbf{F_{pointer}}$()__*/
  /*---*Allocate and reset pointers*---*/
  $pda = data_A$
  for each original malloc-site for last dimension:
    $A[m_1]..[m_n]$ = malloc($size_i$*sizeof($type$)) **do**
    ▷ Replace it by:
      $A[m_1][m_2]..[m_n] = pda$, pda += $size_i$
  end for
  /*---*Generate offload code for coprocessor*---*/
  ▷ Generate $data_A$ malloc clause on coprocessor
  ▷ Replicate $F_{pointer}$() code on coprocessor
  for each offload region R **do**
    if A is used by R **then**
      ▷ Generate data transfer and offload clauses for coprocessor
    end if
  end for
  ▷ Apply data reuse and hoisting opt for $data_A$
end for

```
1: #define ALLOC alloc_if(1) free_if(0) //allocate data
2: #define REUSE alloc_if(0) free_if(0) //keep data persistent
3: #define FREE alloc_if(0) free_if(1) //free data
4: int **A, **B;
5: int size[m];//size array
6: /* calc total data size from malloc stmts for last dim */
7: for (i = 0; i < m; ++i)
8:      total_sz_A += size[i]; total_sz_B += size[i];
9: /* allocate linearized data on CPU */
10:int *data_A = (int *) malloc(total_sz_A * sizeof(int));
11:int *data_B = (int *) malloc(total_sz_B * sizeof(int));
12:/* allocate and reset pointers */
13:A = (int **) malloc(m * sizeof(int *));
14:B = (int **) malloc(m * sizeof(int *));
15:int *pda = data_A; int *pdb = data_B;
16:for (i = 0; i < m; ++i)
17:     A[i] = pda; pda += size[i]; B[i] = pdb; pdb += size[i];
18:/* allocate linearized data on coprocessor */
19:#pragma offload target(mic) nocopy(data_A:length(total_sz_A)
20:    ALLOC) nocopy(data_B:length(total_sz_B) ALLOC)
21:{}
22:/* allocate and set pointers on coprocessor */
23:#pragma offload target(mic) nocopy(A:length(m) ALLOC)
24:    nocopy(B:length(m) ALLOC)
25:{
26:     int *pda = data_A; int *pdb = data_B;
27:     for (i = 0; i < m; ++i){
28:         A[i] = pda; pda += size[i];
29:         B[i] = pdb; pdb += size[i];
30:     }
31:}
32:/* copy data in/out */
33:#pragma offload target(mic) inout(data_A:length(total_sz_A)
34:    REUSE) in(data_B:length(total_sz_B) REUSE )
35:{
36:     /* computation kernel */
37:     #pragma omp parallel for private(i, j)
38:     for (i = 0; i < m; ++i)
39:         for (j = 0; j < size[i]; ++j)
40:             A[i][j] = a * B[i][j]; ...
41:}
```

Figure 5.9: Two-Dimensional Array Computation Offload (using partial linearization with pointer reset)

In the second step, the function $F_{pointer}$ is applied, i.e., malloc statements for the last dimension are replaced by assignment statements, in order to set up the pointers into the contiguous chunk of memory allocated in the first step. As shown in **Figure 5.9**, the statement $A[i] = (int*)malloc(size_i * sizeof(int))$ is replaced by $A[i] = pda$, $pda = pda + size_i$, where pda is a moving pointer. It is initialized to the starting address of $data_A$ (i.e, $A[0]$) and incremented with every pointer assignment. **Figure 5.6**(c) illustrates the data layout transformation for a two-dimensional array.

In the third step, offload statements and data transfer clauses are generated for transferring the memory chunk $data_A$ to the coprocessor and back. The code for pointer allocation and construction (i.e, $F_{pointer}$) is replicated on the coprocessor side. Therefore, no stride information needs to be transferred.

As another note, placement of memory allocation statements and data transfer clauses in the code is important for performance. In our implementation, we hoist malloc statements, offload statements and data transfer clauses as far up the call graph as possible. By hoisting statements outside loops and up the call graph, redundant execution is minimized and memory reuse (across multiple offloads) is enabled.

Legality Checks: A compiler can apply partial linearization with pointer reset only if certain conditions are met. We summarize these conditions in **Table 5.2** and explain them here. The first condition, referred to as the *Homogeneity Check*, ensures that the elements in the multi-dimensional array are of the same size. For example, if the code is in $C++$, we may have the *polymorphism* issue. An existing data flow analysis reported in the literature [117] is used for this purpose.

The second condition is *single malloc site check*, where the goal is ensure that there is no memory reallocation or conditional memory allocation. These possibilities make our

Case Name	Description
I Homogeneous	Multi-dimensional array must be homogeneous, *i.e.*, allocations must be with elements of the same type/size
II Single Malloc Site	A pointer must have only one malloc stmt associated, *i.e.*, no conditional allocation or reallocation
III Escaping Pointers	Pointers to allocated sections must not escape the current scope

Table 5.2: Legality Check Cases

transformation more complicated (if at all applicable), and we prefer not to apply them in our implementation. For performing this check, malloc statements and memory accesses are tracked using *use-def* chains for arrays/pointers that are used in offloadable code regions, as identified by liveness analysis module within Apricot [115]. We collect all malloc sites for a specific multi-dimensional array, and check whether any pointer is represented multiple times.

The third condition is *Escaping Pointers Check*. A pointer used to allocate a lower-dimensional section of the array must not escape the current context, because if it does, it becomes extremely hard to track possible reallocation. This check is implemented using *alias analysis*.

These legality conditions are checked by our source-to-source compiler for each array. If an array fails to satisfy one or more conditions, it is annotated as such and handled by the runtime memory management system, as described in Section 5.3.1. For most scientific computing benchmarks, these legality conditions hold and our proposed approach can be applied.

Discussion: With the partial linearization approach, the memory accesses do not have to be analyzed or modified. This significantly reduces the complexity of the analysis and

the resulting source code. The chances of introducing bugs during the transformation are minimized and code readability is maintained. Since original subscripts are retained, the compiler can choose to linearize the array subscripts as and when it deems fit. As a result, compiler optimizations remain unaffected, which is the key to performance.

There are no redundant data transfers or holes. Only one contiguous chunk of memory is allocated and transferred resulting in good data locality and full utilization of DMA.

As compared to the complete linearization, partial linearization has one drawback– it introduces code execution overhead on the coprocessor for pointer reconstruction. Our experiments show that this overhead is easily offset by the gains. This is a hybrid approach that combines simplicity with performance.

5.2.3 Interaction with Compiler Optimizations

Our source-to-source translator (or another comparable system) depends upon the native compiler (ICC in the case of Xeon Phi) for accelerator for obtaining performance. Our experiments have shown that the various optimizations performed by the native compiler can have a far more significant impact on the overall performance than the overheads of data transfer and other operations associated with the offload. As we have stated throughout, one of the critical considerations in automating handling of data transfers is preserving optimizations that would normally be performed by the compiler.

In Intel MIC (Xeon Phi), the SIMD width of each core is 512-bit, which means up to 16 floating point operations can be executed in one cycle on each of its 60 cores. This makes vectorization crucial for performance. Also, with increasing parallelism, memory accesses can become the bottleneck, and therefore, software prefetching is very important. Loop optimizations such as distribution, tiling, and interchange can also significantly impact performance, especially when they enable additional vectorization or prefetching. A

```
 1: int ***A, ***B;
 2: /* allocate and initialize arrays A and B */
 3: ...
 4: #pragma omp parallel for private(i, j, k)
 5: for (i = 0; i < M; ++i){
 6:      for (j = 0; j < N; ++j){
 7:          for (k = 0; k < X; ++k){
 8:              C[i*Cs2 * Cs1 + j * Cs1 + k]
 9:                  = A[i * As2 * As1 + j * As1 + k]
10:                  + B[i * Bs2 * Bs1 + j * Bs1 + k];
11:              ...
12:          }
13:      }
14:}
```

Figure 5.10: Vectorization Suppression Case I, abstracted from 27stencil: 3-D Array Addition (after complete linearization)

key advantage of partial linearization is that original subscripts are not modified, whereas, complete linearization introduces more complex subscripts. While theoretically a compiler should be able to handle complex linearized subscripts, in practice, product compilers often fall short, due to aliasing, pointer arithmetic and complex interactions between the different optimizations. We have verified this for the latest version of ICC as of writing this paper, as we now show through a couple of examples.

Figure 5.10 shows an example with three-dimensional arrays inside an OpenMP loop. From the optimization reports, we see that for the version with non-linearized subscripts, data dependencies are correctly resolved and the innermost loop is vectorized. While for the linearized version, auto-vectorization does not kick in due to false data dependence. We observe the same behavior for the example in **Figure 5.11** involving a structure and non-unit stride. For the example in **Figure 5.10**, software prefetching does not kick in. For

the corresponding version with non-linearized subscripts, 4 cache lines are prefetched for the outer-most loop and 24 lines are prefetched for the inner-most loop. We continue this discussion in Section 5.4.

```
 1: typedef struct{double W, X, Y;} point;
 2: point *p = (point *) malloc(M * N * sizeof(point));
 3: #pragma omp parallel for
 4: for (i = 0; i < M; ++i){
 5:       for (j = 0; j < N; ++j){
 6:           p[i*N+j].W = i+j+0.1;
 7:           p[i*N+j].X = i+j+0.2;
 8:           p[i*N+j].Y = i+j+0.3;
 9:       }
10:}
11:double sum = 0.0;
12:#pragma omp parallel for reduction(+:sum)
13:for (i = 0; i < M; ++i){
14:       for (j = 0; j < N; ++j){
15:           sum += p[i*N+j].W * p[i*N+j].X
16:                * p[i*N+j].Y;
17:       }
18:}
```

Figure 5.11: Vectorization Suppression Case II, from 330.art: Struct and Non-Unit Stride Access (after complete linearization)

5.3 Runtime Memory Management

As we discussed in the previous section, an optimization like partial linearization may not apply in some cases, if all pointers cannot be properly resolved. Thus, as an enhancement to the static approach, we present both a runtime approach, and an integrated static and runtime approach.

Xeon Phi currently supports an implicit data transfer model and corresponding runtime mechanism (called MYO) to automate data transfers between CPU and coprocessor. MYO stands for *Mine Yours Ours* and provides a virtual shared memory abstraction for the CPU and coprocessor. Any data element annotated with `_Cilk_shared` is allocated in a memory region reserved for MYO, which is automatically synchronized between the CPU and coprocessor. The memory region starts at the same virtual address on both the CPU and coprocessor. This creates the illusion of a shared memory. Particularly, MYO allows seamless sharing of complex pointers and data structures, where no data marshaling is required, and address pointers in the virtual shared memory region are valid on both CPU and the coprocessor. MYO uses the release consistency [43] protocol for handling data transfers, where memory updates are kept completely local till a release point. At release point all prior stores are guaranteed to be globally visible. At an acquire point, all stores that are globally visible are synced up with local memory. The minimum unit of synchronization is an *arena*, and each data element is explicitly assigned to an arena.

As part of our system, we modify the coherence mechanism in MYO, such that the dirty pages are not tracked. The motivation for this was a set of experiments, which revealed that the mechanism used for keeping track of dirty bits imposes huge overheads, as every read and write operation has to be monitored. Particularly, if two arenas A and B are allocated in the virtual shared memory region, and there are no writes to arena B on the coprocessor side, only data associated with arena A will be communicated from coprocessor to CPU at release point, if the original coherence mechanism is used. In the modified coherence mechanism, both arenas will be synchronized, but overheads of tracking the operations are avoided. This simple modification results in significant performance improvement, despite an increase in the amount of data transferred. This is because an increase in the amount

of data transferred does not necessarily imply increase in execution time, because DMA allows overlap of computation with communication.

As a case where this scheme will apply, we consider the following analysis. Let us assume that data of size s_1 is allocated in the virtual shared memory region and data of size s_2 ($\leq s_1$) is written to. The original coherence mechanism may perform better than the new one, if the ratio s_2/s_1 is very small. However, the ideal solution would be to use a hybrid approach– depending on the properties of the data structures and the data flow in the program, selective tracking may be enabled. Such a solution is presented next.

5.3.1 Combined Static and Runtime Approach

The motivation for this combination method is the need for improving the generality of the static method, and the performance of the method based entirely on runtime tracking. The basic idea is as following: if the offloaded data structure passes the legality check, our source-to-source compiler generates the data transfer code for the corresponding offload regions automatically as an optimization. Otherwise, the data structure is marked as shared data structure (`_Cilk_shared`), and the offload region is marked as shared offload region (`_Cilk_offload`) by default. All the shared structures and offload regions will be managed by our optimized MYO runtime library automatically. The formal algorithm is shown in Algo. 6, and we can explain it below.

Given a C program (potentially annotated with OpenMP), the pre-processor identifies the set of data variables D that need to be copied into and out of the coprocessor using liveness analysis [115, 46], assuming that all offloadable code regions C have already been identified. All variables in D are declared to the shared. Now, using currently implementation, MYO can automate data transfers for all the variables in D and the application

Algorithm 6 Integrat($Mul\_dim\_var\_set\ D, Off\_set\ C$)

for each multi-dim var $Mul\_Var \in D$ **do**
  ▷ Insert $\_Cilk\_shared$ before Mul_Var decl
end for
for each offload region $Off\_Reg \in C$ **do**
  ▷ Insert $\_Cilk\_offload$ before Off_Reg
end for
▷ Linearize the possible vars, store into D_{sub}
D_{sub} = Compiler_Decider(D)
for each multi-dim var $Mul\_Var \in D_{sub}$ **do**
  ▷ Replace $\_Cilk\_shared$ by $\_explicit\_transfer$
end for
for each offload region $Off\_Reg \in C$ **do**
  if all $Off.vars \in Off\_Reg$ also $\in D_{sub}$ **then**
    ▷ Replace $\_Cilk\_offload$ by $\_pragma\_offload$
  end if
end for
▷ Generate the final code with offload directives

can execute successfully, though performance will likely be poor. Thus, in the next step, the pre-processor short-lists all variables in D for which explicit data transfer clauses can automatically be inserted at the compile time. This is done by analyzing the memory allocation statements and memory access sites for each variable and running a set of legality checks (as described in Section 5.2.2). For all the variables in D_{sub} that can be handled at compile-time, the pre-processor also identifies the corresponding offloadable code regions C_{sub} where they are accessed, and marks them for explicit transfer.

When the source-to-source transformation is applied, it operates on variables annotated with `_explicit_transfer` and generates corresponding memory allocation and data transfer statements. The #pragma offload directive is inserted for code regions annotated with `_pragma_offload` along with the corresponding `in/out` clauses. The resulting source code uses both explicit data transfers as well MYO runtime memory management. **Figure 5.12** shows an example.

```
1: /* two dimensional array handled using explicit data transfer */
2: int **A;
3: /* linked list handled by implicit data transfer (MYO) */
4: struct node{ int x; struct node *next;} list;
5: list _Cilk_shared *head;
6: /* computation kernel 1 */
7: _Cilk_shared void kernel1(){
8:      /* operations on list */
9:      ...
10:}
11:void main(){
12:     _Cilk_offload kernel1();
13:     /* array A linearized using pointer reset approach */
14:     ...
15:     #pragma offload target(mic) inout(A_data:length(A_len)
16:     REUSE) {
17:         /* computation kernel 2 */
18:         #pragma omp parallel for private(i, j)
19:         for (i = 0; i < m; ++i) {
20:             for (j = 0; j < n; ++j) {
21:                 /* operations on array A*/
22:                 ...
23:             }
24:         }
25:     }
26:     ...
27:}
```

Figure 5.12: Integrating Compile Time and Runtime Solutions: Simultaneous Use of Explicit and Implicit Memory Management

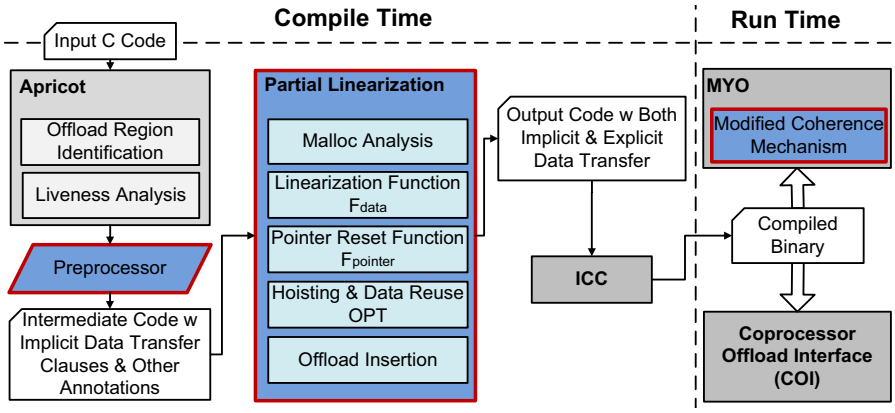


Figure 5.13: Overall Solution Architecture

5.4 Evaluation

In this section, we evaluate our compile-time and runtime solutions in detail, and compare our CPU-MIC solution with multi-core CPU solution.

5.4.1 Implementation

We have implemented the compile-time solution for automatic insertion of data transfer clauses using *partial linearization with pointer reset* approach described in Section 5.2.2. It has been implemented as a source-to-source transformation on top of the Apricot [115] framework. Apricot provides modules for liveness analysis, handling of one-dimensional arrays and identification of offloadable code regions. We have also modified the coherence mechanism in MYO as described in Section 5.3. The solution architecture is shown in **Figure 5.13**.

Benchmark	Source	Description
MG	NAS Parallel in C	Multi-Grid on meshes
FT	NAS Parallel in C	3D fast Fourier Transform
330.art	SPEC OMP	Image recognition by neural network
Heat3D	Heat 3D	Heat transfer simulation
27stencil	EPCC	3-d stencil kernel
convolution	CAPS OpenACC	2-d stencil kernel
dgemm	LINPACK	Double general matrix multiplication

Table 5.3: Benchmarks

5.4.2 Experimental Methodology

The test suite consists of seven C benchmarks from different sources (shown in **Table 5.3**). These benchmarks contain dynamically allocated multi-dimensional arrays/multi-level pointers and OpenMP parallel regions. We particularly note that the first three benchmarks, `MG`, `FT`, and `330.art`, are all more than $1,500$ lines each (`330.art` is more than 2000), and are used to demonstrate the applicability of our approach (and the current implementation) on full-scale applications. All experiments were conducted on a Xeon E5-2609 server equipped with an Intel MIC (Xeon Phi) card and the necessary software. Xeon E5-2609 has 8 cores, each running at 2.40GHz with 2 threads per core. Xeon Phi has 61 cores each running at 1.05GHz with four threads per core, a total of 32MB L2 cache and 8GB GDDR5 memory. Our source-to-source compiler is invoked on each benchmark and the transformed source code is compiled with ICC at `-O3` with additional compiler flags (`-openmp -parallel [-ansi-alias] [-fno-alias]`).

There are several objectives in our evaluation. We evaluate the overall performance of our partial linearization with pointer reset solution, and compare it with the runtime method through MYO, as well as the optimized complete linearization (with stride-bucket).

Besides comparing the execution times, the amount of data transferred over PCIe is also measured and reported. To demonstrate the benefits of using the accelerator after applying our solution, we also evaluate the performance of our best multi-core CPU+MIC version over the multi-core CPU version.

We also individually evaluate the benefits of particular optimizations. Performance of the runtime memory management system (MYO) is evaluated with and without our optimization, and similarly, the performance of the complete linearization approach is evaluated with and without the stride-bucket optimization.

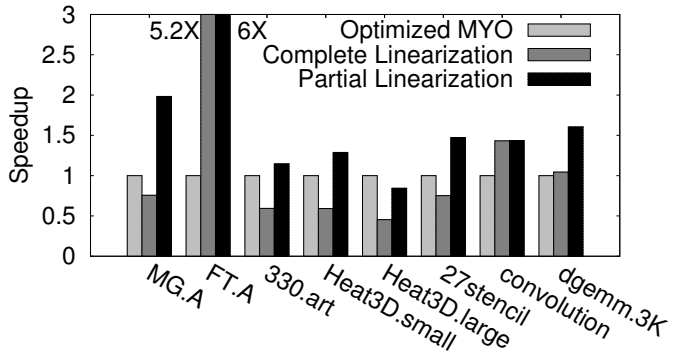
5.4.3 Results and analysis

Overall Performance Evaluation

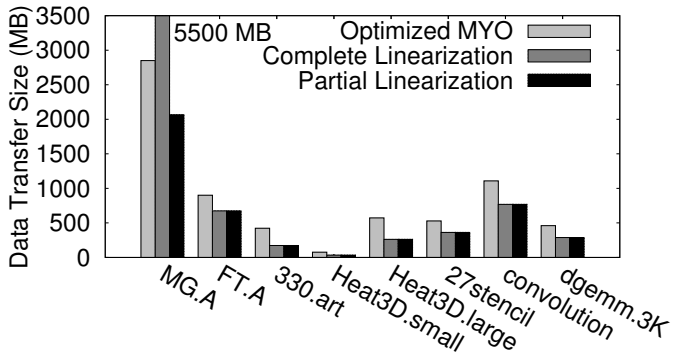
The overall performance comparison is shown as **Figure 5.14**. **Figure 5.14**(a) compares the performance of complete linearization (further optimizes using the stride-bucket method) with our partial linearization approach. 1.6x-2.6x speedup is obtained with the partial linearization approach for five out of the seven benchmarks, whereas nearly 1.25x speedup is observed for the other two.

While the approach benefits all benchmarks, the reasons for performance gains differ considerably. We now explain these, referring also to data transfer volumes (Fig 5.14(b)), and details of compiler optimizations enabled for different versions (**Table 5.4**).

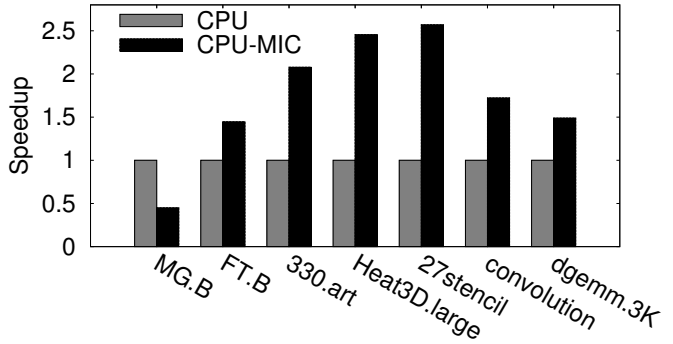
For `MG`, majority of the speedup comes from reduction in the total amount of data transferred as shown in **Figure 5.14**(b), since it is a data-intensive benchmark with variable-size rows. We also notice more aggressive prefetching for the partial linearization version: total number of cache lines prefetched goes up from 131 to 542 (**Table 5.4**). For `Heat3D` and `27stencil`, the main loop gets vectorized for the partial linearization version, resulting in a 2x speedup. Number of prefetched cache lines goes up from 32 to 72 for `Heat3D`.



(a)



(b)



(c)

Figure 5.14: Performance Comparisons for all Benchmarks: Optimized MYO, Complete Linearization with Stride-Bucket, and Partial Linearization Compared with Respect to (a) Execution Time and (b) Total Data Transfer Sizes; (c) Execution Time Comparison between Multi-Core CPU, and Multi-Core CPU+MIC for Large Input Data Sizes. The CPU-MIC Versions are Obtained with our Partial Linearization

Benchmark	Vectorization		Prefetching		LoopDistribution	
	Complete Linear	Partial Linear	Complete Linear	Partial Linear	Complete Linear	Partial Linear
MG	10	10	131	542	0	3
FT	15	16	70	74	0	3
330.art	1	12	50	98	2	0
Heat3D	2	3	32	72	0	0
27stencil	2	3	40	48	0	12
convolution	1	1	10	10	0	0
dgemm	1	1	14	17	0	0

Table 5.4: Impact of the Two Linearization Approaches on Key Compiler Optimizations

We also notice a significant increase in loop distribution for `27stencil`: with the pointer reset version 12 loops are distributed as opposed to none for complete linearization. Both these benchmarks contain three-dimensional arrays. For `330.art` a total of 12 loops are vectorized with partial linearization, as opposed to 1 for complete linearization. Prefetched cache lines go up from 50 to 98. This benchmark contains a two-dimensional struct array (**Figure 5.11**). For `dgemm` the outer loop gets vectorized for the pointer reset version, while the inner loop is vectorized for the complete linearization version. With outer loop vectorization the performance goes up by 1.5x.

Figure 5.14 (a) also compares the performance of optimized MYO with both complete linearization (using stride-bucket) and pointer reset approach. Optimized MYO frequently outperforms complete linearization. However, partial linearization with pointer reset comes out on top. It performs 1.5x-2.5x faster than optimized MYO for most benchmarks and around 6x faster for `FT.A`.

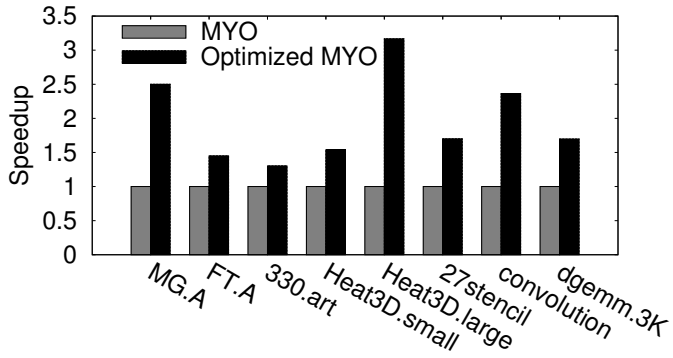
Next, data transfer volumes are shown in **Figure 5.14** (b). Except for `MG`, pointer reset and complete linearization have identical data transfers. Optimized MYO transfers around 1.5x more data on average for most benchmarks.

Finally, **Figure 5.14**(c) shows the performance of the best CPU-MIC version for each benchmark (obtained with the partial linearization approach) and compares it with the original CPU version. The original CPU version uses 16 threads, while the CPU-MIC version uses 16 threads on the CPU and around 240 threads on Intel MIC. The CPU-MIC version runs 1.5x-2.5x faster for six out of the seven benchmarks. No gains are obtained for `MG`, which is a highly data intensive benchmark. Considering the benefits of using partial linearization that we reported earlier, it can be seen that most performance gains from the use of the coprocessor will not be possible without optimizing data transfers.

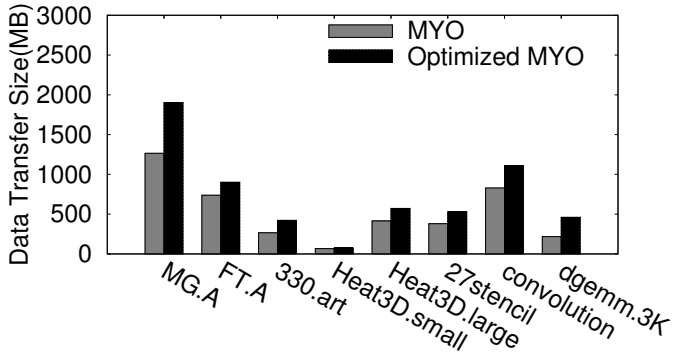
Optimizations Evaluation

In our overall evaluation above, we use the optimized version of runtime MYO solution and complete linearization (with stride-bucket) solution. We evaluate these optimizations as following to validate their efficacy.

Figure 5.15 (a) compares the performance of MYO with optimized MYO. **Figure 5.15** (b) shows the total amount of data transferred for the two MYO versions. With the modified MYO, the amount of data transfer increases by 1.5x on average (most of it comes from the increase in communication from coprocessor to CPU). This is because dirty pages are not tracked in the modified coherence mechanism. Despite an increase in data transfer, significant performance gains (1.5x-3.2x) are observed with modified MYO. There is a noticeable drop in the execution time of coprocessor side code with the modified coherence mechanism. Also, we notice a very small increase in the time spent on data transfers, which can be attributed to DMA.



(a)



(b)

Figure 5.15: Optimized MYO vs. MYO: (a) Execution Time, (b) Total Data Transfer Size

Complete Linearization: Figure 5.16 (a) compares the performance of the complete linearization approach with the optimized one using stride-bucket, for varying input data sizes (class=W,A,B). MG is the only benchmark in our test-suite containing arrays with variable-size elements in the first dimension. Optimized linearization approach yields more than 1.5x speedup for classes A and B. There is no difference in the array data size between classes A and B, hence similar speedup is observed. Xeon Phi coprocessor runs out of memory for class C and above when using complete linearization. Data transfers are shown in **Figure 5.16** (b). Stride-bucket linearization results in around 1.8x reduction in data size.

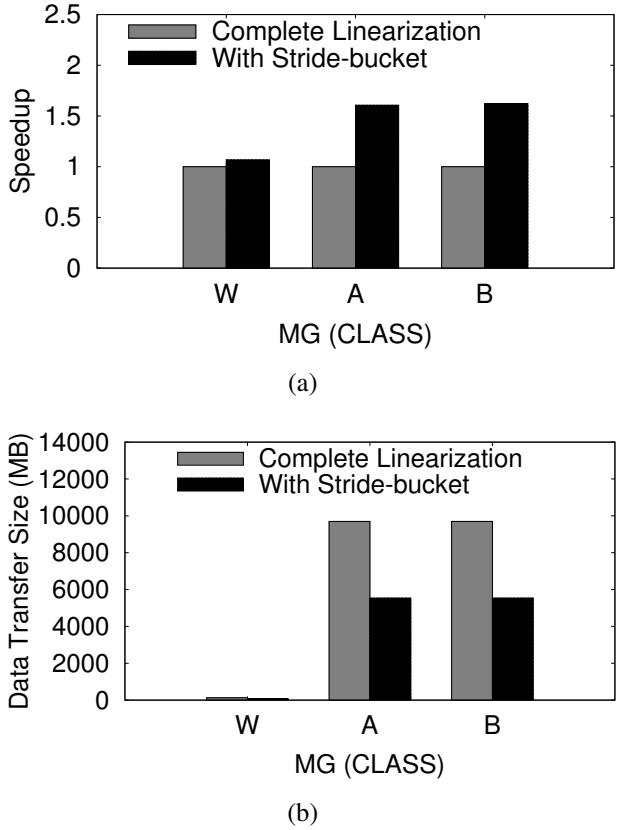


Figure 5.16: Performance of Complete Linearization with and without Stride-Bucket Optimization for Varying Input Data Sizes: (a) Execution Time, (b) Total Data Transfer Size

5.5 Related Work

Over the last 5-6 years, many compilation systems have been built for accelerators, which have also addressed the problem of data transfers from host to accelerators. For example, OpenMPC [81] compiler automatically converts OpenMP code to GPU kernels and in the process inserts data transfer clauses. Baskaran *et al.* [5] do the same in system where the primary focus is on using a polyhedral framework for memory management.

The PGI [46] compiler also automatically inserts data transfer clauses for OpenAcc applications. JCUDA [139] based on Java can automatically transfer GPU function arguments between CPU and GPU memories, however, it requires annotations indicating the live-in/out information for arguments. Because these, as well as other comparable systems, generate the accelerator (CUDA) code also, interaction of the offload mechanism with optimizations inside the native compiler is not a concern for these systems.

Apricot [115] automatically inserts LEO offload and data transfer clauses in OpenMP applications for the Intel MIC coprocessor, using liveness analysis to determine data variables that need to be copied into and out of the coprocessor. It does not handle pointer arithmetic, aliasing or pointer indirection for dynamically allocated data. Similarly, statically allocated arrays can be automatically handled by ICC for Intel MIC without additional support. The challenge we have addressed here is to handle dynamically allocated multi-dimensional arrays and other structures with multi-level pointers.

Our work is closest to CGCM [60], which is a state-of-the-art compiler-based data management and optimization system for GPUs. CGCM incorporates a runtime library that tracks memory allocation at runtime and replicates allocation units on the GPU. It supports two key optimizations– *map promotion* and *alloca promotion*, to hoist runtime library calls and local variables up the call graph. However, CGCM does not linearize the heap. As a result, all the memory regions allocated for a multi-dimensional array or multi-level pointer are allocated and transferred separately. This would suffer from high memory allocation overheads and DMA suppression, as confirmed by our experiments for Intel MIC.

DyMand [59], AMM [108], and ADSM/GMAC [42] are all runtime systems for automatic memory management for GPUs. Each of them implements runtime coherence mechanisms for supporting a virtual shared memory abstraction for the CPU and the GPU. They bear strong resemblance to MYO [120] and inherit the properties of software DSM [82, 7] and PGAS [121, 16] to some extent. AMM uses compiler analysis to optimize placement of coherence checks, but tracks read and write operations in order to monitor coherence status of Rails, similar to MYO's Arenas. We have implemented our optimizations on top of MYO, and a novel component of our effort is integrated static and runtime optimizations.

Our work has some similarities with previous efforts on data layout optimizations [143, 19, 67, 78, 80]. However, our work is distinct in the sense that the context is coprocessors. By modifying the malloc sites and allocating one large chunk of memory instead of numerous small chunks for the array components distributed over memory space, we minimize memory allocation overheads (for both CPU and coprocessor), maximize DMA utilization for fast and asynchronous data transfer over PCIe and improve cache performance for both CPU and coprocessor. By retaining original memory accesses in the code, we allow ICC to be able to apply optimizations for multi-dimensional arrays.

5.6 Summary

This Chapter has addressed the problem of automating and optimizing data transfers for coprocessors, with specific emphasis on dynamically allocated multi-dimensional arrays and other data structures with multi-level pointers. Our work includes a novel compiler-based approach, *partial linearization* with *pointer-reset*. The benefits of this approach include reduced data transfer volumes, use of DMA, reduced overheads of memory allocations, and most importantly, no modification to the memory access subscripts, which turns

out to be crucial for preserving key optimizations from the native compiler for the coprocessor. This approach outperforms complete linearization by 1.6x-2.5x on average. We also devise a *stride-bucket* approach for optimizing the performance of the linearization method.

Because the static approach is not completely general, we also consider runtime solutions, specifically in the context of Xeon Phi. We optimize the performance of MYO by modifying the coherence mechanism, specifically trading additional data transfers for reduced overheads of tracking dirty pages. This results in a significant speedup - 1.5x-3.2x. Finally, we describe a way to integrate the static and runtime by selectively inserting explicit data transfer clauses when possible and using shared memory otherwise.

The most insightful observation from our work is that optimizations from the native compiler can have a far more significant impact on the overall performance than the overheads of data transfers and other operations associated with offload. Largest gains from our pointer-reset approach arise because, as compared to other solutions, it enables vectorization, prefetching, and loop distribution by the native compiler for the coprocessor.

Chapter 6: Compiling Dynamic Data Structures in Python to Enable the Use of Multi-Core and Many-Core Libraries

Another important application scenario of dynamic allocated data structures is in very high level programming languages like Python. As the productivity is increasingly significant for *Scientific Computing*, there is a high demand of applying these languages for relieving the programmers' workloads.

While many efforts focus on providing low level extensions to these languages, and using them more like *glue* languages, our work claims that an automatic or semi-automatic compilation framework can be built to adapt these high level languages down to multi-core and many-core libraries to bridge the gap between high productivity and high performance. We implement our idea in Python, a very popular high level script language. During this process, dynamic allocated data structures in Python are required to be transformed into dense memory buffer to be passed to the low level *HPC* libraries, and our layout optimization work proposed in previous section is apparently a good candidate for such procedure. Moreover, to enable such procedure more efficiently, we have developed several new algorithms. The key contributions include a *demand-driven* inter-procedural version of an existing Partial Redundancy Elimination (PRE) algorithm [109], and an algorithm for determining homogeneity of a list.

In this Chapter, we discuss these topics in detail.

6.1 Challenges and Overview of Our Work

In this section, we will introduce the performance issues of Python, and give an overview to our translation framework.

6.1.1 Python and Performance Issues

While our work is applicable to all languages where dynamic data structures are used, the techniques we have developed and implemented have been motivated by features of Python. Python has been rapidly gaining popularity because of its support for high productivity and easy learning curve. This enables programmers to focus on developing and expressing algorithms, rather than programming itself. While it provides high productivity, performance efficiency of the applications developed using Python is not very good. Thus, for HPC applications, where performance is an important issue, the use of Python creates several challenges. At the same time, programmer productivity has become an important concern within HPC as well, promoting use of Python and similar languages, including specialized parallel languages like X10 [17]. One approach is to use language extensions and/or low level libraries to help improve performance. Successful projects in this area include *NumPy* [107], *SciPy* [66], *PyMPI* [97], *PyCUDA*, and *PyOpenCL* [73], among others.

The reasons for the low efficiency of Python arise because of multiple related reasons. The fact that Python is interpreted and not compiled is clearly a big factor. Moreover, one of the most attractive features of Python, the rich support for dynamic data structures, like *list*, *dictionaries*, and others, adds significant overheads. Dynamic typing, which further gives flexibility to programmers, also adds to the execution time overheads.

To look at the overheads in more details, let us take the *list* data structure supported in Python as an example. An attractive feature of a Python list is that the users can store different data types as different elements of the list. However, now dynamic type checking has to be applied to each element of the list. Moreover, since the list only stores pointers to the objects, rather than the actual objects themselves, the data is not stored continuously. As a result, data locality and cache usage is negatively impacted.

To quantify these overheads, we performed the following experiment. We implemented a linear algebra routine, *Double GEneral Matrix Multiplication* (DGEMM), in Python. We compared the execution time for Python program, executing using Python 2.4.3, with automatically translated C++ code (using Shedkin [35]), and a hand-written C code. In *Python* code, the *list* is used as the input data structure. *C++* code is generated from Python after data type inference is performed, and a user-defined vector-like container is used as the input data structure. The hand-written C code uses the primitive array as the input data structure.

It turned out that the calculation time of the pure Python ($484.46\ sec$) is around 8 times higher than the generated *C++* code ($59.56\ sec$). This is primarily because in the C++ code, all the type inference and type checking is performed at the compilation stage. However, the wrapper functions around this user-defined data structure still incur significant overheads. Thus, the hand-written C program ($11.96\ sec$) is 5 times faster than the generated C++ code, and overall, 40 times faster than the interpreted execution of Python.

In addition to the performance issues noted here, there is another challenge. For obtaining performance, it is increasingly becoming important to parallelize execution on multi-core and many-core architectures. Complex data structures pose significant challenges in parallelization. Moreover, the most common way of parallelizing computational steps is

to use existing libraries. These libraries, however, are based on flatter data structures, like multi-dimensional arrays. Thus, the use of nested and dynamic data structures can prohibit the use of these libraries, and the application cannot benefit from parallelization on multi-core or many-core architectures.

6.1.2 Overview of Our Translation Framework

We now give an overview of the approach we have developed in this work. As a motivating example, we use the Python code in **Figure 6.1**. The nested loop shown at the bottom of the Figure is similar to the computation performed in DGEMM example.

Before Linearization

```
#Data set structure definition
class A:
  def __init__ (self, a1):
      self.a1 = a1
      self.a2 = len (a1)
class B:
  def __init__ (self, b1):
      self.b1 = b1
      self.b2 = len (b1)
points = []
```

```
#Data set initialization
for i in range (t):
    b1 = []
    for j in range (n):
        a1 = []
        for k in range (m):
            a1.append (...)
        b1.append (A (a1))
    points.append (B (b1))
```

```
#Data access before linearization
for i in range (t):
    for j in range (n):
        for k in range (m):
            ... = points[i].b1[j].a1[k] ...
```

Figure 6.1: Python Code to Illustrate Translation Challenges

As stated earlier, we can significantly improve performance over interpreted execution of Python code by using existing tools for translating the code to C++. However, dynamic

data structures still impose a significant performance penalty, and disallow the use of existing libraries for multi-code and many-core systems. One approach for addressing this problem could be to copy the data to a *flatter* data structure, just before the execution of the main computational loop. This way, the main computation step may operate at an efficiency that is similar to that of the hand-written C code. Moreover, the arrays can be passed to the existing libraries that would allow parallel execution of the main loop.

While this idea seems simple, it still involves several challenges. First, flattening nested dynamic data structures may not be trivial, and we need a mechanism to perform the translation and for maintaining the correspondence between the two sets of data structures. Second, the copying step itself can be expensive, especially, if the procedure has to be repeated several times. Thus, we need mechanisms to avoid unnecessary copying of the data. Third, we can store data in arrays and operate on it only if the data in the dynamic data structure is *homogeneous*. We need an efficient mechanism to determine this.

We have developed techniques to address these three challenges, and have implemented them as part of our overall framework. This framework is shown in **Figure 6.2**. There are three main stages in our translation process. In the first stage, the Python code is translated into C/C++ code, using the existing tool, Shedskin. Particularly, this tool transforms high-level containers in Python to pre-defined container classes in C++ (similar to those used in a template library like STL). Type checking and type inferencing is performed during this step.

In the second stage, the generated C/C++ code is translated further with an emphasis on the main computational steps. This is the key novel contribution of this work , with algorithms for *Homogeneity Decision*, *Demand-Driven Inter-procedural Partial Redundancy*

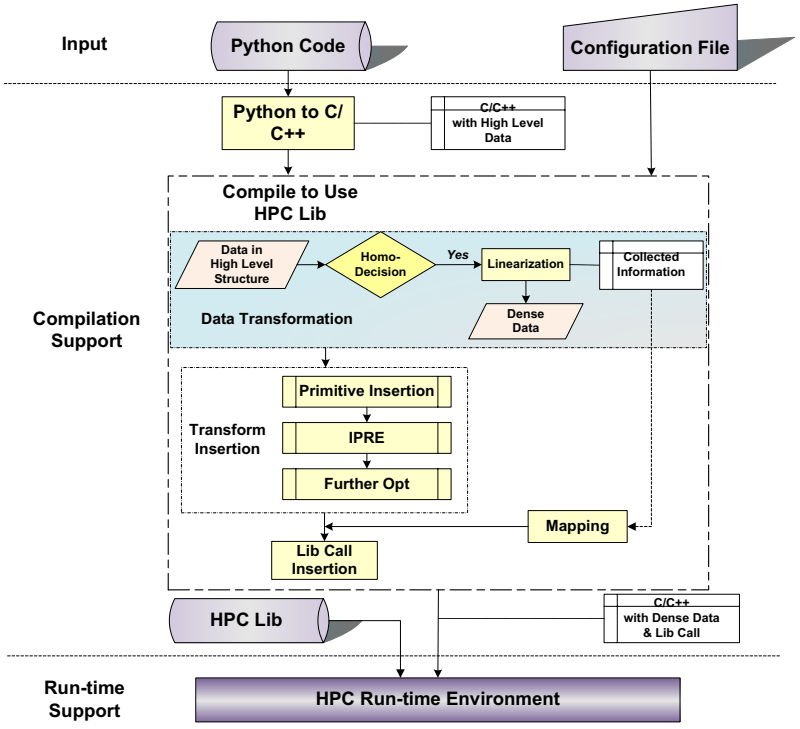


Figure 6.2: Overview of the Translation Framework

Elimination, and *Linearization* involved in this process. These methods are introduced in Sections 6.2 and 6.3, and have been implemented using the ROSE infrastructure [113].

In the last stage, the transformed C/C++ code with dense data structures is further analyzed to make appropriate library calls. This step is based on the existing work [89, 83], and the details are not described in this work.

6.2 Insertion Algorithm

The objective of the insertion algorithm is to reduce the overhead caused by the *linearization* operation, which is done by reducing the frequency of execution of this statement. Our overall approach can be viewed as a two-level one:

Level 1: Insert a dense data structure (A, such as *array*) just before any usage of the high level structure (L, such as *list*). We copy the actual objects in L to A and replace L by A. This work can be followed by an optional step, in which we reorder the members in the objects according to our computational requirement, which can improve the data locality and the efficiency especially for data-intensive applications.

Level 2: In order to avoid multiple (unnecessary) copy operations, a powerful redundancy elimination algorithm, *inter-procedural partial redundancy elimination* (IPRE), is designed.

Level 1 optimization simply requires an ability to *linearize* the data in the dynamic data structure. The method for this is presented in the next Section. We focus on the second level optimization in the next 2 subsections.

6.2.1 Intra-procedural PRE Algorithm

Our Level 2 optimization involves a novel use of an existing partial redundancy elimination (PRE) algorithm, and its extension into a *demand-driven* inter-procedural algorithm. We initially show why our problem is related to PRE.

Along a certain *control flow path*, if a computation is performed more than once without any modification to its operands between them, it will be considered as partially (or fully) redundant. Over the last 30+ years, several PRE algorithms [99, 30, 74] can be applied to optimize the code. Similarly, in our work, if a copy operation is performed more than

once along a certain path without any modification to the relative data elements, the copy operation can be treated as partially (or fully) redundant.

In order to explain the basic idea of the traditional PRE, **Figure 6.3** shows an intra-procedural example. In the left-hand-side of this figure, a *Control Flow Graph* is given, while the transformed code by PRE is introduced in the right-hand-side. In our work, the IPRE algorithm is derived from an existing intra-procedural algorithm that is summarized as following. This algorithm is chosen because of its conceptual simplicity.

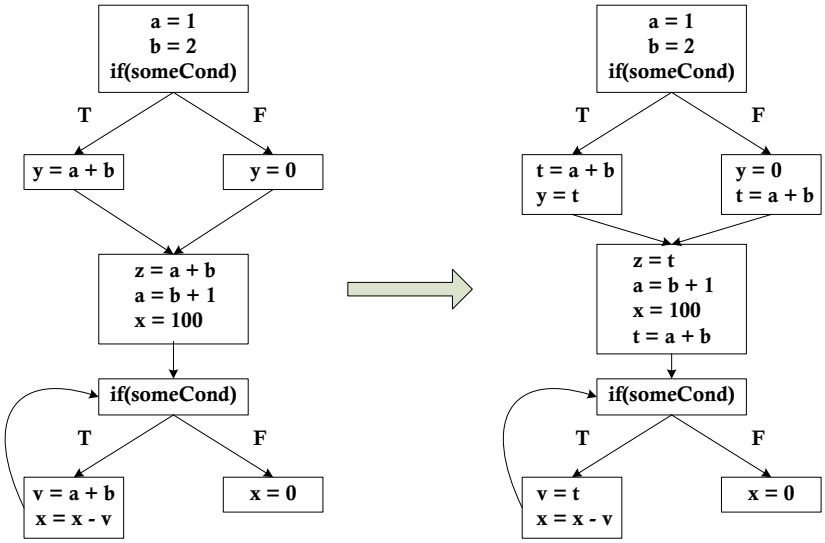


Figure 6.3: An Example to Illustrate Basic PRE: Before (left) and After (right)

An Existing PRE Algorithm

This part summarizes the main steps in the Partial Redundancy Elimination method developed by Paleri *et al.* [109]. While this algorithm uses most of the same ideas as the

original algorithm by Morel and Renvoise [99], as well as the subsequent algorithms by Dhamdhere [30] and Knoop *et al.* [74], it is conceptually simpler and has other properties, like the fact that it does not require any splitting of edges.

$$AVIN_i = \begin{cases} FALSE & \text{if i = s,} \\ \prod_{j \in pred(i)} AVOUT_j & \text{otherwise,} \end{cases} \tag{6.1}$$

$$AVOUT_i = COMP_i + AVIN_i \cdot TRANSP_i. \tag{6.2}$$

$$ANTOUT_i = \begin{cases} FALSE & \text{if i = e,} \\ \prod_{j \in succ(i)} ANTIN_j & \text{otherwise,} \end{cases} \tag{6.3}$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i \cdot TRANSP_i. \tag{6.4}$$

$$SAFEIN_i = AVIN_i + ANTIN_i, \tag{6.5}$$

$$SAFEOUT_i = AVOUT_i + ANTOUT_i. \tag{6.6}$$

$$SPAVIN_i = \begin{cases} FALSE & \text{if i = s or } \neg SAFEIN_i, \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise,} \end{cases} \tag{6.7}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i, \\ COMP_i + SPAVIN_i \cdot TRANSP_i & \text{otherwise.} \end{cases} \tag{6.8}$$

$$SPANTOUT_i = \begin{cases} FALSE & \text{if i = e or } \neg SAFEOUT_i, \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise,} \end{cases} \tag{6.9}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i, \\ ANTLOC_i + SPANTOUT_i \cdot TRANSP_i & \text{otherwise.} \end{cases} \tag{6.10}$$

$$INSERT_i = COMP_i \cdot SPANTOUT_i \cdot (\neg TRANSP_i + \neg SPAVIN_i), \tag{6.11}$$

$$INSERT_{(i,j)} = \neg SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j, \tag{6.12}$$

$$REPLACE_{i_f} = ANTLOC_i \cdot (SPAVIN_i + TRANSP_i \cdot SPANTOUT_i), \tag{6.13}$$

$$REPLACE_{i_l} = COMP_i \cdot (SPANTOUT_i + TRANSP_i \cdot SPAVIN_i), \tag{6.14}$$

Figure 6.4: Basic Intra-procedural PRE Data Flow Equations

The data-flow equations of this algorithm are shown in **Figure 6.4**, and the terms are explained in **Table 6.1**.

This algorithm can be divided into two phases: the *local phase* and the *global phase*. The local phase is applied to each *basic block* to reduce the redundancy within each *basic*

Table 6.1: Terms Used in the PRE Data Flow Equations

Symbols
$\cdot, \Pi$: Boolean conjunctions;
$+, \Sigma$: Boolean disjunctions;
$\neg$: Boolean negation.

Local properties
$TRANSP_i$: *transparent* In node i, if the operands of the expression are not modified;
$COMP_i$: *locally available* In node i, if there is at least one computation of the expression E, and *including* and after the *last computation*, there is no modification of the operands of E;
$ANTLOC_i$: *locally anticipable*. In node i, if there is at least one computation of the expression E, and before the *first computation*, there is no modification of the operands of E.

Global properties
$AVIN_i$/$AVIOUT_i$ The expression is available at the entry/exit of node i;
$ANTIN_i$/$ANTIOUT_i$ The expression is anticipable at the entry/exit of node i;
$SAFEIN_i$/$SAFEOUT_i$ The entry/exit of node i is safe. A point p is safe for some expression E, if we insert a computation of E at p without introducing any new value on any path through p;
$SPAVIN_i$/$SPAVOUT_i$ The expression is safe partial available at the entry/exit of i;
$SPANTIN_i$/$SPANTOUT_i$ The expression is safe partial anticipable at the entry/exit of i;
$INSERT_i$/$INSERT_{(i,j)}$ The computation of the expression should be placed before the *last computation* in node i; or on the edge between nodes i and j;
$REPLACE_{i_f}$/$REPLACE_{i_l}$ The replacement of the expression should be happened to the *first* / *last* computation in node i.

block. After it, only the *first* and the *last* computation of the expression in this block will be considered.

Focusing now on the global phase, from Equation 6.1, we can know an expression is available at the entry of a *basic block*, if it is available at the exit of all the predecessor blocks. An expression is available at the exit of a *basic block*, if it is locally available or available at the entry of the current *basic block* without any operands modification in it (Equation 6.2). Similarly, from Equation 6.3 and Equation 6.4, we can know an expression is anticipable at the exit of a *basic block* if it is anticipable at the entry of all the successor blocks, while an expression is anticipable at the entry of a *basic block* if it is locally anticipable or anticipable at the exit of the current *basic block* without any operands modification in it.

The most interesting part of this algorithm is that it focuses on the *safe* points (SAFEIN and SAFEOUT), the points where we can insert the computation of some expression without introducing a new value along any path. The final insertion points and replace points are decided by Equations 6.11 to 6.14 based on the operators and terms in **Table 6.1**.

6.2.2 Inter-procedural PRE algorithm

For even a modest-sized application, the overheads of linearization cannot be reduced without applying PRE inter-procedurally. Though there have been a couple of efforts on developing an inter-procedural PRE algorithm [2, 75], we have developed a *demand-driven* inter-procedural algorithm, which analyzes procedures only if it is needed for placement of the linearization operations. In our applications, the key data structures are not modified

frequently, so normally, there should be only a few linearization operations placement involved. Thus, our demand-driven algorithm results in analysis of only a small number of procedures from the application.

```
void main (){
    List points;
    Initial_points (points);
    kmeans_reduction (points);
}
void kmeans_reduction (List points){
    List clusters;
    Initial_clusters (clusters, points);
    for (i = 0; i < iterations; i++){
        kmeans (points, clusters);
        update_clusters (clusters);
    }
}
void kmeans (List points, List clusters){
    for (point p in points){
        //min_cluster is the closest centroid
        min_cluster.min_distance = max (double);
        min_cluster.min_position = 1;
        for (cluster c in clusters){
          min_cluster = find_closest_centroid (p, c);
        }
        update_reduction_object (min_cluster);
    }
}
void update_clusters (List clusters){
    for (cluster c in clusters){
        //update the centroid by pre-defined reduction object
        c = ...reduction_object ...;
    }
}
```

Figure 6.5: The C-like Pseudo-code for K-means Application

Our algorithm is based on the *inter-procedural control-flow graph* (ICFG), which has been widely used for inter-procedural analysis. This ICFG contains the control flow graphs

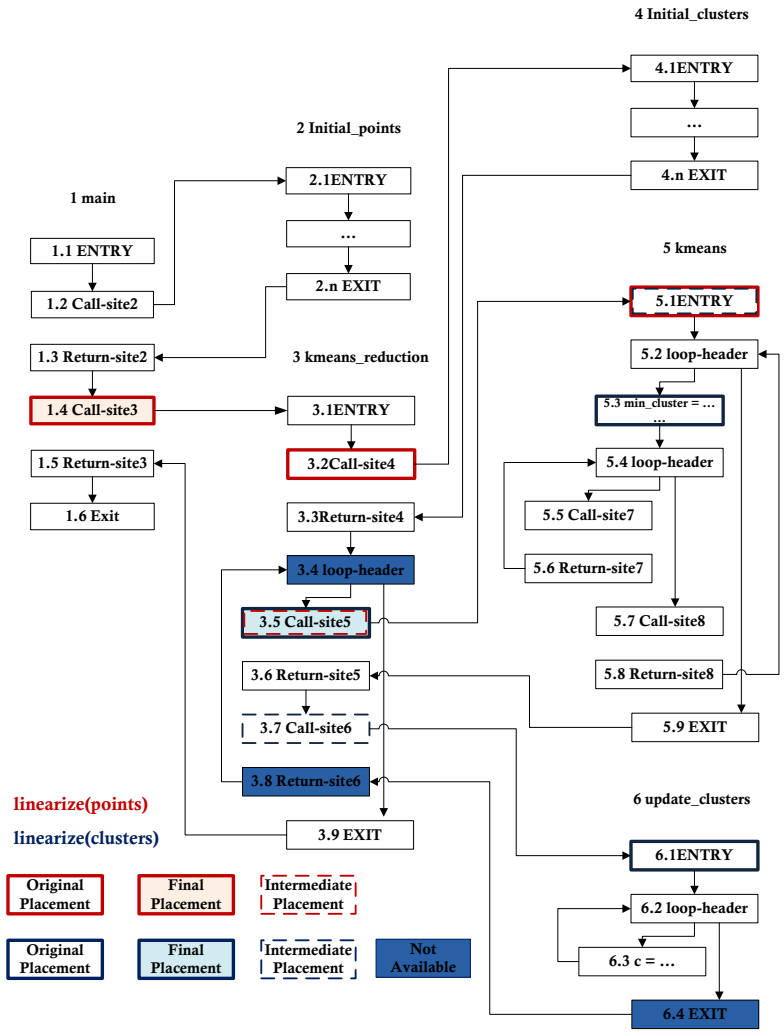


Figure 6.6: The ICFG for K-means Application

(CFG) for the individual procedures. For each procedure p, an entry node $Entry_p$ and an exit node $Exit_p$ are defined. Each *call-site* to p is represented by two nodes: $Call - site_p$ and $Return - site_p$. If a basic block contains a $Call - site_p$, it will be split into two basic nodes $B1$ and $B2$. There is an edge from $B1$ to the entry node of the procedure p, ($B1$, $Entry_p$), and similarly, there is an edge from the exit node of the procedure p to $B2$,

($Exit_p$, $B2$). In **Figure 6.6**, we show ICFG for the K-means application listed earlier in **Figure 6.5**.

Algorithm 7 AnalyzeAll ($procedure\_set$, $linearize\_set$)

```
1:  for each linearization expression linearize(l_i) ∈ linearize_set do
2:     for each procedure p_j ∈ procedure_set do
3:        intra-procedural analysis on linearize(l_i) in p_j
          without considering the effect of call-sites
4:     end for
5:     pick-up p in which l_i is first define
6:     p_parent = p_current = p
7:     if l_i ∈ global_variables then
8:        p_parent = p_current = main
9:     end if
10:    Analyze(p_current, p_parent)
11:    for each procedure p_j ∈ procedure_set do
12:       final insertion and deletion
13:    end for
14: end for
```

Our IPRE method is shown through Algorithms 7 and 8. In our inter-procedural framework, we assume that inter-procedural pointer-analysis [53] and alias-analysis [52] have been performed in the preprocessing stage and all the variables that point to the same space are labeled with the same name.

In order to explain our algorithm, we use K-means example. First, an initial placement of the linearization operation is performed. After this stage, in **Figure 6.6**, the linearization operations for the list $points$ are placed at the beginning of the node 3.2 and the end of the node 5.1. Similarly, linearization operations for the list $clusters$ are placed at the end of the node 5.3 and at the end of the node 6.1.

In next stage, we use the method presented through Algorithm 7 and 8. Initially, intra-procedural analysis is performed in the procedure(s) where the initial placement has been

Algorithm 8 Analyze ($p_{current}$, p_{parent})

```
1: for each node node_i in p_current do
2:   if node_i includes procedure p then
3:     p_parent = p_current
4:     p_current = p
5:     Analyze (p_current, p_parent)
6:   else if node_i = EXIT_{p_current} then
7:     if p_current = p_parent then
8:       return {*arrive at the outer-most procedure*}
9:     else if p_current is completely transparent with l_i then
10:      return {*nothing is affected*}
11:    else if p_current includes modification to l_i then
12:      mark the availability of AVIN_{RETURN-SITE_{p_current}} in p_parent according to the value
         of
         AVOUT_{EXIT_{p_current}}
13:      if linearize(l_i) is safe at the ENTRY_{p_current} then
14:        mark CALL - SITE_{p_current} as
           ANTIN/ANTOUT and COMP
           mark ENTRY_{p_current} as AVIN
15:      end if
16:      propagate effect by work-list in p_parent and p_current
17:      return
18:    end if
19:  end if
20: end for
```

done, without considering the effect of the functions calls. During this phase, we apply only Equations 6.1 through 6.10 in **Figure 6.4**, i.e. insertion or deletion logic is not computed.

Next, we move to the inter-procedural phase. If the list parameter is anticipable at the entry of current procedure, we will pull this linearization operation out of p, and try to propagate it further. For example, by this *pull out* strategy, the linearization operation *linearize(points)* can be pulled from the procedure *kmeans* to *kmeans_reduction*, and until the *main* function. Finally, we will mark the node 1.4 in **Figure 6.6** as $COMP$ and $ANTOUT$ (stronger than $SPANTOUT$) by the line of 14 of the Algorithm 8. From the intra-procedural analysis, we have know that the node 1.4 has already been marked as $\neg SPAVIN$. Based on all of these, we can know that the final insertion for *linearize(points)* can happen at the beginning of the node 1.4. All others placements will be deleted since we have already marked them as $AVIN$ (according to the line 15 of the Algorithm 8 and the propagation operation), and there are no further modifications to $points$.

For a procedure call p from the current procedure $p_{current}$, we consider two possibilities. First, if p is *completely transparent* relative to the parameter list of the copy statement, no further analysis is done on p. Second, if p is not transparent, we just need to copy the availability from the Exit point of p. For example, in **Figure 6.6**, in the Exit node of the procedure *update_clusters*, *linearize(clusters)* is not available, so in the node 3.8, it is also not available, which will cause *linearize(clusters)* is $\neg AVIN$ in the node 3.5. Subsequently, we can infer that the final insertion for *linearize(clusters)* will happen at the beginning of the node 3.5 and other placements will be eliminated.

K-means application is an iteration process: before the reduction loop, the input data set *points* is initialized without any further modification during the whole process, and the output data set *clusters* is updated in each loop. From **Figure 6.6**, it is easy to know that

after our elimination, the final placement of *linearize(points)* is out of the reduction loop, and the placement of *linearize(clusters)* is within the reduction loop, which is coherent to the basic logic as above.

6.2.3 Checking Homogeneity of a List

One of the requirements for converting a dynamic data structure to an array is that each element of the original dataset is of the same type. In this section, we describe an algorithm we have developed for this purpose.

Our description here assumes a list structure, though the algorithm can easily be applied to any other dynamic data structure. This decision algorithm is modeled as a *data flow analysis* problem, similar to the well-known *Constant Propagation* problem, for which many algorithms have been developed [12, 135]. Here, only the intra-procedural version is introduced, and the inter-procedural version can be developed easily similar to how we developed the IPRE algorithm above.

The entire algorithm can be expressed as a four-tuple $< G, D, L', F >$, where, $G = (N, E)$ is a control flow graph. D is the direction of the data flow, which is $FORWARDS$ here. L' is a three-tuple $< V', \wedge, m >$, in which there are three elements: V', $\wedge$ and m. V' is the domain of values, and each element in it is in this form: $(type_1, type_2, \ldots)$, i.e., a cross-product of the basic lattice L_i, in which, *UNDEF* is the top element $\top$, and *NON-HOM* is the bottom element $\bot$. In L', $\wedge$ is a meet operator, which follows the common definition of $\wedge$ for the product lattice, i.e, it is defined as:

$$(type_1, type_2, \ldots) \wedge (type'_1, type'_2, \ldots)$$

$$= (type_1 \wedge type'_1, type_2 \wedge type'_2, \ldots)$$

m is a map function used to map the list definition to the lattice. For example, the result of $m(list_i)$ will be $type_i$. Thus, each element in V' can also be expressed in this form $(m(list_1), m(list_2), \ldots)$. There is a special map function, m_0, which can initialize the type of the list variable into *UNDEF*.

Returning to the last element of the four-tuple, $F : V' \rightarrow V'$ is the domain for transfer functions. It has an identity function f_i, such that $f_i(x) = x$ for all x in V'. Like any standard intra-procedural data flow algorithm, we can consider two levels: within *basic block*, and inter *basic block*. Based upon this, we can classify the elements in F into two groups: working on statements within a *basic block* and working on *basic blocks*. The rules for the former case, denoted as f_s, can be defined as follows:

1. If the statement (s) is irrelevant to the given list, f_s is the identity function f_i;

2. If s is relevant to the given list, $list_a$, then for any $list_i \neq list_a$, $f_s(m(list_i)) = m(list_i)$, and for $list_a$, $m'(list_a) = f_s(m(list_a))$.

We further consider the following cases:

1. if s is a member function call without adding any new elements in the list, such as $list_a.remove()$, $m'(list_a) = m(list_a)$;

2. if s is a member function call adding an element, such as $list_a.append(x)$, or $list_a.insert(k, x)$, $m'(list_a) = m(list_a) \wedge typeof(x)$;

3. if s is a concatenation operation, and more than 1 element, i.e., $[x_1, x_2, \ldots]$ are added to the $list_a$, there are two cases: i) if the type of the new list is already calculated as $type'$, then $m'(list_a) = m(list_a) \wedge type'$, and ii) if this is not the case, examine

Table 6.2: Homogeneity Decision Expression (Global Level)

	Non-Hom decision Expression (Global Level)
Domain	Sets of Basic Blocks
Direction	Forward
Transfer Function	$f_B = x \wedge LOC[B]$
Boundary	$OUT[ENTRY] =$
Meet($\wedge$)	Defined as above
Equations	$IN[B] = \wedge_{P,pred(B)} OUT[P]$; $OUT[B] = f_B(IN[B])$
Initialization	$OUT[B] =$

the type of new elements by $type' = typeof(x_1) \wedge typeof(x_2) \wedge \ldots$, and then $m'(list_a) = m(list_a) \wedge type'$.

After applying f_s to all statements in each *basic block*, we can get the result for each block at the point of $OUT[B]$. We refer to them as $LOC[B]$. The rule for the transfer functions working on each basic block, denoted as f_B, are summarized in the **Table 6.2**.

6.3 Linearization and Mapping Algorithm

The methods presented in the previous section have explained under what conditions contents of a dynamic data structure can be copied into a dense and flat data structure, and where the copy operation can be placed. In this section, we introduce *linearization* and *mapping* algorithms, with the goal of translating the high-level dynamic data structures in Python to low-level dense memory buffer in C++. This, in turn, will allow use of HPC libraries for multi-core and many-core architectures.

Specifically, we need to create a low-level continuous data storage (D_s) from the high-level data view (D_v). The entire process can be formally viewed as of computing the following two functions: 1) $F_t \subseteq \{f \mid f : D_v \rightarrow D_s\}$, a *linearization* function, which

can transform the high level data view to the low level data storage, and 2) $M \subseteq \{m \mid m : D_v \rightarrow D_s\}$, a *mapping* function created to enable mapping of the computations to the low-level data layout.

6.3.1 Linearization

Algorithm 9 ComputeLinearizeSize(Xs)

```
1:  size = 0
2:  if Xs.type = isPrimitive then
3:      size = sizeof(Xs)
4:  else if Xs.type = isIterative then
5:      for x in Xs do
6:          size += ComputeLinearizeSize(x)
7:      end for
8:  else if Xs.type = isStructureType then
9:      for each member m in Xs do
10:         size += ComputeLinearizeSize(m)
11:     end for
12:     ...
13: end if
14: ...
15: return size
```

The basic *linearization* is a two-step algorithm which includes two functions: *ComputeLinearizeSize*, shown as Algorithm 9 and *LinearizeIt*, shown as Algorithm 10. The first function is used to compute the data size while the second one is used to copy the actual data to the continuous memory space.

Let us revisit the code from **Figure 6.1**, where a very common example of using the user defined input data structure in Python was shown. **Figure 6.7** shows information that needs to be collected during the linearization process to enable code generation for the usage of the linearized data structure.

Algorithm 10 LinearizeIt(Xs, $size$)

```
1: ▷ allocate memory with the size of size
2: if Xs.type = isPrimitive then
3:     copy(Xs)
4: else if Xs.type = isIterative then
5:     for x in Xs do
6:         LinearizeIt(x)
7:     end for
8: else if Xs.type = isStructureType then
9:     for each member m in Xs do
10:        LinearizeIt(m)
11:    end for
12:    ...
13: end if
14: ...
15: return addressOfLinearizeData
```

6.3.2 Mapping

Algorithm 11 ComputeIndex($unitSize[]$, $unitOffset[][]$, $myIndex[]$, $position[][]$, i, $levels$)

```
1: ▷ During the linearization phase, collecting necessary information
2: if i < levels − 1 then
3:     index = unitSize[i] × myIndex[i] + unitOffset[i][position[i][]]
4:     index += ComputeIndex(unitSize[], unitOffset[][], myIndex[], position[][], i++,
       levels)
5: else
6:     index = unitSize[i] × myIndex[i]
7: end if
8: return index
```

The *mapping* algorithm can be divided into two stages: in the first stage, collecting the necessary information during the *linearization* process; in the second stage, computing the projected index of the low level data storage D_s from the collected information and the original index in D_v by the recursive strategy in algorithm 11. The parameters used by this algorithm are summarized in **Table 6.3**.

Table 6.3: Descriptions of the Parameters in Mapping Algorithm

Collected During Linearization
$unitSize[]$ 1-Dimensional Array. It stores the unit size of the elements in each level with unitSize[$levels - 1$] storing the inner-most elements.
$unitOffset[][]$ 2-Dimensional Array. It stores the offsets of the variables at each level. The first dimension is used to indicate the level and the second one indicates the start positions of the variables at current level.
$position[][]$ 2-Dimensional Array. It provides the position information for calculating the $unitOffset$.
$levels$ The total number of levels of the data.

Collected From D_v
$myIndex[]$ 1-Dimensional Array. It records the index for each level.
i An indicator to show the current level. Normally, it starts from 0 indicating that the current level is the outer-most.

Information Collected During Linearlization
levels = 3;
unitSize[levels] = {unitSize_B, unitSize_A, sizeof(data_type_a1)};
unitOffset[levels-1][2] = {{unitOffset_B[]}, {unitOffset_A[]}};
unitOffset_B[2] = {0, $unitSize\_A \times n$ }
unitOffset_A[2] = {0, $sizeof(data\_type\_a1) \times m$}
position[levels-1][2] = {{0, 1}, {0, 1}};
{*This should be collected in the accumulate function*}
myIndex[levels] = {i, j, k};

Data access after linearization
```
for(i = 0; i <t; i++){
    for(j = 0; j <n; j++){
        for(k = 0; k <m; k++){
            index = computeIndex(unitSize, unitOffset,
                    myIndex, position, 0, levels);
            ... = linea_points[index] ...
        }
    }
}
```

Figure 6.7: The Example of Using Linearization and Mapping Functions

Figure 6.7 shows the information that should be collected to apply the mapping algorithm. Most of the information should be collected during the *linearization* stage, while the index information is obtained from the usage loop. The entire mapping process is recursive. It starts from the outer-most level and terminates with the inner-most level. At each level, we calculate the offset caused by the index and the position offset.

6.4 Implementation and Experiments

In this section, we describe a prototype implementation of our framework and evaluate it by generating code for execution of *data-intensive* applications on a multi-core system, and *computation-intensive* applications on a GPU.

6.4.1 Implementation Overview

Python code was translated to C++ using an existing tool, Shedskin [35]. Our transformations were implemented on top of the ROSE compiler infrastructure [113]. ROSE is a powerful tool that supports program analysis and source-to-source transformations for C/C++, FORTRAN, and other languages. After the transformations are applied, low-level HPC libraries are invoked to support mapping on the multi-core and many-core libraries. Particularly, we used a data mining middle-ware for mapping data-intensive applications to multi-core architectures, and used existing libraries to execute linear algebra operations on GPUs [90]. All these libraries/middle-ware expect the data to be in multi-dimensional arrays, and cannot support processing of nested or dynamic data structures. The code generation was based on our earlier work, and the details are not presented here.

6.4.2 Evaluation Goals and Platforms

The objective of our evaluation is to compare the execution time of the original Python code (*Python*), Shedskin generated C/C++ code (*Gen C++*), transformed code with and without IPRE optimization (*WOPRE* and *WPRE*, respectively), and the hand-written C/C++ with library functions (*Manual*).

Our experiments are conducted on the following platforms. A multi-core machine with AMD Opteron(tm) Processor (2.6 GHZ frequency) and main memory size of 32 GB was used for data-intensive applications. The GPU used for compute-intensive applications was a Quadro FX 5800 GPU, with 240 cores and 4 GB memory.

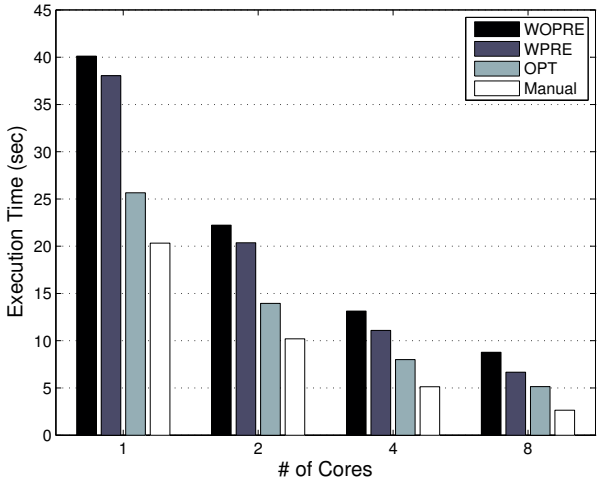


Figure 6.8: K-means: Comparison of Performance of Different Versions (800 MB dataset, $k = 100$, $iter = 1$)

6.4.3 Experiments with Data-Intensive Applications

We invoked a data-intensive computing library from transformed C++ code, and compare the performance of different versions we listed earlier. We used two popular data mining applications, which are K-means clustering and PCA.

An 800 MB representative dataset was used for K-means. In our experiments, we control the computation workload by modifying the iteration numbers. Very similar to the DGEMM example in Section 6.1, the calculation time of the Python code, which uses a list as the main input data structure, is much longer than the generated code and the transformed code. For example, even to a much smaller data set (8 MB), the calculation time of the *Python* code interpreted by *Python 2.4* is 109.60 seconds for 1 iteration and 1122.96 seconds for 10 iterations. For the data set of 800MB, the execution time of the *Gen C++* code is 59.28 seconds for 1 iteration and 593.06 seconds for 10 iterations.

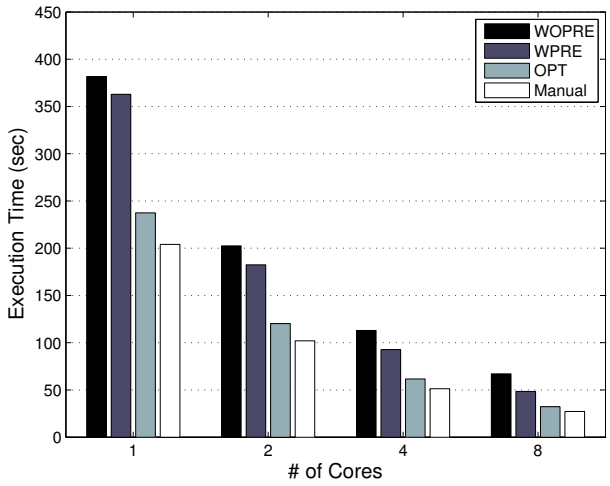


Figure 6.9: K-means: Comparison of Performance of Different Versions (800 MB dataset, $k = 100$, $iter = 10$)

In **Figure 6.8** and **Figure 6.9**, we report the calculation time of the code transformed by our framework. From **Figure 6.8**, we can see that comparing to the *Gen C++* code, the efficiency of the sequential version of our transformed code for 1 iteration is improved by more than 30% even including the linearization overhead of the input data set (*WOPRE* version). Comparing with the *WPRE* version, we found that IPRE can help overcome nearly 50% of the linearization overhead, which is consistent to our analysis in Section 6.2. In K-means, because the centroid set is a frequently accessed data structure, we can also linearize and apply the IPRE on it as described in Section 6.2, resulting in a version we refer to as *OPT*. By comparing the versions *OPT* and *Manual*, we can see that their performance is very similar to each other, and the overhead caused by linearization and mapping is within 30% for 1 iteration. On the other hand, by comparing the sequential versions of *OPT* and *Gen C++*, we can see that by our optimization framework, the efficiency of the

compiled code can be improved by a factor of more than 2 for the sequential version, and furthermore, we have enabled use of a parallel library.

From the comparison of **Figure 6.8** and **Figure 6.9**, we can see that the linearization overhead can be reduced to a large extent by our IPRE method when there are multiple iterations. Finally, for the sequential version, the overhead of the *OPT* version is around 10% of the best version, which is mainly caused by the *mapping* operations and scalable to the number of processors. That is why we see good scalability of the *OPT* version. The relative impact of our optimizations is even more significant for parallel versions, since linearization is performed sequentially.

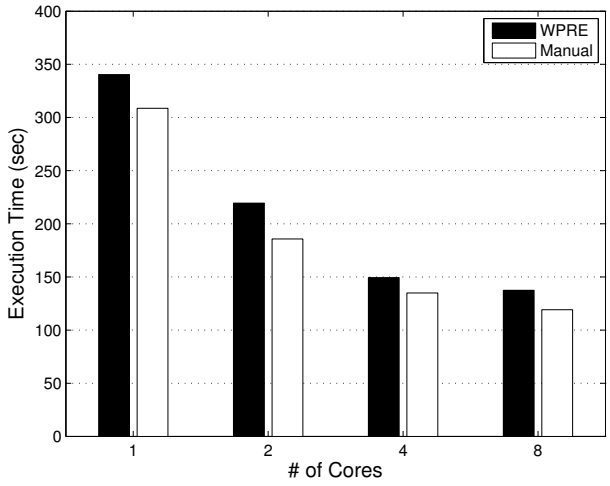


Figure 6.10: PCA: Comparison of Performance of Different Versions ($row = 1000$, $column = 100,000$)

The datasets used for PCA experiments has 1000 rows and 100,000 columns. The calculation time of the *Python* code is very long, for example, even to a much smaller data set (1000×1000) it takes 634.45 seconds. The *Gen C++* code is also relatively slow, for

example, to the data set of $1000 \times 100,000$ it takes 3280 seconds. By using our framework, the efficiency can be improved obviously, however, the IPRE optimization must be applied to the linearization of the input data set. Without the IPRE algorithm, the linearization is inserted in the inner-most loops, resulting in $\Theta(row^2)$ times copy operations to the input data set that is a very large overhead. In **Figure 6.10**, we compare the calculation time of the *WPRE* version generated by our framework and *Manual* versions. As stated above, *WOPRE* version data is not shown, because it is extremely slow. From the comparison, we can see that the efficiency of the *WPRE* version is very similar to the *Manual* version, and the overhead caused by the linearization and mapping operation is around 10% to 20% for both sequential and parallel versions. Especially, the efficency of the sequential *WPRE* version is improved by a factor of 10 comparing to the *Gen C++* code.

6.4.4 Scaling Compute-Intensive Applications with a GPU

GPU has been gaining popularity in recent years because of their very favorable performance to cost ratio. Many GPU related libraries and automatic code generators have been developed in recent years. In our experiments, CUBLAS libraries [104] and tensor contractions [90] generated code are used for accelerating the execution of two linear algebra kernels written in Python.

The first linear algebra kernel is DGEMM. The implementation from the CUBLAS library can be invoked to replace the sequential computations in the Python implementation. Because the mapping function is not needed in this case, the mapping overhead is not considered in this and the next example. We experimented with seven datasets, which range from 1000×1000 to 7000×7000. The results of the experiment are shown in **Figure 6.11**. By comparing the results on the 1000×1000 dataset with the example in

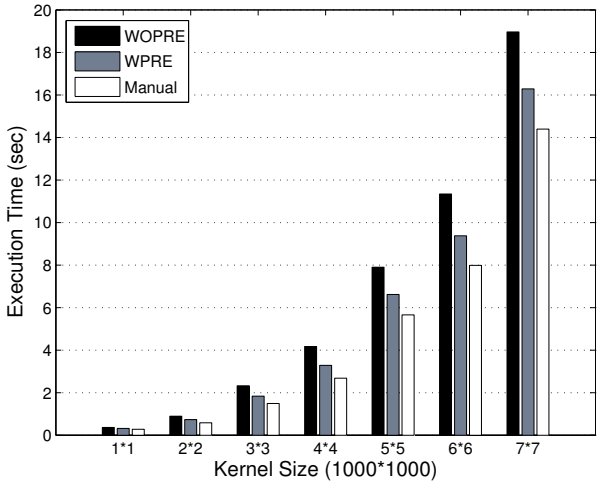


Figure 6.11: Experiment Results for DGEMM

Section 6.1, we can see that the performance of the CUBLAS version is much better than the *Python* code (more than 1000 times speedup) and *Gen C++* code (around 163 times speedup), even before the optimizations are applied.

In this application, the IPRE optimization is mainly used to eliminate the linearization overhead during the matrix dimension validation. Since there is no modification between this stage and the main loop computation, there is no need to linearize the input matrices multiple times. This is also applied to the next experimental case. From the comparison between the versions of *WPRE* and *WOPRE* in **Figure 6.11**, we can see that the linearization overhead can be reduced by more than 50% by using IPRE. And also the linearization overhead becomes less significant with the increase in the data set sizes, and when the size of the kernel is 7000×7000, the linearization overhead is reduced to be less than 15% with our IPRE method comparing to the best *Manual* version.

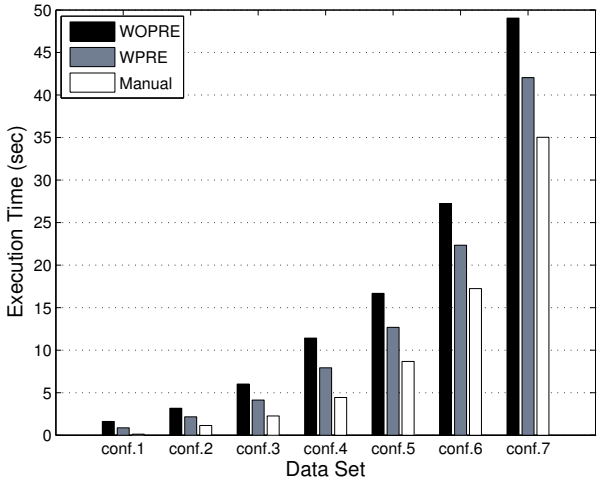


Figure 6.12: Experiment Results for Tensor Multiplication

The second linear algebra kernel is tensor contraction, which is a multi-dimensional matrix multiplication. In pure Python, if we want to perform such a computation, a highly nested list structure needs to be used, which decreases the performance severely. Thus, our transformations are even more crucial. The following expression was used in our work:

$result[h_3, h_1, h_2, p_5, p_4, p_6] += x[h_7, p_4, p_5, h_1] \times y[h_3, h_2, p_6, h_7]$

Figure 6.12 illustrates the execution time for different datasets of increasing sizes. Both *WOPRE* and *WPRE* are the versions generated by our framework and the *Manual* one is the version written manually to feed into the Code Generator. Again, the *Python* and *Gen C++* codes are very slow, i.e. even for *config1* they run for 261.39 seconds, 16.85 seconds, respectively. Details of both these versions are not shown here. The effect of the IPRE optimization is shown in this experiment by comparing the versions of *WOPRE* and *WPRE* that the linearization overhead is decreased by around 50%. From the comparison between the transformed code (*WOPRE* and *WPRE*) and the *Manual* version in **Figure 6.12**, we

can see that the linearization overhead is very large when the data set is relatively small. However, the overhead reduces with increasing in dataset sizes. For instance, in *config7*, comparing the versions of *WPRE* and *Manual*, the linearization overhead is already smaller than 20% with IPRE.

6.5 Related Work

Given the popularity of Python, there have been several efforts focusing on improving Python's efficiency. These efforts can be classified into two groups, which are adding extension libraries or constructs to *Pure* Python, and compiling Python to other languages, such as C/C++ or even CUDA. NumPy and SciPy [107, 66] are examples of the former, where the inefficiency caused by the dynamic data structure usage in pure Python is substantially reduced by adding an N-dimensional array object. For multi-processing, these efforts have been integrated with PyMPI [97]. PyCUDA and PyOpenCL [73] are two library extensions where GPU code can be invoked from Python. More recently, Catanzaro *et al.* [15] developed a data parallel language named Copperhead which is based on Python. Compared to the above efforts, our goal is clearly different, in the sense that we start with pure Python, and automatically replace dynamic data structures with arrays.

In efforts that compile Python to other languages, prominent ones include Cython [6] and Pyrex [37], where type-annotated Python is compiled to C, and Shedskin [35], where a subset of Python is compiled to C++. To use multi-core or many-core system, Garg *et al.* [41] developed a framework to compile Python code to a hybrid CPU-GPU environment. The initial application is assumed to use array-based constructs in their work.

6.6 Summary

In order to bridge the gap between the productivity and the performance in HPC applications, this work has presented a framework to compile pure Python to invoke existing multi-core and many-core libraries. To enable such optimizations, a demand-driven interprocedural algorithm has been developed. We have also developed a novel *Homogeneity Checking* algorithm, and a set of *Linearization-Mapping* schemes. By these algorithms, dynamic data constructs in Python can be transformed into dense memory buffer that can be accepted by the low level libraries.

Two data-intensive and two linear algebra applications were used to evaluate our framework. The evaluation results show that the code generated by our framework is only 10% to 20% slower than the hand-written C code that invokes the same libraries. IPRE optimization we perform turns out to be significant for improving performance in most cases. Moreover, the code generated by our framework outperforms interpreted Python and the C++ code generated by an existing tool by one to two orders of magnitude.

Chapter 7: Future Work

Most of our former efforts emphasize on either fine-grained parallelism on SIMD architectures, or coarse-grained parallelism on multi-core architectures, respectively. The emerging Xeon Phi architecture provides us a good opportunity of putting them together, *i.e.*, exploring *hierarchical parallelism* on multi-core system with SIMD accelerators. We have done some preliminary work on this topic, *i.e.*, parallelizing two irregular applications, *Molecular Dynamic*, and *Euler* on Xeon Phi hierarchically, however, we still have some open issues related to resolving the memory access latency.

7.1 Improving Memory Performance for Hierarchical Parallelism

The *memory wall problem* is obvious even for fine-grained SIMD parallelism as we discussed in our existing work, and the combination of fine-grained and coarse-grained parallelism will make the situation even worse due to the limited memory bandwidth and last level cache sharing among different cores. It is impossible for one to benefit from any level parallelism, if the program is memory bounded, so it can be foreseen that false design without considering the memory impact will result in program with undesirable performance and resources wasting. This issue provides us an open question not even evaluated for regular applications.

Traditionally, there are two candidate solutions for this problem, *software prefetching* and *locality optimizations*. Our existing work emphasizes on the latter, while the *hierarchical parallelism* offers us a good chance of applying the former to reduce the memory latency. However, because of the SIMD processing of each core, the data demand and data access pattern will be much different from traditional cases, and some new challenges may be arising during this process.

7.1.1 Potential Future Research

This research topic exposes us multiple interesting problems that we can explore in the future.

Prefetching Objectives Identification and Prefetching Slices Creation

Data Flow Analysis is helpful for us to identify the irregular data structures that should be prefetched. Based on this kind of analysis, the work [11] provides us an intra-thread prefetching method for irregular data structures, *greedy prefetching*. This kind of inter-procedural data flow analysis is certainly capable to be applied in our work to identify the prefetching objectives. However, we assume that in our specific applications, the number of prefetching objectives is limited, and these objectives are a sort of underlying knowledge of programmer, so this analysis procedure can be reduced to a light-weight one.

There are many different ways of generating the helper threads. The work [22] uses hand-built helper threads. The work [71] uses compiler techniques to generate the reduced version statically with the help of profiling. The works [85, 142] generate helper threads in dynamic run-time compilation environment. In our work, since the prefetching objective is specific, a hand-built creation procedure is capable for us.

The Thread Order Guarantee and Cache Pollution Minimization

Properly optimization on the *pre-computation thread* (p-thread, and the pre-computation code is called p-slice) may guarantee that the prefetching thread is always ahead of the main thread. The work [142] discussed on a series of optimization methods from hardware to software, such as dynamic hardware load stride prediction to speculatively specialize p-slices, allowing for simpler p-slices with lower overhead, dynamically identification of the loop induction variables, allowing to jump start the p-slice execution a few iterations ahead of the main thread, reducing the control flow dependency to speed up the p-slice, continuously monitor the success of prefetching and so on.

Cache pollution minimizing for prefetching algorithm is solved as the topic *prefetching control policy of helper thread* [68]. It is another side of the problem above, i.e. how to guarantee that the helper thread does not prefetch the data too early.

This problem is an important future work for my dissertation, since SIMD units application is going to bring us new research issues when we are applying the traditional approach above.

Combination of Helper Thread Prefetching and Data Layout Optimization

Generally, helper thread prefetching reduces the memory latency however increases the memory bandwidth usage; while data locality optimization reduces both average memory latency and bandwidth usage by using computation reordering or data layout reordering [3, 4]. So for high-bandwidth memory hierarchy (GPU, and MIC), software prefetching works better; and for low-bandwidth memory hierarchy (traditional CPU), data locality optimization works better. This leaves us a question whether it is possible to combine the

benefits of both the helper thread prefetching and data layout optimizations in our hierarchical parallelism for irregular data structures, *i.e.*, we reorder the irregular data structures to explore both inter-thread and intra-thread data locality as our earlier work, and next design a proper helper thread prefetcher to overlap the computation with the memory access.

There are two detailed research topics here: first, exploring the way to allocate the resources properly as we talked in previous section, *i.e.*, developing a scheduling strategy to decide the number of cores as *computation worker*, and the number of cores as *prefetchers*; second, comparing the implementations on both MIC architecture and GPUs architecture, to study the difference between them or to design a portable analytical model as our earlier work.

Chapter 8: Conclusions

This is a summary of the contributions of my dissertation and future work.

8.1 Contributions

Our main objective is to automatically utilize fine-grained and coarse-grained parallelism methodologies to improve the performance of three classes of applications involving dynamic data structures and irregular memory access on various SIMD accelerators and many-core architectures, an important topic nowadays in both high performance computing and compiler communities. Our contributions can be summarized as following:

- We identify the opportunity of exploiting fine-grained data parallelism in important, latency critical irregular algorithms, like *trees and graphs traversals* widely used in production level software, and design a novel intermediate language based approach to implementing such kind of parallelism. Our approach is evaluated by two real random forest applications and one regular expression engine, resulting in good single-core speedup.

- To improve the memory performance of fine-grained SIMD parallelism of irregular data structure traversals as above, we describe three novel data layout optimizations that are designed to extract intra and/or inter thread data locality from applications

that traverse a large number of irregular data structures on SIMD hardware, and propose an analytic model that can remove the burden of *performance portability* from the programmer side by accurately modeling which of our data layout optimizations to use on a particular architecture.

- We design and implement a programming system to parallelize applications with *irregular reduction* communication pattern on emerging Intel Xeon Phi coprocessors with emphasizing on two levels of parallelism, shared memory MIMD, and SIMD vectorization. This programming system incorporates a data reorder scheme to reduce the partition overhead, increase the data locality, and support better SIMD vectorization. Our system is tested on two irregular applications, Euler and Molecular Dynamics, and shows good performance.

- Intel Xeon Phi coprocessor and CPU host have separate memory hierarchies, so data transfer between CPU and coprocessors is an important issue in coprocessor programming model. To address this problem, especially for *dynamic allocated multi-dimension arrays and multi-level pointer data structures*, we design and implement an optimized compile-time and runtime integrated framework to manage such kind of data transfer automatically. Our framework is tested on 7 representative benchmarks, and shows good performance.

- We present a Python based compilation system that invokes libraries for multi-core and many-core architectures for specific types of computations, in which, the *dynamic data structures* of Python, like *List* are transformed into dense memory buffer. To enable such optimizations, we have developed a demand-driven inter-procedural

PRE algorithm, and a novel *Homogeneity Checking* algorithm to reduce the layout optimization overhead.

To improve the performance of applications involving dynamic data structures, and irregular memory access, our existing optimization strategies focus on improving data locality, and better utilizing the cache hierarchy, while in emerging many-core architectures like Intel Xeon Phi, multi-thread is an important resource, so our future work emphasizes on leveraging *helper-thread prefetching* to address the memory wall problem.

Bibliography

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, pages 1–11. IEEE, 2010.

[2] G. Agrawal, J. Saltz, and R. Das. Interprocedural Partial Redundancy Elimination and Its Application to Distributed Memory Compilation. pages 258–269, June 1995. ACM SIGPLAN Notices, Vol. 30, No. 6.

[3] A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations. In *Proceedings of the 15th international conference on Supercomputing (ICS 2001)*, pages 486–500. ACM, 2001.

[4] A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems. *Journal of Instruction-Level Parallelism*, 6(7), 2004.

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *PPoPP*, 2008.

[6] S. Behnel, R. Bradshaw, and D. Seljebotn. Cython: C-extensions for Python, 2008.

[7] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory based on Type-Specific Memory Coherence. In *PPoPP*, 1990.

[8] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, apr 1994.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, pages 207–216, 1995.

[10] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[11] B. Cahoon and K. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *Proceedings of 2001 International Conference on Parallel Architectures and Compilation Techniques, (PACT 2001)*, pages 280–291. IEEE, 2001.

[12] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural Constant Propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[13] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review (SIGCOMM 2010)*, 40(5):20–26, 2010.

[14] C. Cascaval and D. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS 2003)*, pages 150–159. ACM, 2003.

[15] B. C. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *PPOPP*, pages 47–56, 2011.

[16] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Perf. Comput. Appl.*, 21(3), 2007.

[17] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-oriented Approach to Non-uniform Cluster Computing. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[18] S. Chatterjee, G. Blelloch, and M. Zagha. Scan Primitives for Vector Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (SC 1990)*, pages 666–675, Nov. 1990.

[19] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *ICS*, 1999.

[20] S. Che, J. Sheaffer, and K. Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*, pages 1–11. IEEE, 2011.

[21] T. Chilimbi, M. Hill, and J. Larus. Cache-Conscious Structure Layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1999)*, pages 1–12. ACM, 1999.

[22] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proceedings of 28th Annual International Symposium on Computer Architecture, (ISCA 2001)*, pages 14–25. IEEE, 2001.

[23] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[24] R. Cox. Regular Expression Matching Can Be Simple and Fast. http://swtch.com/ rsc/regexp/regexp1.html, January 2007.

[25] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[26] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The Design and Implementation of A Parallel Unstructured Euler Solver Using Software Primitives. Technical report, DTIC Document, 1992.

[27] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, page 4. IEEE/ACM, 2008.

[28] V. De Almeida and R. Güting. Indexing the Trajectories of Moving Objects in Networks*. *GeoInformatica*, 9(1):33–60, 2005.

[29] D. Delling, A. Goldberg, A. Nowatzyk, and R. Werneck. Phast: Hardware-Accelerated Shortest Path Trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, pages 921–931. IEEE, 2011.

[30] D. M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Trans. Prog. Lang. Syst.*, 13(2):291–294, 1991.

[31] H. Dietz and W. Cohen. A Massively Parallel MIMD Implemented by SIMD Hardware? Technical report, Purdue University, 1992.

[32] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *PLDI*, pages 229–241, 1999.

[33] X. Ding, K. Wang, and X. Zhang. ULCC: a User-Level Facility for Optimizing Shared Cache Performance on Multicores. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP 2011)*, pages 103–112. ACM, 2011.

[34] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer. Efficient hybrid execution of c++ applications using intel (r) xeon phi (tm) coprocessor. *arXiv preprint arXiv:1211.5530*, 2012.

[35] M. Dufour. Shed Skin-An Experimental (Restricted) Python to C++ Compiler (2009-09-30).

[36] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93. ACM, 2004.

[37] G. Ewing. Pyrex. A Language for Writing Python Extension Modules, 2006.

[38] F. Franchetti and M. Puschel. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*.

[39] A. Frank and A. Asuncion. UCI Machine Learning Repository, 2010.

[40] C. Garca, R. Lario, M. Prieto, L. Piuel, and F. Tirado. Vectorization of Multigrid Codes Using SIMD ISA Extensions. *Proceedings of the 2003 IEEE Inernational Parallel and Distributed Processing Symposium (IPDPS 2003)*, 0:58a, 2003.

[41] R. Garg and J. N. Amaral. Compiling Python to a Hybrid Execution Environment. In D. R. Kaeli and M. Leeser, editors, *GPGPU*, volume 425 of *ACM International Conference Proceeding Series*, pages 19–30. ACM, 2010.

[42] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *ASPLOS*, 2010.

[43] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, 1990.

[44] R. Ghiya, L. Hendren, and Y. Zhu. Detecting Parallelism in C Programs with Recursive Data Structures. In *Proceedings of 7th International Conference on Compiler Construction (CC 1998)*, pages 159–173. Springer, 1998.

[45] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, volume 1. the MIT Press, 1999.

[46] T. P. Group. PGI Accelerator Compilers OpenACC Getting Started Guide. 2013.

[47] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent Pattern Mining: Current Status and Future Directions. *Data Mining and Knowledge Discovery*, 15:55–86, 2007.

[48] R. Hanxleden and K. Kennedy. Relaxing SIMD Control Flow Constraints using Loop Transformations. In *PLDI*, pages 188–199. ACM, 1992.

[49] J. Hardwick. An Efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms. In *Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments (HIPS 1996)*, pages 105–114, April 1996.

[50] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. *High Performance Computing (HiPC 2007)*, pages 197–208, 2007.

[51] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ICS*, pages 13–24, 2013.

[52] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural Pointer Alias Analysis. *TOPLAS*, 21(4):848–894, 1999.

[53] M. Hind and A. Pioli. Which Pointer Analysis Should I Use? In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 113–123. ACM, 2000.

[54] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[55] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, pages 78–88. IEEE, 2011.

[56] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An Execution Strategy and Optimized Runtime Support for Parallelizing Irregular Reductions on Modern GPUs. In *Proceedings of the International Conference on Supercomputing (ICS 2011)*, pages 2–11. ACM, 2011.

[57] X. Huo, B. Ren, and G. Agrawal. A Programming System for Xeon Phis with Runtime SIMD Parallelization. In *Proceedings of the International Conference on Supercomputing*. ACM, 2014.

[58] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2.44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3D Graphics Computing. *IEEE Journal of Solid-State Circuits*, 35(7):1025 –1033, Jul 2000.

[59] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically Managed Data for CPU-GPU Architectures. In *CGO*, 2012.

[60] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. In *PLDI*, 2011.

[61] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011.

[62] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[63] W. Jiang, V. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-Core Environments. In *Proceedings of Conference on Cluster Computing and Grid (CCGRID)*, 2010.

[64] Y. Jo and M. Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2011)*, pages 463–482. ACM, 2011.

[65] Y. Jo and M. Kulkarni. Automatically Enhancing Locality for Tree Traversals with Traversal Splicing. In *Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2012)*. ACM, 2012.

[66] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open Source Scientific Tools for Python. *http://www. scipy. org/*, 2001.

[67] Y.-L. Ju and H. G. Dietz. Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation. In *LCPC*, 1992.

[68] C. Jung, D. Lim, J. Lee, and Y. Solihin. Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems. In *The 20th International Conference on Parallel and Distributed Processing Symposium, (IPDPS 2006)*, pages 10–pp. IEEE, 2006.

[69] Junichiro and Makino. Vectorization of a Treecode. *Journal of Computational Physics*, 87(1):148 – 160, 1990.

[70] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD 2010)*, pages 339–350. ACM, 2010.

[71] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-execution. *ACM SIGARCH Computer Architecture News (ASPLOS 2002)*, 30(5):159–170, 2002.

[72] S. Kim and H. Han. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, pages 55–64. ACM, 2012.

[73] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. *Arxiv preprint arXiv:0911.3456*, 2009.

[74] J. Knoop, O. Ruething, and B. Steffen. Lazy Code Motion. volume 27, pages 224–234, San Francisco, CA, 1992.

[75] J. Knoop and B. Steffen. Optimal Interprocedural Partial Redundancy Elimination. In *Proceedings of the Poster Session of the 4th International Conference on Compiler Construction (CC92)*, 1992.

[76] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 127–138. ACM, 2013.

[77] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How Much Parallelism is There in Irregular Applications? In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*. ACM, 2009.

[78] R. Ladelsky. Matrix Flattening and Transposing in GCC. In *GCC Summit Proceedings*, volume 2007, 2006.

[79] Lars and Hernquist. Vectorization of Tree Traversals. *Journal of Computational Physics*, 87(1):137 – 147, 1990.

[80] C. Lattner and V. S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI*, pages 129–142. ACM, 2005.

[81] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *SC*, 2010.

[82] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4), Nov. 1989.

[83] X. Li and G. Agrawal. Supporting XML-Based High-level Interfaces Through Compiler Technology. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC)*, Oct. 2003.

[84] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 273–282. ACM, 2013.

[85] J. Lu, A. Das, W. Hsu, K. Nguyen, and S. Abraham. Dynamic Helper Threaded Prefetching on the Sun UltraSPARC® CMP Processor. In *Proceedings of 38th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO 2005)*, pages 12–pp. IEEE, 2005.

[86] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. Optimizing the mapreduce framework on intel xeon phi coprocessor. *arXiv preprint arXiv:1309.0215*, 2013.

[87] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *ASPLOS*, pages 222–233, 1996.

[88] L. Luo, M. Wong, and W. Hwu. An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference (DAC 2010)*, pages 52–55. ACM, 2010.

[89] W. Ma and G. Agrawal. A Compiler and Runtime System for Enabling Data Mining Applications on GPUs. In *Proceedings of Principles and Practices of Parallel Programming (PPoPP)*, Feb. 2009.

[90] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Proceedings of the 2010 IEEE Cluster*. IEEE, 2010.

[91] S. S. Mannarswamy, R. Govindarajan, and R. Surendran. Region Based Structure Layout Optimization by Selective Data Copying. In *PACT*, pages 338–347. IEEE Computer Society, 2009.

[92] K. McKinley. A Compiler Optimization Algorithm for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, 1998.

[93] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. In *PPOPP*, pages 3–14. ACM, 2010.

[94] J. Meng, J. Sheaffer, and K. Skadron. Exploiting Inter-Thread Temporal Locality for Chip Multithreading. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010)*, pages 1–12. IEEE, 2010.

[95] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[96] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, pages 117–128. ACM, 2012.

[97] P. Miller. PyMPI-An Introduction to Parallel Python Using MPI. *Livermore National Laboratories, Jan*, 2002.

[98] J. S. Moore and R. Boyer. A Fast String Searching Algorithm. *Communications of the ACM*, pages 762–772, 1977.

[99] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *Commun. ACM*, 22(2):96–103, 1979.

[100] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[101] A. D. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *SC*, pages 1–13. IEEE, 2010.

[102] M. Nicola and J. John. XML Parsing: a Threat to Database Performance. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM 2003)*, pages 175–178. ACM, 2003.

[103] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI*, pages 132–143. ACM, 2006.

[104] C. NVIDIA. CUBLAS Library. *NVIDIA Corporation, Santa Clara, California*, 2008.

[105] C. NVIDIA. Programming Guide, 2008.

[106] C. NVIDIA. NVIDIA CUDA C Programming Guide. *NVIDIA Corporation*, 2013.

[107] T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006.

[108] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-Assisted Runtime Coherence Scheme. In *PACT*, 2012.

[109] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial Redundancy Elimination: a Simple, Pragmatic, and Provably Correct Algorithm. *Sci. Comput. Program.*, 48(1):1–20, 2003.

[110] S. Pennycook, C. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. 2013.

[111] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. Rice University, Department of Computer Science, 1989.

[112] J. Prins and D. Palmer. Transforming High-Level Data-Parallel Programs into Vector Operations. In *PPOPP*, pages 119–128. ACM, 1993.

[113] D. J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[114] J. Rao and K. Ross. Making B+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000)*, volume 29, pages 475–486. ACM, 2000.

[115] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar. Apricot: an Optimizing Compiler and Productivity Tool for x86-Compatible Many-Core Coprocessors. In *ICS*, pages 47–58, 2012.

[116] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Inc., 2010.

[117] B. Ren and G. Agrawal. Compiling Dynamic Data Structures in Python to Enable the Use of Multi-core and Many-core Libraries. In *PACT*, pages 68–77. IEEE Computer Society, 2011.

[118] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA 1999)*, pages 229–238. USENIX, 1999.

[119] T. Rognes and E. Seeberg. Six-Fold Speed-up of Smithwaterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. 16(8):699–706, 2000.

[120] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming Model for a Heterogeneous x86 Platform. In *PLDI*, 2009.

[121] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: Concurrent Programming for Modern Architectures. In *PPoPP*, 2007.

[122] D. Scarpazza and G. Russell. High-Performance Regular Expression Scanning on the Cell/B.E. Processor. In *Proceedings of the 23rd International Conference on Supercomputing (ICS 2009)*, pages 14–25. ACM, 2009.

[123] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-Throughput Sequence Alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.

[124] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB*, 4(11):795–806, 2011.

[125] T. Sharp. Implementing Decision Trees and Forests on a GPU. *Computer Vision (ECCV 2008)*, pages 595–608, 2008.

[126] A. Singh. Optimization of geometric multigrid for emerging multi-and manycore processors.

[127] S. Solomon, P. Thulasiraman, and R. Thulasiram. Exploiting Parallelism in Iterative Irregular Maxflow Computations on GPU Accelerators. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC 2010)*, pages 297 –304. IEEE, 2010.

[128] M. M. Strout, L. Carter, and J. Ferrante. Compile-Time Composition of Run-Time Data and Iteration Reorderings. In *PLDI*, pages 91–102, 2003.

[129] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11:419–422, June 1968.

[130] C. Trapnell and M. Schatz. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing*, 35(8):429–440, 2009.

[131] S. Unkule, C. Shaltz, and A. Qasem. Automatic Restructuring of GPU Kernels for Exploiting Inter-Thread Data Locality. In *Compiler Construction (CC 2012)*, pages 21–40. Springer, 2012.

[132] H. L. A. van der Spek, S. Groot, E. M. Bakker, and H. A. G. Wijshoff. A Compile/Run-time Environment for the Automatic Transformation of Linked List Data Structures. *IJPP*, 36(6):592–623, 2008.

[133] H. L. A. van der Spek, C. W. M. Holm, and H. A. G. Wijshoff. How to Unleash Array Optimizations on Code Using Recursive Data Structures. In T. Boku, H. Nakashima, and A. Mendelson, editors, *ICS*, pages 275–284. ACM, 2010.

[134] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 265–283. Springer Berlin / Heidelberg, 2009.

[135] M. Wegman and F. Zadeck. Constant Propagation with Conditional Branches. *TOPLAS*, 13(2):181–210, 1991.

[136] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 57–68. ACM, 2013.

[137] S. Wu, D. Jiang, B. Ooi, and K. Wu. Efficient B-Tree Based Indexing for Cloud Data Processing. *PVLDB*, 3(1):1207–1218, 2010.

[138] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721 – 724. IEEE, 2002.

[139] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par 2009*, pages 887–899. Springer, 2009.

[140] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-Fly Elimination of Dynamic Irregularities for GPU Computing. In *ASPLOS*, pages 369–380, 2011.

[141] E. Zhang, Y. Jiang, and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPOPP 2010)*, pages 203–212. ACM, 2010.

[142] W. Zhang, D. Tullsen, and B. Calder. Accelerating and Adapting Precomputation Threads for Effcient Prefetching. In *13th International Symposium on High Performance Computer Architecture, 2007. (HPCA 2007)*, pages 85–95. IEEE, 2007.

[143] Y. Zhang, W. Ding, J. Liu, and M. Kandemir. Optimizing Data Layouts for Parallel Computation on Multicores. In *PACT*, 2011.

[144] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 2004)*, pages 255–266. ACM, 2004.

[145] X. Zhou, W. Chen, and W. Zheng. Cache sharing management for performance fairness in chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT 2009)*, pages 384–393. IEEE, 2009.

[146] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP 2012)*, pages 129–140. ACM, 2012.