TOWARD COMPILER-DRIVEN ADAPTIVE EXECUTION AND ITS APPLICATION TO GPU ARCHITECTURES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Seyong Lee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2011

Purdue University

West Lafayette, Indiana

Dedicated to my lovely wife and daughter

ACKNOWLEDGMENTS

I would like to thank my advisor, Rudolf Eigenmann, for his insightful guidance throughout the research. His vision, experience, and patience helped me stay on the right track and finish this work. I am grateful to my committee members, Professor Samuel P. Midkiff, Professor Y. Charlie Hu, and Professor Dongyan Xu for the feedback they have provided throughout my doctoral studies. I am also thankful to Professor T. N. Vijaykumar and Professor Mithuna S. Thottethodi for their insightful comments during the MaRCO project.

Many thanks go to the former and current members of the ParaMount Research Group at Purdue ECE; especially I thank Hansang Bae, Sangik Lee, and Chirag Dave for their substantial support and advice on implementing my ideas using the Cetus compiler infrastructure. I also would like to thank Seung-Jai Min for his mentoring on both research and life as a graduate student, and Okwan Kwon for his help with various research tools.

I would like to acknowledge the financial support provided by the IT Scholarship of Ministry of Information and Communication Republic of Korea and the Samsung Lee Kun Hee Scholarship Foundation.

Completing a graduate degree is impossible without the support of friends and family. In particular, I want to give my thanks to Brian Armstrong and other members of Emmanuel Bible Church for their spiritual support and encouragement. Finally, special thanks go to my parents, my wife, my brother, and two sisters, for their continuous support and love.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

ABSTRACT

Lee, Seyong Ph.D., Purdue University, May 2011. Toward Compiler-driven Adaptive Execution and Its Application to GPU Architectures. Major Professor: Rudolf Eigenmann.

A major shift in technology from maximizing single-core performance to integrating multiple cores has introduced a new, heterogeneous arena to high-performance computing communities. Among several new parallel platforms, hardware accelerators, such as General-Purpose Graphics Processing Units (GPGPUs), have emerged as promising alternatives for high-performance computing. While a GPGPU provides an inexpensive, highly parallel system to application developers, its programming complexity poses significant challenges for developers. This dissertation explores compile-time and runtime techniques to improve programmability and to enable adaptive execution of programs in such architectures.

First, this dissertation examines the possibility of exploiting OpenMP shared memory programming model on stream architectures such as GPGPUs. This dissertation presents a compiler framework for automatic translation and optimization of standard OpenMP applications for executing on GPGPUs.

Second, this dissertation studies runtime tuning systems to adapt applications dynamically. In preliminary work, an adaptive runtime tuning system with emphasis on parallel irregular applications has been proposed.

Third, this dissertation focuses on creating an integrated framework where both the compiler framework and the tuning system are synergistically combined, such that compiler-translated GPGPU applications will be seamlessly adapted for the underlying system. For this goal, a new programming interface, called OpenMPC - OpenMP extended for CUDA, is proposed. OpenMPC provides an abstraction of the

complex CUDA programming model and offers high-level control over the involved parameters and optimizations. We have developed a fully automatic compilation and user-assisted tuning system supporting OpenMPC. Experiments on various programs demonstrate that the proposed system achieves performance comparable to hand-coded CUDA programs.

1. INTRODUCTION

1.1 Motivation

Recent shift in computer technology has introduced a range of diverse computing resources, such as multicore architectures and hardware accelerators, resulting in a new blue ocean for general-purpose high-performance computing. With the increased complexity of these new computing environments, however, finding efficient and convenient ways of programming these resources and achieving reasonable performance is one very challenging issue. Specially, hardware accelerators, such as General-Purpose Graphics Processing Units (GPGPUs), provide inexpensive, highly parallel systems to application developers. However, their programming complexity poses a significant challenge for developers. Moreover, complex interactions among the limited hardware resources make it more difficult to achieve a good performance. There has been growing research and industry interest in lowering the barrier of programming these devices [1–4]. Even though the CUDA programming model [5], recently introduced by NVIDIA, offers a more user-friendly interface, programming GPGPUs is still complex and error-prone, compared to programming general-purpose CPUs and parallel programming models, such as OpenMP [6].

OpenMP has established itself as an important method and language extension for programming shared-memory parallel computers. There are several advantages of OpenMP as a programming paradigm for GPGPUs.

- OpenMP is efficient at expressing loop-level parallelism in applications, which is an ideal target for utilizing GPU's highly parallel computing units to accelerate data-parallel computations.

- The concept of a master thread and a pool of worker threads in OpenMP's fork-join model represents well the relationship between the master thread running in a host CPU and a pool of threads in a GPU device.

- Incremental parallelization of applications, which is one of OpenMP's features, can add the same benefit to GPGPU programming.

The CUDA programming model provides a general-purpose multi-threaded Single Instruction, Multiple Data (SIMD) model for implementing general-purpose computations on GPUs. Although the unified processor model in CUDA abstracts underlying GPU architectures for better programmability, its unique memory model is partly exposed to programmers. Therefore, the manual development of high-performance codes in CUDA is more involved than in other parallel programming models such as OpenMP [7].

In this dissertation, we present a compiler framework for translating standard OpenMP shared-memory programs into CUDA-based GPGPU programs. For an automatic source-to-source transformation, several translation strategies have been developed. The goal of the proposed translation is to further improve programmability and make existing OpenMP applications amenable to execution on GPGPUs. In addition, we have identified several compile-time transformation techniques to fill the performance gap, caused by architectural differences between traditional shared-memory multiprocessors, served by OpenMP, and stream architectures, adopted by most GPUs.

The preliminary measurements of the resulting performance show that the proposed translator and compile-time optimizations work well on both regular and irregular applications. However, the performance variations in the preliminary results also reveal that complex interactions among the hardware resources may need fine-tuning for optimal performance.

For that purpose, this dissertation studies compiler-driven runtime tuning systems for dynamic adaptation of applications onto the underlying execution platform. In

preliminary work, we have proposed an adaptive runtime tuning system with emphasis on parallel irregular applications. For irregular applications, such as sparse matrix-vector (SpMV) multiplication kernels, static performance optimizations are difficult because memory access patterns may be known only at runtime. On parallel applications, computational load balancing and communication cost reduction are two key issues. To address these issues, we have developed an adaptive load-mapping and communication-algorithm-selection system.

For enabling the seamless integration of the OpenMP-to-CUDA translation system and the adaptive runtime tuning system, we propose a new programming interface, called OpenMPC - OpenMP extended for CUDA. Because OpenMPC is based on OpenMP, it provides the same level abstraction of the complex CUDA programming model as OpenMP. In addition, it provides users with high-level tools to control the involved parameters and optimizations without knowing the details of the complex CUDA programming and memory models. We have developed a fully automatic compilation and user-assisted tuning system supporting OpenMPC. The system includes tuning capabilities for generating, pruning, and navigating the search space of compilation variants.

1.2 Contributions of This Dissertation

This dissertation makes the following contributions:

- We present the first compiler framework for an automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications. It includes (1) the interpretation of OpenMP semantics under the CUDA programming model, (2) an algorithm to identify *kernel regions* (code sections to be executed on GPU), and (3) an algorithm for reducing CPU-GPU memory transfers. We have also identified several compile-time transformation techniques to optimize memory accesses in stream architectures such as GPGPUs: (1) generic OpenMP stream optimizations for general stream architectures

and (2) OpenMP-to-CUDA optimizations for the CUDA-based GPGPU architecture. Evaluation on fourteen OpenMP programs (two kernel benchmarks, three NAS OpenMP Benchmarks, and nine Rodinia Benchmarks), which include both regular and irregular applications, shows that the proposed system improves performance up to 68X (7X on average) over unoptimized translations (up to 245X over serial).

- We present an adaptive runtime tuning system for parallel irregular applications, such as distributed SpMV multiplication kernels. The proposed adaptive mapping system solves computational load-balancing problems by dynamically re-assigning loop iterations to processes, and the runtime selection system finds the best communication method to minimize the incurred communication cost. Evaluation of the proposed tuning system on 26 real sparse matrices from a wide variety of scientific and engineering applications shows that our tuning system reduces execution time up to 68.8% (30.9% on average) over a base parallel SpMV algorithm on a 32-node platform.

- To enable compiler-driven, adaptive execution of standard OpenMP programs for the underlying GPGPU architectures, we propose a new API called OpenMPC. OpenMPC provides a tuning environment that assists users in generating CUDA programs in many optimization variants without detailed knowledge of the CUDA programming and memory models. We have developed a fully automatic, parameterized compilation system to support OpenMPC, which includes several tools that assist users in performance tuning by suggesting applicable optimization parameters and generating all necessary compilation variants automatically. Evaluation of the effectiveness of OpenMPC on the fourteen OpenMP benchmarks shows that user-assisted tuning with optional manual modification on the input OpenMP programs improves the performance 1.24 times on average (up to 7.71 times) over un-tuned versions, which is 75% of the performance of hand-written CUDA versions. If we exclude one exceptional case, the aver-

age tuned performance is 92% of the manual CUDA performance. Moreover, a tuning-assisting tool eliminates on average 98.7% of the optimization search space for the tested programs.

1.3 Thesis Organization

The rest of this dissertation is organized as follows: Chapter 2 presents the compiler framework for an automatic source-to-source translation of standard OpenMP shared-memory programs into CUDA-based GPGPU programs, and Chapter 3 presents a preliminary work of the adaptive runtime tuning system. Chapter 4 discusses the new API called OpenMPC and the reference compilation and tuning system, which can be used as a basis of the combined framework that enables automatic translation and compiler-assisted, dynamic adaptation of OpenMP applications for the specific characteristics of the underlying systems, and Chapter 5 summarizes this dissertation.

2. OPENMP TO GPGPU: A COMPILER FRAMEWORK FOR AUTOMATIC TRANSLATION AND OPTIMIZATION

2.1 Introduction

GPGPUs have recently emerged as powerful vehicles for general-purpose high-performance computing. Although a new Compute Unified Device Architecture (CUDA) programming model from NVIDIA offers improved programmability for general computing, programming GPGPUs is still complex and error-prone. In this chapter, we present an automatic OpenMP to GPGPU translator to extend the ease of creating parallel applications with OpenMP to GPGPU architectures. Due to the similarity between OpenMP and CUDA programming models, we were able to convert OpenMP parallelism, especially loop-level parallelism, into the forms that best express parallelism in CUDA. However, the baseline translation of existing OpenMP programs does not always yield good performance. Performance gaps are due to architectural differences between traditional shared-memory multiprocessors (SMPs), served by OpenMP, and stream architectures, adopted by most GPUs. Even though the OpenMP programming model is platform-independent, most existing OpenMP programs were tuned to traditional shared-memory multiprocessors. We refer to *stream architectures* as those that operate on a large data space (or stream) in parallel, typically in SIMD manner and tuned for fast access to regular, consecutive elements of the data stream. In GPU architectures, optimization techniques designed for CPU-based algorithms may not perform well [8]. Also, GPUs face bigger challenges in handling irregular applications than SMPs, because of the stream architectures' preference for regular access patterns.

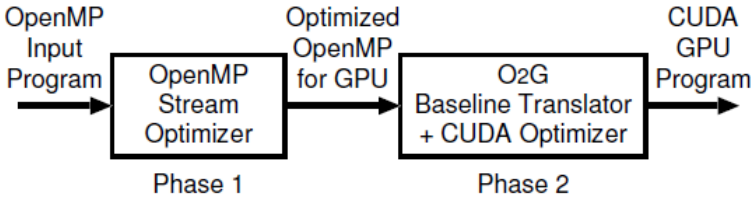


Fig. 2.1. Two-phase OpenMP-to-GPGPU compilation system. Phase 1 is an OpenMP stream optimizer to generate optimized OpenMP programs for GPGPU architectures, and phase 2 is an OpenMP-to-GPGPU (O_2G) translator with CUDA optimizations.

To address these issues, we propose compile-time optimization techniques and an OpenMP to GPGPU translation system, consisting of two phases: the OpenMP stream optimizer and the OpenMP-to-GPGPU (O_2G) baseline translator with CUDA optimizer, shown in Figure 2.1. The OpenMP stream optimizer transforms traditional CPU-oriented OpenMP programs into OpenMP programs optimized for GPGPUs, using our high-level optimization techniques: *parallel loop-swap* and *loop collapsing*. The O_2G translation converts the output of the OpenMP stream optimizer into CUDA GPGPU programs. The O_2G CUDA optimizer exploits CUDA-specific features.

We evaluate the baseline translation methods and the compile-time optimization techniques using diverse applications consisting of two kernel benchmarks, three NPB3.0-OMP NAS Parallel Benchmarks, and nine Rodinia Benchmarks [9], which have both regular and irregular applications. Experimental results show that the proposed compile-time optimization techniques can boost the performance of translated GPGPU programs up to 68 times over the unoptimized GPGPU translations.

The rest of this chapter is organized as follows: Section 2.2 provides an overview of the CUDA programming model, and Section 2.3 presents the baseline OpenMP to CUDA GPGPU translator. In Section 2.4, various compiler techniques to optimize the performance of GPGPU programs are explained, and Section 2.5 explains a set of transformation techniques supporting OpenMP to CUDA translation. Experimental results are shown in Section 2.6, and related work and summary are presented in Section 2.7 and Section 2.8, respectively.

2.2 Overview of GPGPU Architecture and CUDA Programming Model

GPGPUs supporting CUDA consist of a set of multiprocessors called *streaming multiprocessors (SMs)*, each of which contains a set of SIMD processing units called *streaming processors (SPs)*. Each SM has a fast on-chip *shared memory*, which is shared by SPs in the same SM, a fixed number of registers, which are logically partitioned among threads running on the SM, and special read-only caches (*constant cache* and *texture cache*), which are shared by SPs. A slow off-chip *global memory* is used for communications among different SMs.

The CUDA programming model is a general-purpose multi-threaded SIMD model for GPGPU programming. In the CUDA programming model, a GPU is viewed as a parallel computing coprocessor, which can execute a large number of threads concurrently. A CUDA program consists of a series of sequential and parallel execution phases. Sequential phases have little or no parallelism, and thus they are executed on the CPU as host codes. Parallel phases that exhibit rich data parallelism are implemented as a set of *kernel functions*, which are executed on the GPU. Each kernel function specifies GPU code to be executed in an SIMD fashion, by a number of threads invoked for each parallel phase.

In the CUDA model, threads are grouped as a grid of thread blocks, each of which is mapped to a streaming multiprocessor (*SM*) in the GPU device. In CUDA-supported GPU architectures, more than one thread block can be assigned to an SM, and threads within each thread block are mapped to SIMD processing units (*SPs*) in the SM. Threads in each thread block are split into SIMD groups called *warps*, which are the minimal scheduling units that can be switched dynamically to maximize hardware resource utilization. The number of thread blocks and the number of threads per thread block, which constitute a *thread batching*, are specified through language extensions at each kernel invocation.

In the CUDA programming model, a host CPU and a GPU device have separate address spaces. For a CPU to access GPU data, the CUDA model provides an API

```
1 __global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
 {
2   int i = threadIdx.x;
3   int j = threadIdx.y;
4   C[i][j] = A[i][j] + B[i][j];
 }

5 int main()
 {
   // Codes for allocating CPU input and output matrices are omitted.

   // Allocate GPU input and output matrices
6   float* gpu_A, gpu_B, gpu_C;
7   cudaMalloc((void **) &gpu_A, mem_size_A);
8   cudaMalloc((void **) &gpu_B, mem_size_B);
9   cudaMalloc((void **) &gpu_C, mem_size_C);

   // Copy input matrices from a host CPU to a GPU device
10  cudaMemcpy(gpu_A, host_A, mem_size_A, cudaMemcpyHostToDevice);
11  cudaMemcpy(gpu_B, host_B, mem_size_B, cudaMemcpyHostToDevice);

   // Execute the Kernel on the GPU
12   dim3 dimGrid(1,1,1);
13   dim3 dimBlock(N,N,1);
14   matAdd<<<dimGrid, dimBlock>>>(A, B, C);

   // Copy the result on the GPU back to the host
15   cudaMemcpy(host_C, gpu_C, mem_size_C, cudaMemcpyDeviceToHost);
 }
```

Fig. 2.2. Example CUDA program computing matrix addition

for explicit GPU memory management, including functions to transfer data between the CPU and the GPU.

Figure 2.2 shows an example CUDA program that computes a simple matrix addition. In this example, each thread calculates one element of the output matrix (line 4). The kernel function (*matAdd()*) in Figure 2.2 is executed by one thread block (line 12), which consists of $N * N$ threads (line 13). Line 10, 11, and 15 in the figure are the API calls to transfer data between the CPU and the GPU.

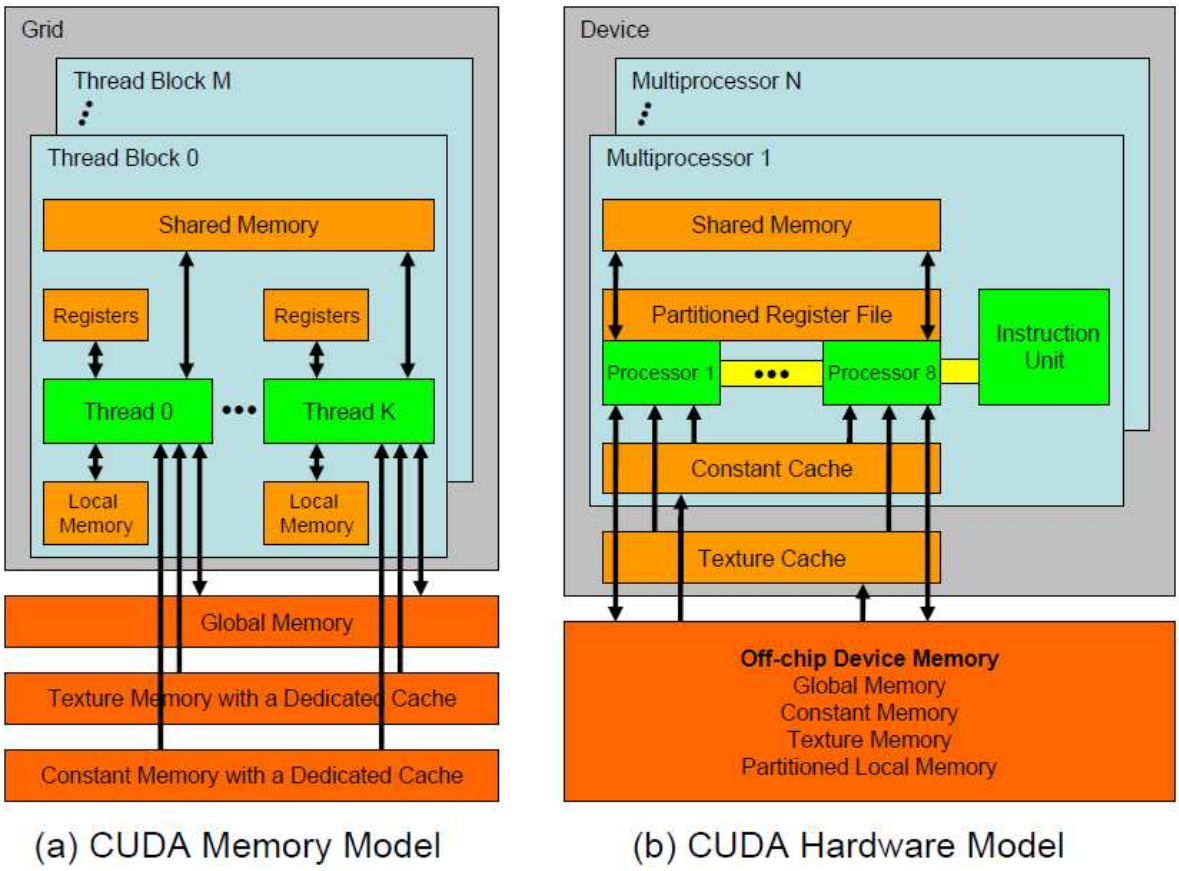


Fig. 2.3. CUDA memory model and hardware model

The CUDA memory model has an off-chip *global memory* space, which is accessible by all threads, an off-chip *local memory* space, which is private to each thread, a fast on-chip *shared memory* space, which is shared only by threads in a thread block, and *registers*, which are private to each thread. The CUDA memory model also has separate memory spaces to exploit specialized hardware memory resources: *constant memory* with a dedicated small cache for read-only global data that are frequently accessed by many threads across multiple thread blocks, and *texture memory* with a dedicated small cache for read-only array data accessed through built-in texture functions. The shared memory and the register bank in a streaming multiprocessor are dynamically partitioned among the active thread blocks running on the multiproces-

sor. Therefore, register and shared memory usages per thread block can be a limiting factor preventing full utilization of execution resources. Figure 2.3 shows the CUDA memory model and the underlying hardware model. As shown in Figure 2.3 (b), both *local memory* and *global memory* reside in the same off-chip DRAM memory; the cost to access the local memory is the same as the cost to access the global memory.

One fundamental limit of the CUDA model is that it does not support global synchronization mechanisms. Synchronization within a thread block can be enforced by using the *__syncthreads()* runtime primitive, which guarantees that all threads in the same thread block have reached the same program point, and data modified by threads in the same thread block are visible to all threads in the same block. However, synchronization across thread blocks can be accomplished only by returning from a kernel call, after which all threads executing the kernel function are guaranteed to be finished, and *global memory* data modified by threads in different thread blocks are guaranteed to be globally visible.

2.3 Baseline Translation of OpenMP into CUDA

This section presents a baseline translator, which performs a source-to-source conversion of an OpenMP program to a CUDA-based GPGPU program. The translation consists of several steps: (1) interpreting OpenMP semantics under the CUDA programming model and identifying *kernel regions* (code sections executed on the GPU), (2) outlining (extracting into subroutines) kernel regions and transforming them into CUDA kernel functions, and (3) analyzing shared/threadprivate data that will be accessed by the GPU and inserting necessary memory transfer calls. We have implemented these translation steps using the Cetus compiler infrastructure [10].

2.3.1 Interpretation of OpenMP Semantics under the CUDA Programming Model

OpenMP directives can be classified into four categories:

(1) *Parallel* construct (*omp parallel*) – this is the construct that specifies parallel regions. Parallel regions may be further split into sub-regions. The translator identifies eligible *kernel regions* among the (sub-)regions and transforms them into GPU kernel functions.

(2) Work-sharing constructs (*omp for*, *omp sections*) – the compiler interprets these constructs to partition work among threads on the GPU device. Each iteration of an *omp for* loop is assigned to a thread, and each section of *omp sections* is mapped to a thread.

(3) Synchronization constructs (*omp barrier*, *omp flush*, *omp critical*, etc.) – these constructs constitute *split points*, points where a parallel region must be split into two sub-regions; each of the resulting sub-regions may become a kernel region. This split is required to enforce a global synchronization in the CUDA programming model, as explained in Section 2.2.

(4) Directives specifying data properties (*omp shared*, *omp private*, *omp threadprivate*, etc.) – these constructs are used to map data into the GPU memory spaces. As mentioned in Section 2.2, the CUDA memory model requires explicit memory transfers for threads invoked for a kernel function to access data on the CPU. OpenMP *shared* data are shared by all threads, and OpenMP *private* data are accessed by a single thread. In the CUDA memory model, the shared data can be mapped to *global memory*, and the private data can be mapped to *registers* or *local memory* assigned for each thread. OpenMP *threadprivate* data are private to each thread, but they have global lifetimes, as do static data. The semantics of *threadprivate* data can be implemented by expansion, which allocates copies of the *threadprivate* data on *global memory* for each thread. Because the CUDA memory model allows several specialized memory spaces, certain data can take advantage of the specialized memory resources; read-only shared data can be assigned to either *constant memory* or *texture memory* to exploit temporal locality through dedicated caches, and frequently reused shared data can use fast memory spaces, such as *registers* and *shared memory*, as a cache.

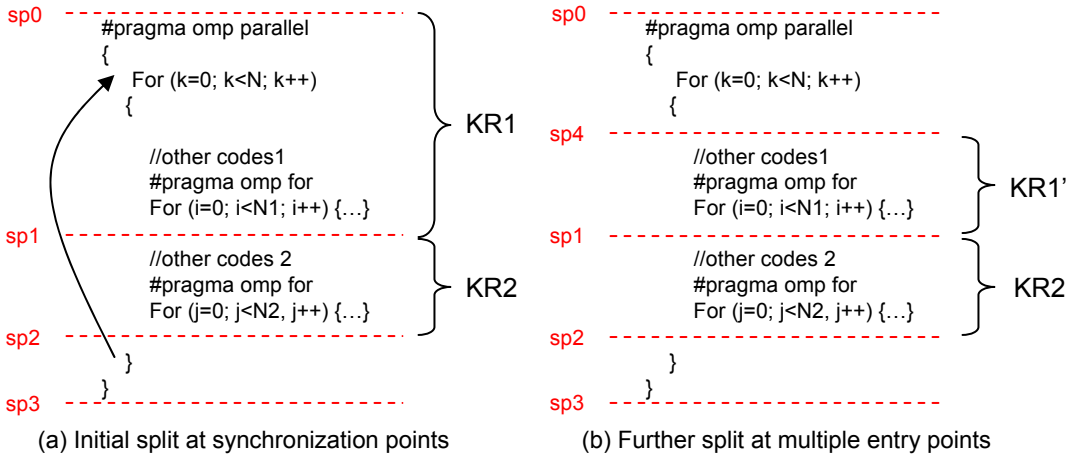


Fig. 2.4. Parallel region example showing how multiple splits are applied to identify kernel regions. *sp0* - *sp4* are split points enforced to preserve OpenMP semantics, and *KR1'* and *KR2* are kernel regions to be converted into kernel functions.

2.3.2 OpenMP to CUDA Baseline Translation

The previous subsection described the interpretation of OpenMP semantics under the CUDA programming model. The next step performs the actual translation into a CUDA program. A simple translation scheme might convert all code sections specified by work-sharing constructs into kernel functions, since work-sharing constructs contain the only true parallel code in OpenMP. Other sub-regions, within an *omp parallel* but outside of work-sharing constructs, are executed by one thread (*omp master* and *omp single*), serialized among threads (*omp ordered* and *omp critical*), or executed redundantly among participating threads. However, our compiler includes some of these sub-regions into kernel regions, thus redundantly executing them; this method may reduce expensive memory transfers between the CPU and the GPU.

With these concepts in mind, we will explain the baseline translation schemes in the following subsections.

Identifying Kernel Regions

The compiler targets OpenMP parallel regions as potential *kernel regions*. As explained above, these regions may be split at synchronization constructs. Among the resulting sub-regions, the ones containing at least one work-sharing construct become kernel regions.

The translator must consider that split operations may break the control flow semantics of the OpenMP programming model, if the *split points* lie within control structures. In the OpenMP programming model, most directives work only on a *structured block* – a block of code with one entry and one exit point. If a parallel region is split in the middle of a control structure, the resulting kernel regions may become unstructured blocks. Figure 2.4 (a) shows an example where a split operation would result in incorrect control flow. In the OpenMP programming model, a *flush* synchronization construct is implied at the entry to and exit from *parallel* regions (*sp0* and *sp3* in Figure 2.4 (a)) and at the exit from work-sharing regions (*sp1* and *sp2*), unless a *nowait* clause is present. The split operation identifies two kernel regions: *KR1* and *KR2*. Because the first kernel region, *KR1*, has multiple entry points, outlining this region will break control flow semantics. To solve this problem, our translator splits *KR1* further at multiple entry points. In Figure 2.4 (b), an additional split is applied at *sp4*, turning *KR1* into a structured block with correct control flow (*KR1'*).

The overall algorithm to identify kernel regions is shown in Figure 2.5. The rationale behind this top-down splitting algorithm is to merge as many work-sharing regions as possible for minimal overheads by kernel invocations and CPU-GPU data transfers.

Transforming a Kernel Region into a Kernel Function

The translator outlines the identified kernel regions into CUDA kernel functions and replaces the original regions with calls to these functions.

```
Kernel splitting algorithm
Input: OpenMP program
Output: OpenMP program where eligible kernel regions are identified
for each parallel region R in the input program
    for each split point in R
        divide R into two sub−regions at the split point
    build control flow graph cfg for R
    for each sub−region SR in R
        for each entry/exit points other than the one at the top/bottom of SR
            divide SR into two sub−regions at such entry/exit points
        for each sub−sub−region SSR in SR
            if SSR contains an OpenMP work−sharing construct
                annotate SSR as an eligible kernel region
```

Fig. 2.5. Algorithm to identify kernel regions, which are parallel regions to be transformed into kernel functions

At this stage, two important translation steps are involved: work partitioning and data mapping. For work partitioning, each iteration of *omp for* loops and each section of *omp sections* are assigned to a thread, and remaining code sections in a kernel region are executed redundantly by all participating threads. To decide the *thread batching* for a kernel function, the translator calculates the maximum partition size among parallel work contained in the kernel region. By default, the maximum partition size becomes the total number of threads executing the kernel function. Because the number of thread blocks and the thread block size determine the mapping of threads onto SMs (*thread batching*), these two parameters can be set through command-line options or user directives. In this case, the translator performs necessary tiling transformations to fit the work partition into the specified *thread batching*. Figure 2.6 shows a work partitioning example, where input OpenMP code contains a parallel region with two omp-for loops. In Figure 2.6 (a), the partition size (the number of iterations) of the first omp-for loop is 4096, and that of the second loop is 8192. Therefore, the maximum partition size is 8192, and the GPU code in Figure 2.6 (b) will be executed by 32 thread blocks, if the default size of a thread block is 256 (8192

/ 256 = 32). However, the GPU code in Figure 2.6 (c), where a programmer explicitly set the number of thread blocks to 16, will be executed by 16 thread blocks, regardless of the actual maximum partition size. Instead, the iteration spaces of each loop are tiled by the chunk size of 4096, which is a maximum partition size enforced by the programmer (256 * 16 = 4096).

After work partitioning, the compiler constructs the sets of shared data and private data used in the kernel region, using the information specified by the OpenMP data property constructs. In the OpenMP programming model, data is shared by default, including data with global scope or with heap-allocated storage. For the data that are referenced in the region, but not in a construct, the compiler can determine their sharing attributes using the OpenMP data sharing rules. Basic data mapping follows the rule explained in Section 2.3.1; shared data are mapped to *global memory*, threadprivate data are replicated and allocated on *global memory* for each thread, and private data are mapped to *register banks* assigned for each threads. If the size of private data mapped to the *register banks* is too big, the CUDA compiler automatically reassigns some of the data on *local memory*.

As a part of the data mapping step, the compiler inserts necessary memory transfer calls for the *shared* and *threadprivate* data accessed by each kernel function. A basic strategy is to move all the shared data that are accessed by kernel functions, and copy back the shared data that are modified by kernel functions. (threadprivate data transfers are decided by OpenMP semantics; for example, if an enclosing parallel region has a *copyin* clause, threadprivate variables in the copyin clause are copied from CPU to GPU before the corresponding kernel function is launched.) However, not all shared data are used by the CPU after the kernel completes. Also, data in the GPU *global memory* are persistent across kernel calls. Therefore, not all these data transfers are needed. The compiler optimization techniques to eliminate redundant data transfers will be discussed in the following section.

Additionally, during this translation, if a *omp for* loop contains a *reduction* clause, the compiler replaces the reduction operation with the two-level tree reduction form

```
#pragma omp parallel private(i)
{
    #pragma omp for nowait
    for(i = 0; i < 4096; i++)
        c[i] = a[i] + b[i];
    #pragma omp for
    for(i = 0; i < 8192; i++)
        f[i] = d[i] * e[i];
}
```

(a) input OpenMP code, where a parallel region contains two omp-for loops

```
i=_gtid; //_gtid is a global GPU thread identifier
if(i < 4096) //Check the bound of the first omp-for loop
    c[i] = a[i] + b[i];
i=_gtid;
if(i < 8192) //Check the bound of the second omp-for loop
    f[i]=  d[i] * e[i];
```

(b) GPU code when thread block size is set to 256

```
_ti_100_100 = _gtid; //_gtid is a global GPU thread identifier
if(_ti_100_100 < 4096) //Check the bound set by the user
    for(i = _ti_100_100 + 0; i < 4096; i += 4096)
        c[i] = a[i] + b[i];
_ti_100_100  = _gtid;
if(_ti_100_100 < 4096) //Check the bound set by the user
    for(i = _ti_100_100 + 0; i < 8192; i += 4096)
        f[i] = d[i] * e[i];
```

(c) GPU code when thread block size is set to 256, and the number of thread blocks is set to 16

Fig. 2.6. Work partitioning example showing how *thread batching* parameters affect the iteration-to-thread mapping. Figure (b) and (c) show only a part of transformed GPU kernel functions; other codes, such as the corresponding kernel function calls, GPU memory allocations, and memory transfer calls, are not shown.

proposed in [11]: a local parallel reduction within each thread block, followed by a host-side global reduction across thread blocks.

In the baseline translation scheme, *omp critical* regions are executed on the host CPU since the *omp critical* construct involves global synchronizations, which are

expensive on GPU kernel executions due to kernel splits, and the semantic of *omp critical* requires serialized execution of the specified region. However, if the *critical* regions have reduction forms, the same transformation technique used to interpret a *reduction* clause [11] can be applied.

2.4 Compiler Optimizations

We describe our two-phase optimization system; the first is the OpenMP stream optimizer, and the second is O_2G (OpenMP-to-GPGPU) CUDA optimizer.

2.4.1 OpenMP Stream Optimizations

Both the OpenMP and GPGPU models are suitable for expressing data parallelism. However, there are important differences. GPUs are designed as massively parallel machines for concurrent execution of thousands of threads, each executing the same code on different data (SIMD). GPU threads are optimized for fine-grain data parallelism, which is characterized by regular memory accesses and regular program control flow. On the other hand, OpenMP threads are more autonomous, typically execute coarse-grain parallelism, and are able to handle MIMD computation.

The OpenMP stream optimizer addresses these differences by transforming the traditional CPU-oriented OpenMP programs into GPU-style OpenMP programs. One benefit of this high-level translation method is that the user can see the optimization result at the OpenMP source code level.

Intra-Thread vs. Inter-Thread Locality

In OpenMP, data locality is often exploited within a thread (intra-thread locality), but less so among threads that are executed on different CPU nodes (inter-thread locality). This is because spatial data locality within a thread can increase cache utilization, but locality among OpenMP threads on different CPUs of cache-coherent

shared-memory multiprocessors can incur false-sharing. Therefore, in OpenMP, parallel loops are usually block-distributed, rather than in a cyclic manner.

In contrast to OpenMP, inter-thread locality between GPU threads plays a critical role in optimizing off-chip memory access performance. For example, in CUDA, GPU threads that access the off-chip memory concurrently can coalesce their memory accesses to reduce the overall memory access latency. These coalesced memory accesses can be accomplished if such GPU threads exhibit inter-thread locality where adjacent threads access adjacent locations in the off-chip memory. In the OpenMP form, a cyclic distribution of parallel loops can expose inter-thread locality.

The following subsections describe our two compile-time optimization techniques, *parallel loop-swap* and *loop collapsing*, to enhance inter-thread locality of OpenMP programs on GPGPUs.

Parallel Loop-Swap for Regular Applications

In this section, we introduce a *parallel loop-swap* optimization technique to improve the performance of regular data accesses in nested loops. Previously, we have described how cyclic distribution can improve inter-thread locality in a singly nested loop via an OpenMP `schedule(static, 1)` clause. However, in a nested loop, we need advanced compile-time techniques to achieve this goal. For example, the input OpenMP code shown in Figure 2.7 (a) has a doubly nested loop, where the outer loop is parallelized with a block distribution of iterations, which prevents the O_2G translator from applying the coalesced memory optimization to the accessed arrays.

To solve this problem, our OpenMP stream optimizer performs a *parallel loop-swap* transformation. We define *continuous memory access* as a property of an array in a loop-nest, where the array subscript expression increases monotonically with a stride of one, and the adjacent elements accessed by the subscript expression are continuous in the memory. An array access with *continuous memory access* is a candidate for the coalesced memory access optimization.

```
#pragma omp parallel for
for(i = 1; i <= SIZE; i++)
    for(j = 1; j <= SIZE; j++)
      a[i][j] = (b[i - 1][j] + b[i + 1][j]
              + b[i][j - 1] + b[i][j + 1]) / 4;
```

(a) input OpenMP code

```
#pragma omp parallel for
for(i = 1; i <= SIZE; i++) {
    #pragma cetus parallel
    for(j = 1; j <= SIZE; j++)
        a[i][j] = (b[i - 1][j] + b[i + 1][j]
                + b[i][j - 1] + b[i][j + 1]) / 4;
}
```

(b) Cetus parallelized OpenMP code

```
#pragma omp parallel for schedule(static, 1)
for(j = 1; j <= SIZE; j++)
    for(i = 1; i <= SIZE; i++)
        a[i][j] = (b[i - 1][j] + b[i + 1][j]
                + b[i][j - 1] + b[i][j + 1]) / 4;
```

(c) OpenMP output by OpenMP stream optimizer

```
// tid is a GPU thread identifier
for(tid = 1; tid <= SIZE; tid++)
    for(i = 1; i <= SIZE; i++)
        a[i][tid] = (b[i - 1][tid] + b[i + 1][tid]
                  + b[i][tid - 1] + b[i][tid + 1]) / 4;
```

(d) internal representation in O2G translator

```
// Each iteration of the parallel-for loop
// is cyclic-distributed to each GPU thread
if(tid <= SIZE) {
    for(i = 1; i <= SIZE; i++)
        a[i][tid] = (b[i - 1][tid] + b[i + 1][tid]
                  + b[i][tid - 1] + b[i][tid + 1]) / 4;
}
```

(e) GPU code

Fig. 2.7. Regular application example from JACOBI, to show how *parallel loop-swap* is applied to nested parallel loops to improve the inter-thread locality

The OpenMP stream optimizer performs *parallel loop-swap* in five steps: (1) for a given OpenMP parallel loop nest with an OpenMP work-sharing construct on loop L, the compiler finds a set of *candidates*, arrays with *continuous memory access* within the loop-nest. (2) The compiler selects the loop, L^*, whose index variable increments the subscript expression of the array accesses in *candidates* by one. (3) The compiler finds all possible parallel loops. (4) If all the loops between L and (including) L^* can be parallelized, these two loops are interchanged. (5) An OpenMP parallel work-sharing construct with cyclic distribution is added to loop L^*.

Figure 2.7 (b) shows the result of step 3, where a Cetus *parallel* pragma is added to the discovered parallel loop. Figure 2.7 (c) illustrates the result of the *parallel loop-swap* transformation performed by the OpenMP stream optimizer. The O_2G translator converts this transformed OpenMP code into the internal representation, as shown in Figure 2.7 (d). Figure 2.7 (e) shows the CUDA GPU code.

Loop Collapsing for Irregular Applications

Irregular applications pose challenges in achieving high performance on GPU programs because stream architectures are optimized for regular program patterns. In this section, we propose a *loop-collapsing* technique, which improves the performance of irregular OpenMP applications on such architectures.

There are two main types of irregular behavior in parallel programs: (1) data access patterns among threads due to indirect array accesses and (2) different control flow paths taken by different threads, such as in conditional statements. Irregular data accesses prevent the compiler from applying the memory coalescing technique because it cannot prove *continuous memory access* in the presence of indirect references. Irregular control flow prevents the threads from executing fully concurrently on the GPU's SIMD processors.

Figure 2.8 (a) shows one of the representative irregular program patterns in scientific applications. It exhibits both irregular data access patterns caused by the

```
#pragma omp parallel for
for(i = 0; i < NUM_ROWS; i++)
    for(j = rowptr[i]; j < rowptr[i + 1]; j++)
        w[i] += A[j] * p[col[j]];
```

(a) input OpenMP code

```
#pragma omp parallel for
for(i = 0; i < NUM_ROWS; i++) {
    #pragma cetus parallel reduction(+:w[i])
    for(j = rowptr[i]; j < rowptr[i + 1]; j++)
        w[i] += A[j] * p[col[j]];
 }
```

(b) Cetus parallelized OpenMP code

```
#pragma omp parallel for collapse(2) schedule(static, 1)
for(i = 0; i < NUM_ROWS; i++)
    for(j = rowptr[i]; j < rowptr[i + 1]; j++)
        w[i] += A[j] * p[col[j]];
```

(c) OpenMP output by OpenMP stream optimizer

```
// collapsed loop
for(tid1 = 0; tid1 < rowptr[NUM_ROWS]; tid1++)
    l_w[tid1] = A[tid1] * p[col[tid1]];

// For each GPU thread, a new thread-id, tid2,
// is assigned for the reduction loop
for(tid2 = 0; tid2 < NUM_ROWS; tid2++)
    for(j = rowptr[tid2]; j < rowptr[tid2 + 1]; j++)
        w[tid2] += l_w[j];
```

(d) internal representation in O2G translator

```
if(tid1 < rowptr[NUM_ROWS])
    l_w[tid1] = A[tid1] * p[col[tid1]];
if(tid2 < NUM_ROWS)
    for(j = rowptr[tid2]; j < rowptr[tid2 + 1]; j++)
        w[tid2] += l_w[j];
```

(e) GPU code

Fig. 2.8. Example of an irregular application, CG NAS Parallel Benchmark. *Loop-collapsing* eliminates irregular control flow and improves inter-thread locality

indirect array accesses in `A`, `p`, and `col` and control flow divergence because the inner loop depends on the value of array `rowptr`; different GPU threads will execute different numbers of inner loop iterations. Our first optimization technique, *parallel loop-swap*, cannot be applied to this irregular case because the dependency prevents loop interchange.

In this example, the compiler tries to collapse the doubly nested loop into a single loop and prove that the accesses to array `A` and `col` are *continuous memory accesses*. This technique eliminates both irregular data accesses to arrays `A` and `col` as well as control flow divergence in the inner loop. To prove the *continuous memory access* property to array `A` and `col`, one needs to prove that `i`, `rowptr`, and `j` are all monotonically increasing with stride one. Since both `i` and `j` are loop index variables, this proof can be done at compile-time. However, the monotonicity of array `rowptr` cannot be proved at compile-time if it is read from a file. In that case, the compiler inserts a runtime check, after the array `rowptr` is defined, to verify monotonicity.

The *loop-collapsing* optimization is implemented in three steps. First, for each perfectly nested loop in the OpenMP source, the OpenMP stream optimizer applies the Cetus compiler's parallelization techniques to find all possible parallel loops in a given nest (Figure 2.8 (b)). Second, among parallel loops, the optimizer performs monotonicity checks to identify the parallel loops eligible for loop-collapsing. Third, the optimizer annotates an OpenMP clause *collapse (k)* to indicate that the first k nested loops can be collapsed (Figure 2.8 (c)). The O_2G translator reads this optimized OpenMP program and transforms the loops into a single loop. Figure 2.8 (d) shows two loops in the internal representation of the O_2G translator. The first loop is the output of *loop-collapsing*, where each GPU thread executes one iteration of the collapsed loop. The second loop performs reduction computation.

The *loop-collapsing* transformation improves the performance of irregular OpenMP applications on GPUs in three ways:

- The amount of parallel work (the number of iterations, to be executed by GPU threads) is increased.

- Inter-thread locality is increased, especially for cases where *parallel loop-swap* cannot be applied.
- Control flow divergence is eliminated, such that adjacent threads can be executed concurrently in an SIMD manner.

2.4.2 O_2G CUDA Optimizations

This section describes the CUDA optimizer, a collection of optimization techniques for translating OpenMP programs into actual GPGPU programs. These optimizations differ from those in the OpenMP stream optimizer in that they are specific to features of the CUDA memory architecture.

Matrix Transpose for Threadprivate Array

The OpenMP-to-GPGPU translator must handle the mapping of *threadprivate* data. When *threadprivate* array is placed into *global memory*, the translator implements correct semantics by expanding the *threadprivate* array for each thread. Row-wise expansion is a common practice to preserve intra-thread locality in traditional shared-memory systems. However, row-wise array expansion would cause uncoalesced memory accesses in GPUs. In this case, *parallel loop-swap* can not be applied due to dependencies or *threadprivate* semantics.

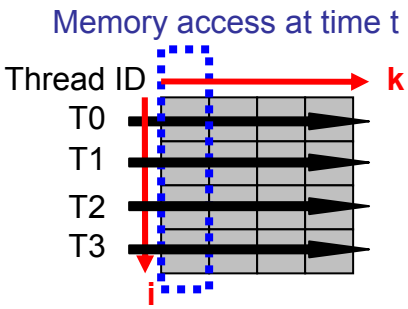
The *matrix transpose* transformation solves this problem by converting the array into a form where the *threadprivate* array is expanded in a column-wise manner. *Matrix transpose* changes intra-thread array access patterns from row-wise to column-wise, so that adjacent threads can access adjacent data, as needed for coalesced accesses. Figure 2.9 illustrates the *matrix transpose* optimization; Figure 2.9 (a) shows a synthetic parallel region, where an one-dimensional array b is a threadprivate variable. The naively translated kernel function of the parallel region is shown in Figure 2.9 (b), and the corresponding memory access pattern is shown in Figure 2.9

```
#pragma omp threadprivate(b)
...
#pragma omp parallel for private(i, k)
for (i = 0; i < 4096; i++)
    for (k=0; k<10; k++)
        b[k] = a[i] + k;
```

(a) Input OpenMP code

```
__global__ void kernel(int *a, int b[][10]) {
    int * b_0; int i; int k;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=_gtid;
    if (i<4096) {
        b_0=(((int * )b)+(_gtid*10));
        for (k=0; k<10; k ++ )
            b_0[k]=(a[i]+k);
    }
}
```

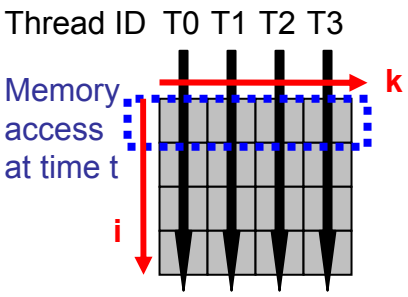
(b) GPU code w/o Matrix Transpose optimization



(c) GPU Thread memory access pattern 1

```
__global__ void kernel(int *a, int *b, size_t pitch__b) {
    int * b_0; int i; int k;
    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
    int _gtid = (threadIdx.x+(_bid*blockDim.x));
    i=_gtid;
    if (i<4096) {
        b_0=((int *)b)+_gtid;
        for (k=0; k<10; k ++ )
            *((int *)(((char *)b_0)+k*pitch__b)) = a[i]+k;
    }
}
```

(d) GPU code w/ Matrix Transpose optimization



(e) GPU Thread memory access pattern 2

Fig. 2.9. Matrix transpose example

(c), where each row represents a replicated, private copy of the threadprivate array *b* in the GPU global memory. In this memory layout, each thread will access each row, and thus adjacent threads can not access adjacent data at a given time *t*. However, if the memory layout is transposed such that each column represents a private copy of the threadprivate array, as shown in Figure 2.9 (e), adjacent threads will access data in adjacent columns, enabling *coalesced global memory access*. Necessary code change for accessing this transposed array is shown in Figure 2.9 (d), where a variable *pitch__b* contains the column length in bytes of the two-dimensional array of the replicated threadprivate variable. Because threadprivate data are allocated in the global memory, coalesced accesses enabled by the matrix transpose optimization can lead to quite huge performance improvement, as shown in the evaluation of NAS OpenMP Parallel Benchmark *EP* (refer to Section 2.6.2).

Techniques to Exploit GPU On-chip Memories

In the CUDA model, the global memory space is not cached – this is not true for some CUDA GPUs, but most of existing CUDA GPUs do not support hardware-maintained caching of the global memory. Therefore, to exploit temporal locality, frequently accessed global data must be explicitly loaded into fast memory spaces, such as *registers* and *shared memory*. In CUDA, exploiting temporal locality both within threads (intra-thread locality) and across threads (inter-thread locality) is equally important. *Shared memory* is shared by threads in a thread block, and the dedicated caches for *texture memory* and *constant memory* are shared by all threads running on the same multiprocessor.

In traditional shared-memory systems, temporal locality across threads is automatically exploited by the hardware caches. Therefore, in the OpenMP programming model, most existing caching-related optimizations focus on temporal locality within a thread only. However, under the CUDA memory model, software cache management for both intra- and inter-thread locality is important.

Our compiler performs the requisite *data flow analysis* to identify temporal locality of global data and inserts the necessary caching code. Table 2.1 shows caching strategies for each global data type. The baseline translator maps global shared data into *global memory*, but depending on the data attributes, they can be cached in specific fast memory spaces, as shown in the table. Even though no locality exists, putting read-only shared scalar variables in shared memory can also be beneficial, since it can reduce global memory traffic; passing read-only shared scalar variables as kernel arguments puts the data on shared memory without involving global memory.

Table 2.1
Caching strategies for global shared data with temporal locality. In *A(B)* format, *A* is a primary storage for caching, but *B* may be used as an alternative. *Reg* denotes *Registers*, *CC* means *Constant Cache*, *SM* is *Shared Memory*, and *TC* represents *Texture Cache*.

	Temporal locality	
	Intra-thread	Inter-thread
R/O shared scalar	Reg (CC or SM)	CC (SM)
R/W shared scalar	Reg (SM)	SM
R/O shared array	TC (CC or SM)	TC (CC or SM)
R/W shared array	SM	SM

Figure 2.10 shows the overall algorithm to identify possible caching strategies for OpenMP shared variables, which are main components constituting GPU global memory space. One limit of the current implementation is that it does not cache the whole array of an OpenMP shared, array-type variable onto the GPU *shared memory*, even though each element of the shared array may be cached on the *shared memory*. Because *shared memory* is very small, caching the whole shared array often requires tiling transformation, which is not yet supported.

Another CUDA-specific caching optimization is to allocate a CUDA-private array in *shared memory* if possible. In the CUDA memory model, a device-local array is allocated in *local memory*, which is a part of the off-chip DRAM; accessing the *local*

```
Locality analysis
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program annotated with possible caching strategies for shared
        variables accessed in each kernel region
for ( kernel region p : list of kernel regions in the input program )
    sharedVars = a set of shared variables accessed in p
    for ( shared variable s : sharedVars )
        useCnt = number of instances in p where s is used (read)
        defCnt = number of instances in p where s is defined (written)
        isStruct = true  //If s is a struct type
                   false //Otherwise
        isScalar = true  //If s is a scalar variable
                   false //Otherwise
        if ( (useCnt <= 1) and (defCnt <= 1) )
            if ( defCnt == 0 ) //R/O variable, but no intra-thread locality
                add s into ConstantSet //s is eligible for caching on constant
                                       //cache due to possible inter-thread locality.
                if ( isScalar )
                    add s into SharedROSet //Caching R/O scalar variable on shared
                                           //memory can reduce global memory access.
                else if ( !isStruct and (s is one-dimensional array) )
                    add s into TextureSet //s is eligible for caching on texture
                                          //cache due to possible inter-thread locality.
        else if ( defCnt == 0 ) //R/O variable with locality
            add s into ConstantSet
            if ( isScalar ) add s into SharedROSet
                if ( !isStruct ) add s into RegisterROSet
            else if ( !isStruct ) //s is array type and not struct type
                if ( s is one-dimensional array ) add s into TextureSet
                for (expression exp : array-access expressions accessing s in p)
                  if ( exp is used more than once in p )
                      add exp into RegisterROSet //An array element with locality
                                                 //is eligible for caching on a register.
        else //R/W variable with locality
            if ( isScalar ) add s into SharedRWSet
                if ( !isStruct ) add s into RegisterRWSet
            else if ( !isStruct ) //s is array type and not struct type
                for (expression exp : array-access expressions accessing s in p)
                  if ( exp is used more than once in p )
                      add exp into RegisterRWSet
```

Fig. 2.10. Locality analysis to identify possible caching strategies for OpenMP shared variables

memory is as slow as accessing *global memory*. If the private array size is small, it may be allocated on *shared memory* using an array expansion technique.

However, complex interactions among limited hardware resources may cause performance effects that are difficult to control statically. Therefore, our compiler framework provides language extensions and command line options for a programmer or automatic tuning system to guide these optimizations.

Resident GPU variable analysis

Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program annotated with user−directives containing shared variables that are resident on the GPU global memory

gResidentGVars_in(program entry node) = {}
for(node m : predecessor nodes of a node n)
  gResidentGVars_in(n) ^= gResidentGVars_out(m) *//^ is an intersection operation*
gResidentGVars_out(n) = gResidentGVars_in(n) + GEN(n) − KILL(n)
  where,
    GEN(n) = a set of shared variables whose GPU variables are globally allocated
        //If n is an exit node from a kernel region
      {} *//Otherwise*
    KILL(n) = a set of reduction variables in a kernel region *//If n is an exit*
        //node from the kernel region
      a set of shared variables modified *//If n represents a node in a CPU*
        //region
      a set of R/O shared scalar variables in a kernel region
        //If the variables do not exist in gResidentGVars_in set,
        //and if optimization to cache shared scalar variable on shared
        //memory is on,
        //and if n is an exit node from the kernel region
      {} *//Otherwise*

Fig. 2.11. Interprocedural analysis to identify resident GPU variables, which does not include a function-call-handling part and annotation generation part

Techniques to Optimize Data Movement between CPU and GPU

A final, important step of the actual OpenMP to GPGPU translation is the insertion of CUDA memory transfer calls for the *shared* and *threadprivate* data accessed by each kernel function. The basic data movement strategy is to transfer data accessed by a kernel function from the CPU to the GPU before the kernel function is called, and transfer back modified data from the GPU to the CPU after the kernel function returns. However, if a compiler can know that GPU global memory already has up-to-date data, they do not have to be copied again from the CPU. For this, we have developed an interprocedural data flow analysis that identifies *resident GPU variables*, which are the variables that reside in the GPU global memory and contain the same contents as the corresponding OpenMP *shared* variables in the CPU. The overall algorithm is shown in Figure 2.11. The algorithm recognizes if an OpenMP *shared* variable is used as a reduction variable in a kernel region and removes the variable from the resident GPU variable set (*gResidentGVars*). The rationale is that the translator implements reduction operations using a two-level tree reduction algorithm [11], where the final reduction is performed on the CPU; after the reduction operation finishes, only the CPU has the final reduction output. Moreover, if a read-only shared scalar variable is cached on the GPU shared memory for the current kernel execution, the variable is directly copied to the shared memory through kernel-argument passing, which does not use global memory. In this case, the variable is not added to the resident GPU variable set, since global memory may contain stale data.

We have developed another interprocedural data flow analysis to identify redundant memory transfers from the GPU to the CPU. We define a *live CPU variable* as the variable that resides in the CPU and may be potentially read before its next write. Even though a shared variable is modified by a kernel function, if it is not a live CPU variable at the exit of the kernel function, it does not have to be copied from the GPU to the CPU, since it will not be used by the CPU before it is modified again. We can not blindly apply a traditional live analysis, because the CUDA memory model

```
Live CPU variable analysis
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program annotated with user−directives containing shared variables
        that do not need to be transferred back to CPU
gLiveCPUVars_out(program exit node) = {}
for( node m : successor nodes of a node n )
    gLiveCPUVars_out(n) += gLiveCPUVars_in(m) //+ is a union operation
gLiveCPUVars_in(n) = gLiveCPUVars_out(n) − KILL(n) + GEN(n)
    KILL(n) = a set of modified shared variables //n can be a node in either a
                                                 //CPU region or a GPU region
    GEN(n) = a set of shared variables used in a node n //If n represents a node
                                                        //in a CPU region
```

Fig. 2.12. Interprocedural analysis to identify live CPU variables, which does not include a function-call-handling part and annotation generation part

has two separate address spaces, while a traditional live analysis assumes only one address space. The overall algorithm of the new, modified live analysis is shown in Figure 2.12. The main difference between the new live analysis and the traditional analysis is that in the new live analysis, the KILL set (*KILL(n)* in Figure 2.12) includes variables modified by either GPU or CPU, while the GEN set (*GEN(n)* in Figure 2.12) includes variables used only by CPU; even though the analysis works on an input OpenMP program, the compiler distinguishes code sections to be executed by CPU and those by GPU. Including variables modified by GPU in the KILL set has an important optimization effect; it can additionally reduce redundant memory transfers from GPU to CPU. For example, if two consecutive kernel regions modify the same shared variable, *a*, and the variable is used in the following CPU code section, the new live analysis will find that the variable *a* is live at the end of the second kernel region, but not at the end of the first kernel region, since the variable will be killed by the second kernel region while live variables are propagated backwardly during the analysis. Therefore, the translator will insert codes to transfer the variable

```
1  for(i = 0; i < 4096; i++)
2  {   a[i] = 0; b[i] = 0; c[i] = 0;   }
3  for(step = 0; step < 100; step++) {
4      #pragma omp parallel for private(i)
5      for(i = 0; i < 4096; i++)
6          c[i] = a[i] + b[i];
7      #pragma omp parallel for private(i)
8      for(i = 0; i < 4096; i++)
9          b[i] = a[i] + c[i];
10 }
11 for(i = 0; i < 4096; i++)
12     d[i] = a[i] + b[i];
```

(a) Input OpenMP code

```
1  ...
2  for(i = 0; i < 4096; i++)
3  {   a[i] = 0; b[i] = 0; c[i] = 0;   }
4  for(step = 0; step < 100; step++) {
5      gpuBytes = (4096 * sizeof (int));
6      cudaMemcpy(gpu__a, a, gpuBytes, cudaMemcpyHostToDevice);
7      gpuBytes = (4096 * sizeof (int));
8      cudaMemcpy(gpu__b, b, gpuBytes, cudaMemcpyHostToDevice);
9      kernel0 <<< dimGrid0, dimBlock0, 0, 0 >>> (gpu__a, gpu__b, gpu__c);
10     kernel1 <<< dimGrid1, dimBlock1, 0, 0 >>> (gpu__a, gpu__b, gpu__c);
11     gpuBytes = (4096 * sizeof (int));
12     cudaMemcpy(b, gpu__b, gpuBytes, cudaMemcpyDeviceToHost);
13 }
14 for(i = 0; i < 4096; i++)
15     d[i] = (a[i] + b[i]);
16 ...
```

(b) GPU code where redundant-memory-transfer optimizations are applied

Fig. 2.13. Motivation of memory transfer promotion optimization

a back to CPU only after the second kernel region, even though both kernel regions modified the same variable *a*.

The main focus of the analyses mentioned above is to identify redundant memory transfers between CPU and GPU; code generation for the redundant transfers will be skipped by the OpenMP-to-CUDA translator. For necessary memory transfer

calls, however, finding optimal points to insert the codes may also have a profound impact on program performance. For this, we have developed *Memory Transfer Promotion* optimization. The promotion optimization operates on a loop containing kernel regions. A motivating example for the promotion optimization is shown in Figure 2.13; Figure 2.13 (a) is a synthetic OpenMP code, where a serial loop contains two kernel regions, and Figure 2.13 (b) is the translated GPU code, where various redundant memory transfer optimizations including the ones described above are applied. In the output GPU code, no memory transfer calls were added before the second kernel call (*kernel1* in line 10 of Figure 2.13), since all the shared variables accessed in the second kernel are resident on the GPU global memory when the second kernel is called. After finishing kernel executions, only a shared variable b is copied back to CPU; a variable c is is not transferred back to CPU, since it is not a live CPU variable upon exit from the kernel-enclosing loop (line 13 in Figure 2.13 (b)), and copying of a variable a is also not necessary, since it is not modified by GPU. Remaining transfer calls (line 6, 8, and 12 in Figure 2.13 (b)) are required for correct program semantics. However, the CPU-to-GPU memory transfer calls in line 6 and 8 are needed only at the first iteration of the enclosing loop (line 4 in Figure 2.13), and the GPU-to-CPU memory transfer call in line 12 is needed only at the last iteration of the enclosing loop; memory transfers in all the other iterations are redundant. In this case, if we promote the CPU-to-GPU memory transfer calls before the enclosing loop and the GPU-to-CPU memory transfer call after the enclosing loop, the redundancy problems caused by the enclosing loop are solved without breaking program correctness, resulting in more optimized code. These promotions are safe if the enclosing loop contains only kernel regions. The promotion can go out of nested loops and above a procedure if the kernel-containing loop or procedure consists of kernel regions only.

An algorithm to identify optimal insertion points to which memory-transfer calls can be hoisted is shown in Figure 2.14, and an algorithm for corresponding promotion transformation is shown in Figure 2.15. In case where the promotion can go above a

```
Analysis algorithm for memory transfer promotion
Input: OpenMP program where both kernel splitting algorithm and selective
       procedure cloning transformation* are applied
Output: OpenMP program where kernel regions are annotated with information
        on optimal insertion points for memory transfer calls
for each kernel region KR in the input program
    enclosingLoopList = {}
    cs = KR
    containsKernelsOnly = true
    while ( containsKernelsOnly )
        cs = parent of cs
        if ( cs is a loop )
            if ( kernelLoopList does not contain cs )
                interprocedurally check whether cs contains only kernel regions
                if yes add cs into kernelLoopList
                else containsKernelsOnly = false
            if ( containsKernelsOnly is true )
                add cs into enclosingLoopList
        else if ( cs is a procedure )
            if ( kernelProcedureList does not contain p )
                interprocedurally check whether cs contains only kernel regions
                if yes add cs into kernelProcedureList
                else containsKernelsOnly = false
            if ( containsKernelsOnly is true )
                cs = function call statement of cs
        else
            containsKernelsOnly = false
    if ( enclosingLoopList is not empty )
        annotate KR with enclosingLoopList
```

Fig. 2.14. Memory transfer promotion analysis algorithm, which interprocedurally finds optimal insertion points for memory transfer calls (* is explained in Section 2.5.2)

procedure, we can either (1) hoist memory transfer calls up to the call graph or (2) leave the calls in the procedure but making them conditionally executed only at the first iteration or the last iteration of the iteration space of the enclosing loops. Our transformation algorithm uses the second approach, as shown in line 19 through 24 in Figure 2.15, since the first approach may require to move additional code, which may

```
Transformation algorithm for memory transfer promotion
Input: OpenMP program where memory transfer promotion analysis algorithm
       is applied
Output: CUDA program where memory transfer calls are promoted into
        optimal insertion points
1  for each kernel region KR in the input program
2      L = null
3      elList = enclosingLoopList of KR, which is inserted by memory
4                transfer promotion analysis
5      if ( elList is empty )
6          refPoint = KR
7      else
8          L = the outmost loop in elList
9          if ( both L and KR are in the same procedure )
10             refPoint = L
11         else
12             refPoint = KR
13     if( (refPoint is equal to KR) and (L is not null) )
14         for each shared variable s accessed in KR
15             if ( s is a local variable or value−passed function parameter )
16                 insert CPU−to−GPU memory transfer call before refPoint
17                 insert GPU−to−CPU memory transfer call after refPoint
18             else
19                 insert conditional CPU−to−GPU memory transfer call before
20                   refPoint so that data are transferred only at the first
21                   iteration of the iteration space of all loops in elList
22                 insert conditional GPU−to−CPU memory transfer call after
23                   refPoint so that data are transferred only at the last
24                   iteration of the iteration space of all loops in elList
25     else
26         for each shared variable s accessed in KR
27             insert CPU−to−GPU memory transfer call before refPoint
28             insert GPU−to−CPU memory transfer call after refPoint
```

Fig. 2.15. Memory transfer promotion transformation algorithm, which promotes required memory transfer calls into optimal insertion points based on the memory transfer promotion analysis output

not be allowed if some variables used in the additional code are not visible in the new scope. Even though a procedure contains only kernel regions, some memory transfer calls can not go above the procedure due to scope issue, if the data to transfer are

```
1   void comp(int a[4096], int b[4096], int c[4096], int m) {
2       int k;
3       #pragma omp parallel for private(k)
4       for(k = 0; k < 4096; k++)
5           b[k] = a[m] + c[k];
6   }
7   int main (int argc, char *argv[]) {
8       ...
9       for(step = LB; step < UB; step++)
10          for(i = 0; i < 1024; i++)
11              comp(a, b, c, i);
12      ...
13  }
```

(a) Input OpenMP code

```
1   void comp(int a[4096], int b[4096], int c[4096], int m, int step_Index0,
                    int step_LB0, int step_UB0) {
2       ...
3       gpuBytes=(4096*sizeof (int));
4       if (((step_Index0==step_LB0)&&(m==0)))
5           cudaMemcpy(gpu__a, a, gpuBytes, cudaMemcpyHostToDevice);
6       gpuBytes=(4096*sizeof (int));
7       if (((step_Index0==step_LB0)&&(m==0)))
8           cudaMemcpy(gpu__c, c, gpuBytes, cudaMemcpyHostToDevice);
9       gpuBytes=sizeof (int); cudaMalloc(((void **)(&gpu__m)), gpuBytes);
10      cudaMemcpy(gpu__m,  &m, gpuBytes, cudaMemcpyHostToDevice);
11      comp_kernel<<<dimGrid0, dimBlock0, 0, 0>>>(gpu__a, gpu__b, gpu__c, gpu__m);
12      gpuBytes=sizeof (int); cudaFree(gpu__m);
13      gpuBytes=(4096*sizeof (int));
14      if (((step_Index0==step_UB0)&&(m==1023)))
15          cudaMemcpy(b, gpu__b, gpuBytes, cudaMemcpyDeviceToHost);
16      return ;
17  }
18  int main(int argc, char * argv[]) {
19      ...
20      for(step = LB; step < UB; step++)
21          for(i = 0; i < 1024; i++)
22              comp(a, b, c, i, step, LB, (-1 + UB));
23      ...
24  }
```

(b) GPU code where memory-transfer-promotion optimization is applied

Fig. 2.16. Example translation of memory transfer promotion optimization

local to the procedure. An example translation is shown in Figure 2.16, where an input OpenMP code contains a kernel region residing in a procedure, called *comp*, and the procedure is called in a nested loops, as shown in Figure 2.16 (a). Because the procedure *comp* contains only one kernel region, and the nested loop calling the procedure does not have any other CPU code, memory transfer calls for the kernel region can be hoisted above the procedure boundary. The optimized code is shown in Figure 2.16 (b), where CPU-to-GPU memory transfers for shared variable a and c are executed only at the first iteration of the enclosing loop (line 3 through 8 in Figure 2.16 (b)), and GPU-to-CPU memory transfer for shared variable b is executed only at the last iteration of the enclosing loop (line 13 through 15 in Figure 2.16 (b)). However, memory transfers for shared variable m is not promoted (line 9 through 10 in Figure 2.16 (b)), since the variable is a value-passed function parameter, whose corresponding GPU memory will be allocated and deallocated locally in the enclosing procedure. Overhead reduction by the promotion optimization can be huge if the number of iterations of the enclosing loops is big.

The information obtained from these analyses is passed to the actual translator in the form of annotations, and the translator will perform necessary transformations depending on the passed information. Currently, the algorithm performs array-name-only analysis for shared array data. Depending on the optimization types, applying the array-name-only analysis may be unsafe. For this case, the compiler provides users with options to selectively apply the unsafe, aggressive optimizations.

2.5 Transformation Techniques Supporting OpenMP to CUDA Translation

This section explains transformation techniques that are used to address various issues arising during the OpenMP to CUDA translation.

```
1  #pragma omp parallel private(i, m)
   {
2      m = foo(n);
3      #pragma omp for
4      for(i = 0; i < 4096; i++)
5          b[m][i] = a[m][i];
6      #pragma omp for
7      for(i = 0; i < 2048; i++)
8          c[m][i] = (b[m][i] + b[m][4096 - i]) / 2;
   }
```

(a) Input OpenMP code

```
1  #pragma omp parallel shared(a, b, n) private(i, m)
   {
2      m = foo (n);
3      #pragma omp for nowait
4      for(i = 0; i < 4096; i++)
5          b[m][i] = a[m][i];
   }
6  #pragma omp parallel shared(b, c) private(i, m)
   {
7      #pragma omp for nowait
8      for(i = 0; i < 2048; i++)
9          c[m][i] = (b[m][i] + b[m][4096 - i]) / 2;
   }
```

(b) OpenMP code after kernel splitting is applied

```
1  m=foo(n);
2  #pragma omp parallel shared(a, b) private(i) firstprivate(m)
   {
3      #pragma omp for nowait
4      for(i = 0; i < 4096; i++)
5          b[m][i] = a[m][i];
   }
6  #pragma omp parallel shared(b, c) private(i) firstprivate(m)
   {
7     #pragma omp for nowait
8     for(i = 0; i < 2048; i++)
9          c[m][i] = (b[m][i] + b[m][4096 - i]) / 2;
   }
```

(c) OpenMP code after UEP variable removal transformation is applied

Fig. 2.17. Motivation of upwardly-exposed private variable removal transformation

2.5.1 Upwardly-Exposed Private Variable Removal

In Section 2.3.2, we have described an algorithm to identify kernel regions, which involves parallel region splitting to preserve a global synchronization under the CUDA programming model. However, this splitting may incur an *upwardly exposed private variable problem (UEP problem)*. A variable is upwardly exposed if the variable is used before it is written (defined) in a code structure such as a loop. In the OpenMP memory model, the value of a private variable is not defined upon entry to a parallel region unless the variable is in a *firstprivate* clause, and the value is also not defined upon exit from the region if the variable does not appear in a *lastprivate* clause. Therefore, private variables without firstprivate clauses should be defined first in a parallel region before they are used. A UEP problem occurs if a private variable is used in a parallel region before it is defined first. Figure 2.17 shows an example of the case. Figure 2.17 (a) is an input OpenMP code, where a parallel region contains two omp-for loops, and Figure 2.17 (b) shows the output OpenMP code after kernel splitting algorithm is applied. The kernel splitting algorithm splits the original parallel region into two sub-regions, since omp-for loops without *nowait* clauses have implicit barriers at the end of the loops. The second sub-region in Figure 2.17 (b) has a UEP problem since a private variable m is used without any definition. To fix this problem, the definition statement of the private variable m (line 2 in Figure 2.17 (b)) should be moved out of the enclosing sub-region, and the private variable should be changed to a firstprivate variable in both sub-regions. Resulting code is shown in Figure 2.17 (c).

A transformation algorithm to remove UEP problems is shown in Figure 2.18. To find UEP variables, a traditional live analysis is applied; live-in private variables upon entry of a kernel region are the UEP variables for the kernel region. However, some private variables may become upwardly exposed after the kernel region is transformed into a kernel function, if they are used in the control part (initial statement, condition expression, or step expression) of any omp-for loop in the kernel region.

```
Algorithm to remove upwardly−exposed private variables
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program where upwardly−exposed private variables are removed
1  for each procedure p containing kernel regions
2      perform live analysis on p
3      for each kernel region KR in p
4          for each private variable s, which is live upon entry to KR
5              if ( s is not a reduction variable )
6                  add s to UEPSymSet
7          for each omp−for loop L in KR
8              for each private variable s used in L
9                  if ( s is not a index variable of L )
10                     if ( s is used in initial statement, condition expression,
11                          or step expression of L )
12                         add s to UEPSymSet
13     for each private variable s in UEPSymSet
14         convType = 0
15         for each kernel region KR in p
16             if ( s is written in KR )
17                 if ( s is upwardly−exposed in KR )
18                     convType = −3; break
19                 else if ( s is written in any loop in KR )
20                     if( (s is not used as index variable) and
21                         (s is not used as reduction variable) )
22                         convType = −1; break
23                 else if ( s is written in a simple statement SS in KR )
24                     add SS into removeStmtSet
25                 else convType = −2; break
26         if ( convType < 0 ) continue
27         for each statement SS in removeStmtSet
28             move SS before the enclosing kernel region
29         for each kernel region KR where s is upwardly−exposed
30             remove s from OpenMP private clause
31             add s to OpenMP firstprivate clause
```

Fig. 2.18. Transformation algorithm to remove upwardly exposed private variable problems

As explained in Section 2.3.2, the translator decides the *thread batching* of a kernel region by calculating the maximum partition size among parallel work contained the kernel region, and resulting code for the thread batching is inserted somewhere before the kernel function call site. (Actual insertion point can be changed depending on related optimizations.) Therefore, private variables that are used in the control part of any omp-for loop will be included in the thread batching code for the kernel region, and depending on the insertion location of the code, the private variables may become upwardly exposed. Current algorithm conservatively include these variables as potential UEP variables, as shown between line 7 and 12 in Figure 2.18. The next step is to find statements where the UEP variables are defined (*DEF statements*) and move them before their enclosing kernel regions, so that the DEF statements are reachable to kernel regions with UEP problems. However, we may not be able to move all DEF statements; all variables used in a DEF statement should be visible in the lexically enclosing block, the value of the UEP variable should be thread-independent, and moving the DEF statement out of the enclosing region should not change the program semantics. If all of these conditions are met, we can safely move the DEF statement out of the enclosing parallel region. To check these conditions, the algorithm performs three tests:

- Check whether a kernel region containing a DEF statement has a UEP problem related to the DEF statement (lines from 17 to 18 in Figure 2.18). If so, moving the DEF statement out of the kernel region may change the program semantics.

- Check whether a DEF statement resides in any of loops in the kernel region, and the corresponding UEP variable is neither used as index variable nor used as reduction variable (lines from 19 to 22 in Figure 2.18). If so, the UEP variable is likely to have thread-dependent value, not suitable for DEF statement movement.

- Check whether a DEF statement is a simple assignment statement without conditions (lines from 23 to 24 in Figure 2.18). If not, moving the DEF statement may alter the program semantics.

If all of these tests are passed, the algorithm will perform code transformation; move DEF statements out of the enclosing parallel regions and change UEP variables from OpenMP *private* variables to *firstprivate* variables (lines from 27 to 31 in Figure 2.18). However, the safety tests conducted in the algorithm are not conservative enough to guarantee the correctness. Therefore, the compiler provides multiple option levels so that a user can choose an appropriate transformation level under the user's approval.

2.5.2 Selective Procedure Cloning

Selective procedure cloning transformation selectively clones procedures if a procedure contains kernel regions or if kernel regions exist in any of functions called from the procedure (possibly indirectly). This transformation is performed as a preprocessing step for context-sensitive, interprocedural analyses, such as the ones described in Section 2.4.2. This preprocessing allows that every kernel function can be called in a unique calling context, making it easy to perform interprocedural analyses on kernel functions. This selective cloning may be necessary for a context-sensitive analysis if different outputs of the analysis result in different translations. Without cloning, analyses should be conservative and thus may lose possible optimization opportunities. Motivating example is shown in Figure 2.19; Figure 2.19 (a) is the input OpenMP code, where the same procedure, *foo*, which contains a parallel region (kernel region), is called twice in different calling contexts. If we apply *resident GPU variable analysis* technique described in Section 2.4.2, the analysis will find that shared variable *a*, *b*, and *c* are resident on the GPU global memory upon entry to the parallel region in the second function call of *foo* (line 8 in Figure 2.19 (a)), while there is no resident GPU variable upon entry to the parallel region in the first call of *foo* (line 7 in Figure 2.19 (a)). The analysis results say that we need to insert codes to transfer the

```
1  void foo(int out[4096], int in1[4096], int in2[4096]) {
2      int i;
3      #pragma omp parallel for private(i)
4      for(i = 0; i < 4096; i++)
5         out[i] = in1[i] + in2[i];
   }
6  int main (int argc, char *argv[]) {
7      foo(c, a, b);
8      foo(d, a, c);
9      return 0;
   }
```

(a) Input OpenMP code

```
1  void foo(int out[4096], int in1[4096], int in2[4096]) {
2      int i;
3      #pragma omp parallel for private(i)
4      for(i = 0; i < 4096; i++)
5          out[i] = in1[i] + in2[i];
}
6  void foo_clnd1(int out[4096], int in1[4096], int in2[4096]) {
7      int i;
8      #pragma omp parallel for private(i)
9      for(i = 0; i < 4096; i++)
10         out[i] = in1[i] + in2[i];
   }
11 int main (int argc, char *argv[]) {
12     foo(c, a, b);
13     foo_clnd1(d, a, c);
14     return 0;
   }
```

(b) OpenMP code after selective cloning is applied

Fig. 2.19. Motivation of selective cloning of procedures containing kernel regions

data of shared variable a and c from CPU to GPU when the kernel function, which corresponds to the parallel region, is called the first time, but the data copying of a and c is not necessary for the second call of the same kernel function. Without cloning, however, the translator should conservatively insert memory transfer codes for a and c before the call of the kernel function, since the kernel function is called in

the same procedure, *foo*, in both contexts. If the selective procedure cloning transformation is applied, the procedure *foo* will be cloned, since it contains a kernel region. Figure 2.19 (b) shows the output OpenMP code when the selective cloning is applied. As a result of the cloning, the kernel region shown in line 3 through 5 in Figure 2.19 (a) is also cloned: one is shown in line 3 through 5, and the other is shown between line 8 and 10 in Figure 2.19 (b). If the new OpenMP code is translated into CUDA code, each kernel function will be called in a unique context, and thus the translator can insert memory transfer codes selectively, according to the output of the resident GPU variable analysis, generating more efficient output CUDA code.

The core algorithm of the selective procedure cloning transformation is explained in Figure 2.20; first, the algorithm creates a list of procedures, where callee procedures come before their caller procedures, which can be done by traversing the *call graph*, which is a directed graph that represents calling relationships between procedures in a program, in post-order traversal. Second, the algorithm finds the procedures that contain kernel regions and their caller procedures including indirect callers. These procedures are candidates for cloning; we include caller procedures, since if they are called multiple times, it will cause that kernel regions in the callee procedures are called in different contexts. Third, the algorithm clones procedures in the candidate list if they are called more than once. In this step, cloning should be performed on the procedures that are higher in call graph first, since cloning of a caller procedure will result in more calling of its callee procedures.

An alternative to this selective cloning transformation is to use a selective inlining transformation. However, inlining transformation is much more complex than the cloning method, and it may lose some information during the inlining and may incur more aliasing issues. Therefore, our translator uses cloning instead of inlining.

```
Selective procedure cloning algorithm
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program where procedures are cloned so that each procedure
        containing kernel regions is called in a unique context
1  procedureList = a list of procedures in ascending order of call graph
2  for each procedure p in procedureList
3      if ( p contains kernel regions )
4          add p into callingProcList
5      while ( callingProcList is not empty )
6          c_p = callingProcList.removeFirst()
7          for each procedure p_p which calls c_p
8              if ( (p_p is not a main procedure) and
9                   (p_p is not in visitedProcList) )
10                  add p_p into callingProcList
11                  add p_p into visitedProcList
12         if ((c_p is called more than once) and (c_p is not in clonedProcList))
13             add c_p into clonedProcList in the order of procedureList
14         if ((c_p is called only once) and (c_p is not in mayClonedProcList))
15             add c_p into mayClonedProcList in the order of procedureList
16 while ( (clonedProcList is not empty) or (mayClonedProcList is not empty) )
17     if ( clonedProcList is not empty )
18         c_p = clonedProcList.removeLast()
19     else if ( mayClonedProcList is not empty )
20         c_p = mayClonedProcList.removeLast()
21     if ( c_p is called more than once )
22         for each function call fcall to c_p
23             clone c_p and replace fcall with a new call to the cloned procedure
```

Fig. 2.20. Procedure cloning algorithm which clones procedures containing kernel regions interprocedurally

2.5.3 Converting Pointer Variables to Array Variables

As explained in Section 2.3.2, when the translator transforms a kernel region into a kernel function, the translator generates codes for allocating memory on the GPU memory space and necessary memory transfer calls for the data accessed by the kernel function. For correct GPU memory allocation and memory transfer, the compiler should be able to detect the data size of OpenMP *shared* or *threadprivate* variables. If the variables are either scalar or array type, finding the data size is

```
1  void comp(float **a) {
2      int i, m;
3      #pragma omp parallel for shared(a) private(i, m)
4      for(i = 0; i < SIZE1; i++)
5          for(m = 0; m < SIZE2; m++)
6              a[i][m] = i + m;
7  }
8  int main (int argc, char *argv[]) {
9      int i;
10     float **a;
11     a = (float **)malloc(SIZE1 * sizeof(float *));
12     for(i = 0; i < SIZE1; i++)
13         a[i] = (float *)malloc(SIZE2 * sizeof(float));
14     ...
15     comp(a);
16     ...
17 }
```

(a) Input OpenMP code

```
1  void comp(float a[SIZE1][SIZE2]) {
2      int i, m;
3      #pragma omp parallel for shared(a) private(i, m)
4      for (i = 0; i < SIZE1; i++)
5          for (m = 0; m < SIZE2; m++)
6              a[i][m] = i + m;
7  }
8  int main (int argc, char *argv[]) {
9      float (*a)[SIZE2];
10     a = (float (*)[SIZE2])malloc(SIZE1 * SIZE2 * sizeof(float));
11     ...
12     comp(a);
13     ...
14 }
```

(b) Modified OpenMP code where a pointer variable is converted into an array variable

Fig. 2.21. Example conversion of a pointer-type variable into an array-type variable

trivial. However, if the variables are pointer-type, it may require a complex pointer analysis to detect the size. An example case is shown in Figure 2.21 (a). In the figure, even though an OpenMP *shared* variable a is accessed using an array-access

expression in a parallel region (line 6), *a* is a pointer variable, pointing to a two-dimensional data structure, where each row is separately allocated, as shown in line 10 through 13. In this case, for the translator to insert correct memory transfer calls, an advanced analysis with complex pointer chasing will be needed. Due to the lack of the advanced pointer analysis, the current translation system can not transform some kernel regions, if they contain pointer-type *shared/threadprivate* variables. However, if a pointer variable points to an array structure, it can be converted to an array variable through parameter passing; converting a pointer variable pointing to a one-dimensional data structure is trivial. Converting a pointer variable, which points to multi-dimensional data structure, can be tricky depending on the way it is allocated. An example conversion of the two-dimensional array case in Figure 2.21 (a) is shown in Figure 2.21 (b). In the new code, the shared variable *a* is pointer-to-array type (line 9 in Figure 2.21 (b)), and the whole data is allocated as a linear data as shown in line 10. When the variable is passed into a function called *comp*, the type is changed to array type, since the corresponding formal parameter is declared as two-dimensional array type (line 1). For now, an automatic type conversion is not supported by the translator, and thus programmers should convert types manually. Providing an automatic conversion or an advanced pointer analysis technique is a future work.

2.6 Evaluation

This section discusses the performance of the proposed OpenMP to GPGPU translator and compiler optimizations. In our experiments, fourteen OpenMP programs (two kernel benchmarks (*JACOBI* and *SPMUL*), three NAS OpenMP Parallel Benchmarks (*EP*, *CG*, and *FT*), and nine Rodinia Benchmarks [9] (*BACKPROP*, *BFS*, *CFD*, *HEARTWALL*, *SRAD*, *HOTSPOT*, *KMEANS*, *LUD*, and *NW*) were transformed by the translator. The tested programs include both regular and irregular applications in diverse domains such as Medical Imaging, Bio-informatics, Fluid Dynamics, Linear Algebra, Data Mining, Graph Algorithms, and so on. For performance

comparison, the hand-tuned CUDA versions of the tested programs were also evaluated; Rodinia Benchmark Suite provides both OpenMP versions and CUDA versions, and thus the corresponding CUDA versions are used for the tested Rodinia Benchmark programs, and the manual version of FT benchmark is from Hpcgpu Project [12]. For *JACOBI*, *SPMUL*, *EP*, and *CG* programs, we created hand-tuned CUDA versions from the OpenMP versions of the programs. (The Hpcgpu Project also has CUDA versions for *EP* and *CG*, but our manual versions perform better than those, and thus we used in-house versions.)

We used an NVIDIA Quadro FX 5600 GPU as an experimental platform. The device has 16 multiprocessors with clock rate at 1.35 GHz and has 1.5GB DRAM. Each multiprocessor is equipped with 8 SIMD processing units, amounting to 128 processing units in total. The device was connected to a host system consisting of two Dual-Core AMD 3 GHz Opteron processors. Because the tested GPU does not support double precision, we manually converted the OpenMP source programs into single precision programs before they are fed to our translator. (NVIDIA recently announced GPUs supporting double precision computations.) In addition, the current translator can not translate kernel regions containing pointer-type *shared/threadprivate* variables, as explained in Section 2.5.3. Therefore, we manually converted pointer variables to array variables using the conversion method explained in Section 2.5.3, if existing in kernel regions. We compiled the translated CUDA programs with the NVIDIA CUDA Compiler (NVCC) to generate device codes, and compiled the host programs with the *GCC* compiler version 4.2.3, using option *-O3*.

Table 4.9 presents the overall performance of the proposed OpenMP-to-GPU translation and optimization system when the largest available input data were used; in the table, *Modified OpenMP* refers to the performance when the original, CPU-oriented OpenMP programs were manually transformed to GPU-oriented OpenMP programs before they are fed to the translator. The table shows that there are very large performance variations; the low minimum speedups are because some programs have small amount of computations even with the largest available input, and thus

GPU memory allocation and data transfers between CPU and GPU dominate the execution of the translated GPU versions. The following sections present the results in detail.

Table 2.2
Overall performance of the proposed OpenMP-to-GPGPU translation and optimization system, when the largest available input data were used. *Translator Input* refers to the types of the translator input sources; *Modified OpenMP* means that the input OpenMP code is manually modified before fed to the translator.

Translator Input	Speedup over Serial on CPU			Speedup over Unoptimized Versions		
	MIN	MAX	AVG	MIN	MAX	AVG
Original OpenMP	0.22	245	3.73	1	67.7	3.6
Modified OpenMP	0.79	245	7.41	1.2	64.3	7.14

2.6.1 Performance of JACOBI

A *JACOBI* kernel is a widely used code containing the main loop of an iterative solver method in regular scientific applications. Due to its simple structure, the *JACOBI* kernel is easily parallelized in many parallel programming models. However, the base-translated GPU code does not perform well, as shown in Figure 2.22; *OrgBase* in the figure represents the speedups of the unoptimized GPU version over serial on the CPU. This performance degradation is mostly due to the overhead in large, uncoalesced global memory access patterns. These uncoalesced access patterns can be changed to coalesced ones by applying *parallel loop-swap* transformation (P in Figure 2.22). These results demonstrate that, in regular programs, uncoalesced global memory accesses may be converted to coalesced accesses by basic loop transformations. *Pitched Malloc* optimization (T in the figure) uses a special CUDA runtime library (*cudaMallocPitch()*) to allocate two-dimensional arrays, instead of using a default CUDA memory allocation method (*cudaMalloc()*). Using *cudaMallocPitch()* makes sure that the allocation is appropriately padded to meet the alignment re-

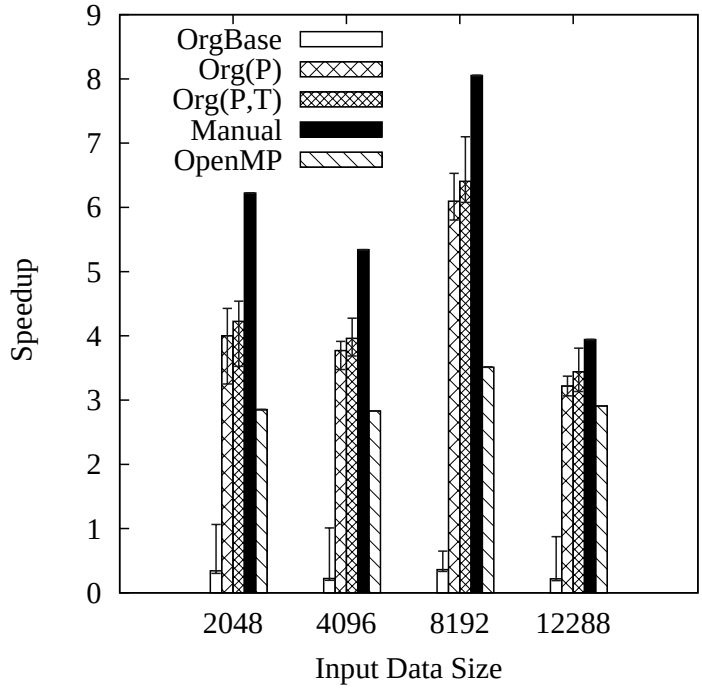


Fig. 2.22. Performance of JACOBI (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *P* means *Parallel Loop-Swap*, and *T* refers to *Pitched Malloc* optimization. The bars in each box show the performance variations caused by different thread batchings. The performance irregularity is caused by reproducible, inconsistent behavior of the CPU execution; the average CPU execution time per array element is longer at N = 8192 than others.

quirements for coalesced GPU global memory accesses, therefore ensuring best performance when accessing the row addresses of the two-dimensional arrays, as shown in Figure 2.22. However, *Pitched Malloc* optimization is not always applicable; if the size of the two-dimensional array is too big, the runtime library can not handle it (e.g. *HEARTWALL*), and if array access patterns are very complex, our translator may not be able to change codes correctly (e.g. *KMEANS*). The manual versions (*Manual*) use tiling transformations to exploit shared memory, which is not yet supported

by the current translator. We attribute the performance difference between versions generated by hand and by our translator primarily to this reason.

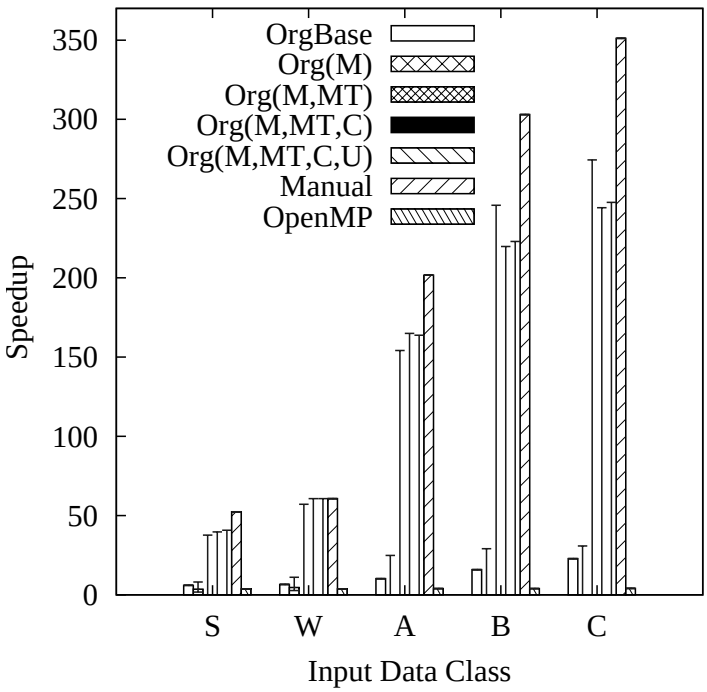


Fig. 2.23. Performance of EP (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *Org-Base* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *M* means redundant memory transfer optimizations, *MT* refers to *Matrix Transpose*, *C* shows caching optimizations, and *U* is *Loop Unrolling on Reduction* optimization. The bars in each box show the performance variations caused by different thread batchings.

2.6.2 Performance of EP

EP is one of the NPB3.0-OMP, NAS OpenMP Parallel Benchmarks. *EP* has abundant parallelism with minimal communication; it is often used to demonstrate the performance of parallel computing systems. However, the baseline translation of *EP* shows surprisingly low speedups on the tested GPU (*OrgBase* in Figure 2.23).

At the core of *EP* is a random number generator; each participating thread or process performs some computations based on the chosen numbers. In an OpenMP version, each thread stores the random numbers in a *threadprivate* array. The baseline translation scheme allocates the memory space for the *threadprivate* array by expanding the array in a row-wise fashion. However, this row-wise array expansion causes uncoalesced memory access problems under the CUDA memory model.

The *matrix transpose* transformation in Section 2.4.2 resolves this limitation; it changes the access patterns of the *threadprivate* arrays into coalesced accesses. *MT* in Figure 2.23 shows the speedups when the *matrix transpose* transformation is applied. The results show that *matrix transpose* increases the performance tremendously. To increase the performance of the translated *EP* further, additional CUDA-specific optimizations are applied; if the size of a private array, which is normally mapped to *local memory* in the GPU device, is small, it can be allocated on *shared memory* by using array expansion; *C* in the figure includes this optimization. Also, *loop unrolling* can be applied to reduction computations (*U* in the figure). The bars in each box in Figure 2.23 represents the performance variations due to different *thread batchings*. The figure shows that the performance variations are quite huge and the corresponding average speedups are not visible in most cases, which reveal that some thread batchings are applicable only to a small set of cases where input data size is small and caching optimizations are not applied. The huge performance variations indicate that complex interactions among the hardware resources may need fine-grained tuning for optimal performance.

The performance gap between the automatically translated version (*Org(M,MT,C,U)* in the figure) and the manual version (*Manual*) is due to the difference in handling a *critical* section; *EP* uses an OpenMP *critical* construct to implement an array reduction under the OpenMP programming model. Both hand-written and system-translated versions transform the critical section into an array reduction code, but the manual version optimizes further by removing a redundant private array, which was used as a local reduction variable. Improved array section analysis would be able

to detect this redundancy. *EP* is a representative case showing the importance of compile-time transformations.

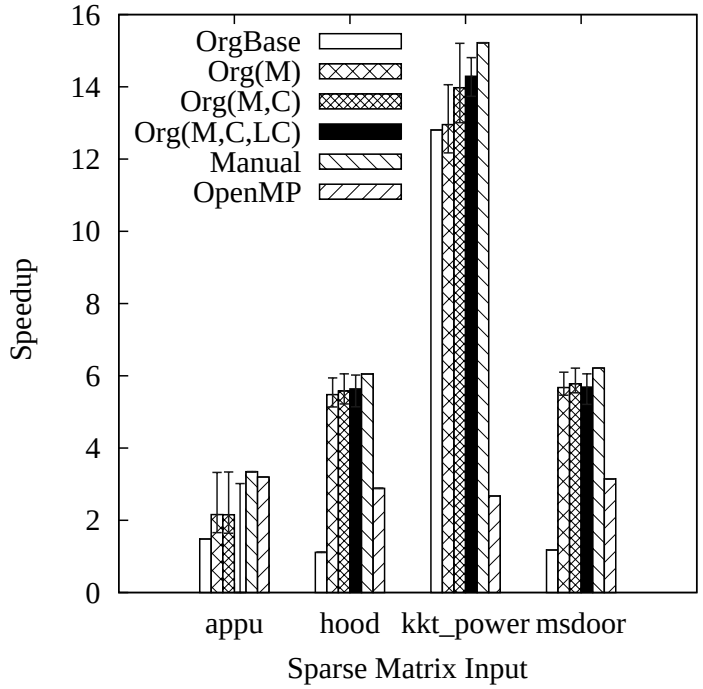


Fig. 2.24. Performance of SPMUL (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *M* means redundant memory transfer optimizations, *C* shows caching optimizations, and *LC* refers to *Loop Collapsing* optimization. The bars in each box show the performance variations caused by different thread batchings.

2.6.3 Performance of SPMUL

Sparse matrix-vector (SpMV) multiplication is an important representative of irregular applications. In our experiments, we have translated two codes using SpMV, an actual *SPMUL* kernel and the NAS OpenMP Parallel Benchmark *CG*, to measure the performance of the proposed system on irregular applications.

In *SPMUL*, a baseline translation with redundant memory transfer optimizations gives reasonable speedups (*Org(M)* in Figure 2.24) on several real sparse matrices in the UF Sparse Matrix Collection [13], even though *SPMUL* is an irregular application. Simple *local caching* optimizations, such as caching frequently accessed global data on registers and using *texture memory* to exploit a dedicated cache for read-only global data, work well (*C* in Figure 2.24), since some data in the *global memory* are accessed repeatedly either within each thread or across threads.

From an algorithmic perspective, the GPU-translated *SPMUL* conducts SpMV multiplications by assigning each row to each thread and letting each thread compute the inner product for the assigned row, which takes a reduction form. This algorithmic structure is implemented as a nested loop, which exhibits blocked access patterns, where each thread accesses a block of continuous data in *global memory*. As mentioned in [7], blocked access results in significantly lower memory bandwidth than cyclic access, due to uncoalesced memory access. The *loop collapsing* technique, explained in Section 2.4.1, can change the blocked access patterns to cyclic access patterns, increasing the effective bandwidth on *global memory*. However, the performance improvement due to the *loop collapsing* optimization is not very visible in *SPMUL* case (*LC* in Figure 2.24), while it is beneficial in *CG* case, as shown in the next section.

2.6.4 Performance of CG

CG is another sparse linear solver program. While both *CG* and *SPMUL* use similar SpMV multiplication algorithms, *CG* poses additional challenges. In *CG*, each parallel region contains many work-sharing regions including *omp single*. Depending on kernel-region-extracting strategies, the amount of overheads related to kernel invocations and memory transfers will be changed. The kernel-region-identifying algorithm described in Section 2.3.2 merges as many work-sharing regions as possible to minimize the kernel-related overheads. As indicated in Figure 2.25, however, base

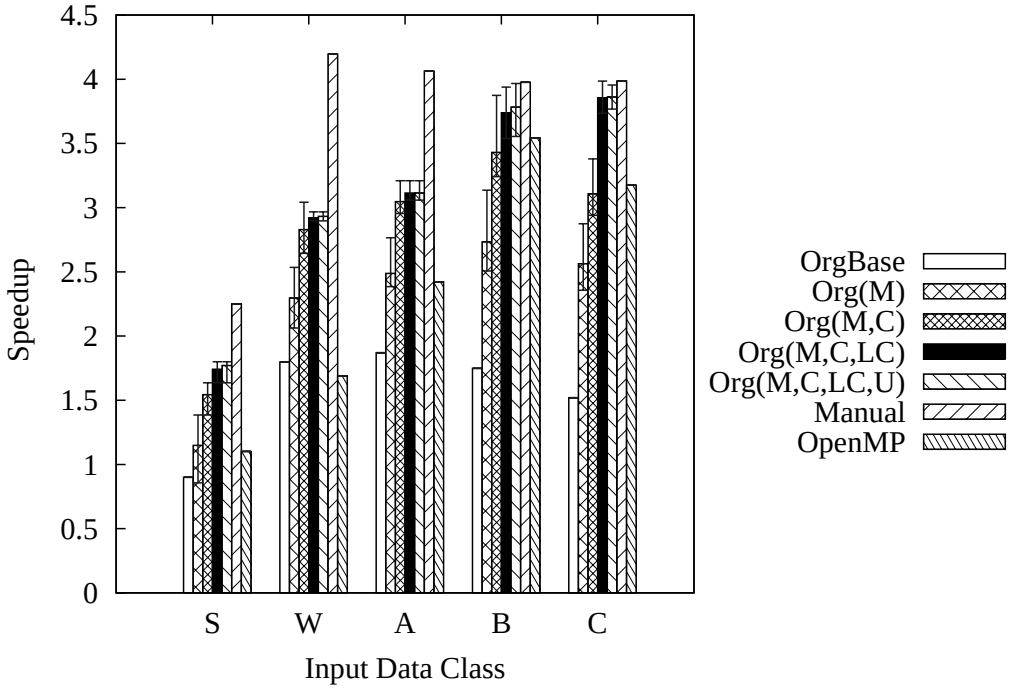


Fig. 2.25. Performance of CG (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *M* means redundant memory transfer optimizations, *C* shows caching optimizations, *LC* refers to *Loop Collapsing*, and *U* is *Loop Unrolling on Reduction* optimization. The bars in each box show the performance variations caused by different thread batchings.

GPU version (*OrgBase*) still incurs very large memory transfer overheads, degrading the performance. Eliminating redundant memory transfers with the algorithm described in Section 2.4.2 can reduce these overheads. In *CG*, an interprocedural data flow analysis is needed to identify this redundancy, since work-sharing regions are distributed among several subroutines. *M* in Figure 2.25 represents the case where the redundant memory transfers are minimized. The results show that eliminating redundant data transfers can increase the performance, and *loop collapsing* transformation works well on *CG* (*LC* in Figure 2.25).

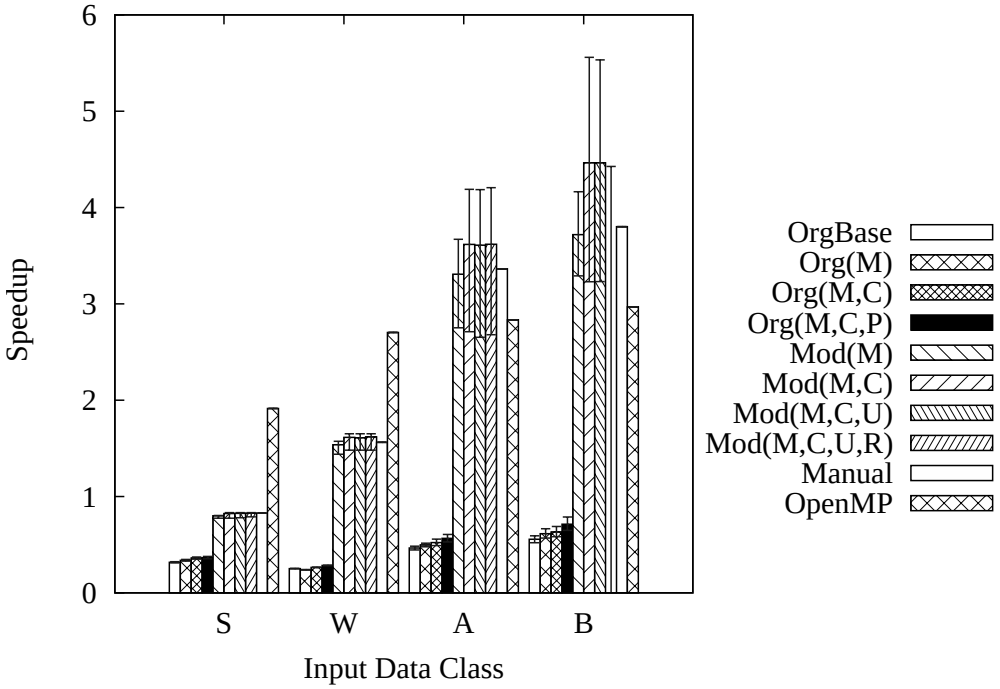


Fig. 2.26. Performance of FT (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *Org-Base* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, *P* refers to *Parallel Loop-Swap*, *U* is *Loop Unrolling on Reduction*, and *R* means caching of in-block reduction data on the shared memory. The bars in each box show the performance variations caused by different thread batchings.

2.6.5 Performance of FT

The NAS OpenMP Parallel Benchmark *FT* solves a 3-D partial differential equation using the Fast Fourier Transform (FFT), which is used in many different fields such as physics, applied mathematics, cryptography, and computational finance. The original OpenMP version heavily uses data blocking to exploit intra-thread locality

on traditional, cache-based systems. Moreover, for the best intra-thread locality, nested loops performing FFT are parallelized across 3rd or 2nd dimensions, instead of 1st dimension. However, this data partitioning scheme allows little opportunity for coalesced global memory accesses. Therefore, the automatically translated versions (*Org* in Figure 2.26) perform very poorly even with various optimizations, such as local caching optimizations (*C* in the figure) and *parallel loop-swap* optimization (*P* in the figure). The hand-written CUDA version (*Manual* in the figure) applies different data partitioning scheme; it transposes the whole 3-D matrix so that 1st dimension is always parallelized for all 1-D FFT computations. It also linearizes all 2-D and 3-D arrays to reduce memory-address-computation overhead. Figure 2.26 shows the the manual CUDA version (*Manual*) performs reasonably as the input data size increases. These changes may not be easy to be performed automatically by the translator due to their complexity. However, we can express all these changes in the OpenMP source code. *Mod* in the figure shows the performance when we manually modify the data partitioning scheme of the original OpenMP program according to the one used in the manual version, and then the compiler translates and optimizes the modified OpenMP code. Because our translator applies various caching optimizations more aggressively, the new, translated versions (*Mod(M,C,U,R)*) performs better than the manual versions in most cases. The *FT* case demonstrates that our OpenMP-to-CUDA translation system can be useful even for advanced programmers who know well CUDA memory and execution models, since they can apply their knowledge directly on the input OpenMP codes without dealing with the complex CUDA programming syntax.

2.6.6 Performance of BACKPROP

Back Propagation (*BACKPROP*) is one of Rodinia Benchmarks. *BACKPROP* is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The program consists of two phases: the forward phase and

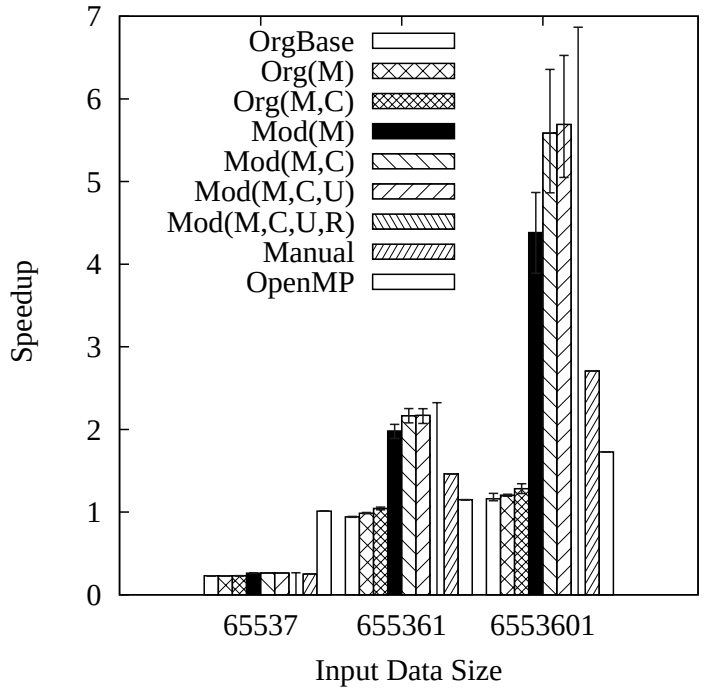


Fig. 2.27. Performance of BACKPROP (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, *U* is *Loop Unrolling on Reduction*, and *R* means caching of in-block reduction data on the shared memory. The bars in each box show the performance variations caused by different thread batchings.

the backward phase, and in the both phases, the processing of all the nodes is done in parallel. Figure 2.27 shows that the automatically translated versions (*Org*) perform very poorly; the main performance bottleneck is due to uncoalesced accesses of two-dimensional weight arrays, which are read in the forward phase and updated in the backward phase. In fact, the *parallel loop-swap* technique proposed in Section 2.4.1 can fix the uncoalesced global memory problems. However, the current compiler does

not perform the *parallel loop-swap* transformation automatically; the current compiler analysis can not detect eligible loops either because they are not perfectly nested loops or because conservative dependence analysis fails to detect parallelism. *Mod* in the figure shows the performance when we applied *parallel loop-swap* transformation manually.

The manual version (*Manual* in the figure) applies different techniques; first it linearizes the two-dimensional weight arrays and then applies complex tiling transformations to exploit GPU shared memory. However, results in Figure 2.27 reveal that the compiler-translated versions with *parallel loop-swap* optimization (*Mod*) performs much better than the manual versions (*Manual*). Generally, tiling optimization using GPU shared memory works well, since it can exploit both intra-thread and inter-thread locality. In *BACKPROP* case, however, intra-thread locality does not exist in accessing the weight arrays, and thus exploiting shared memory allows only the benefit of coalesced accesses, which is the same as the one that *parallel loop-swap* gives. Contrary to *parallel loop-swap*, tiling using shared memory requires additional synchronizations, leading to less overall performance gain than the simple loop transformation technique (*parallel loop-swap*). *BACKPROP* is another example suggesting that careful selection of optimizations is needed to achieve the best performance on the GPU.

2.6.7 Performance of BFS

Breadth-First Search (*BFS*) is one of fundamental graph algorithms widely used in many scientific and engineering applications. Even though it has a very simple algorithm, its irregular access patterns using a subscript array make it difficult to achieve performance on the GPU. Therefore, results in Figure 2.28 show that OpenMP versions (*OpenMP*) perform much better than any GPU versions.

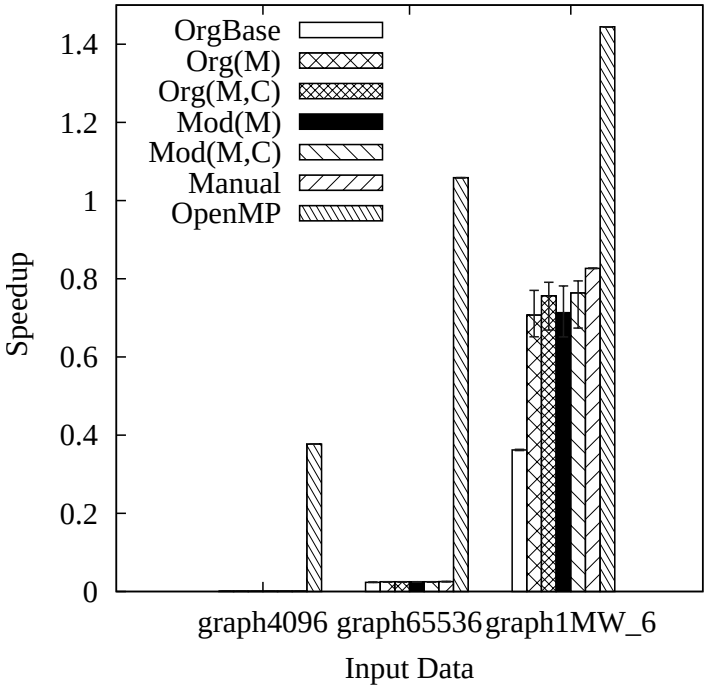


Fig. 2.28. Performance of BFS (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *Org-Base* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, and *C* shows caching optimizations. The bars in each box show the performance variations caused by different thread batchings.

2.6.8 Performance of CFD

CFD is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. Figure 2.29 shows that the automatic versions (*Org*) have some speedups but much less then the manual versions (*Manual*). The performance gap is largely due to the uncoalesced global memory accesses. *CFD* uses several two-dimensional matrix data, and accessing these matrices causes unco-

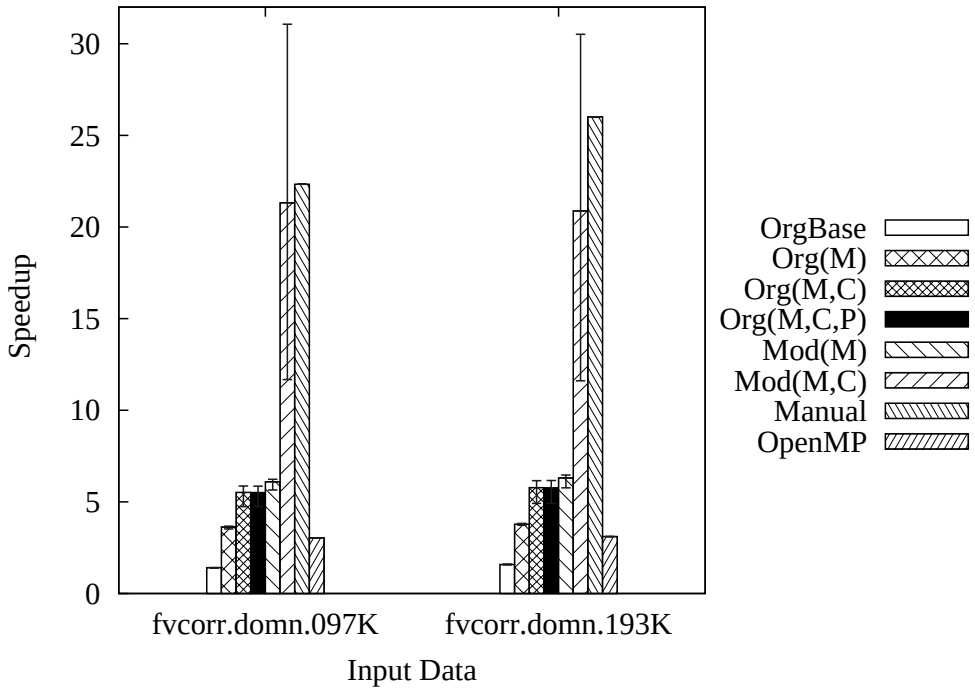


Fig. 2.29. Performance of CFD (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, and *P* refers to *Parallel Loop-Swap* optimization. The bars in each box show the performance variations caused by different thread batchings.

alesced memory access problems. However, the compiler could not recognize these two-dimensional data access patterns as candidates for *parallel loop-swap* transformation, since they are stored in one-dimensional arrays and accessed using complex subscript expressions. The manual version modifies the layout of these two-dimensional data when they are stored into one-dimensional arrays and changes the corresponding array subscript expressions, such that adjacent threads can access adjacent data in the memory, resulting in coalesced memory accesses. These changes can be expressed

in the input OpenMP codes, and *Mod* in the figure represents the performance when these modifications are applied to the input OpenMP program before the program is translated. Our modified, translated versions (*Mod*) perform better than the manual versions, since the translator applies more caching optimizations than the manual versions. For example, the manual versions allocate some data on the GPU constant memory to exploit the constant cache, if they are never modified by the GPU. However, our translator uses more fine-grained approach to exploit both constant cache and texture cache; if shared variables are never written by the GPU, the translator put them on the constant memory, and if shared variables are read-only in some kernel functions, but not all, the translator allocates them on the global memory, and let only read-only kernels access the data using the built-in texture fetching functions, which internally exploit the texture cache. The large performance variations in the figure also reveal that *thread batching* plays an important role in achieving the optimal performance.

2.6.9 Performance of HEARTWALL

The Rodinia Benchmark *HEARTWALL* program tracks the movement of a mouse heart over a sequence of ultrasound images to record response to the stimulus. In fact, both *HEARTWALL* and *SRAD*, which is described in the next section, are two stages of an application called *Heart Wall*. However, the Rodinia Benchmark Suite includes these two stages as two separate kernels, since they have distinct types of workloads.

The tracking code, which is a main part of the *HEARTWALL* kernel, is implemented in the form of multiple nested loops that process batches of 10 frames and 51 points in each image. There is a sequential dependency between processed frames, but the processing of each point in a frame can be done in parallel, and thus OpenMP version parallelizes the point-processing loop across participating threads. However, the number of iterations of the parallel loop, 51, is too small to utilize the abun-

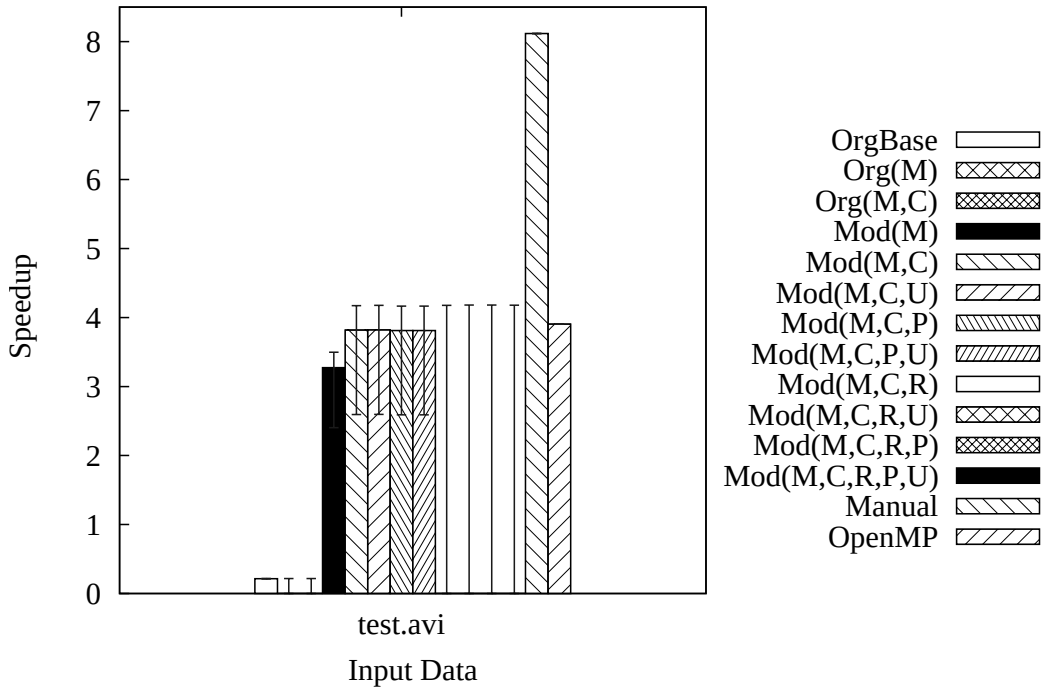


Fig. 2.30. Performance of HEARTWALL (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, *P* refers to *Parallel Loop-Swap*, *U* is *Loop Unrolling on Reduction*, and *R* means caching of in-block reduction data on the shared memory. The bars in each box show the performance variations caused by different thread batchings.

dant processing units in the GPU, and threads executing each iteration suffer from control flow divergences and uncoalesced memory access problems. Therefore, the performance of the automatically translated versions (*Org* in Figure 2.30) is very low.

The manual version (*Manual* in the figure) uses a different approach called braided parallelism; the manual version assigns processing of sample points to each thread block, while processing of individual pixels in each sample image is assigned to threads

in each thread block. Even though neither OpenMP API nor our translator provides a direct way to control the fine-grained mapping of each thread block and threads in each block to nested parallel loops, we could mimic the braided parallelism using the OpenMP *collapse* clause; (1) we manually apply loop distribution transformation to the original parallel loop, (2) for each resulting nested loop, we modify each statement in the loop body so that point-specific data are selected by the index variable of the outmost loop, while pixel-specific data are selected by the index variables of the inner loops, and (3) we add OpenMP *collapse* clauses to each nested loop. When the translator transforms the modified OpenMP program, it will collapse each nested loop annotated with an OpenMP *collapse* clause, and then each iteration of the collapsed loop will be mapped to each thread, and by setting thread batching properly, we can have mapping similar to the manual version. *Mod* in the figure is the version where the braided parallelism is mimicked, and the layout of some two-dimensional arrays are modified for coalesced accesses. The modified versions (*Mod*) perform much better than the original versions (*Org*), but the manual versions (*Manual*) still outperform the modified versions.

This performance gap is largely incurred by additional synchronization overheads in the modified versions. In the *HEARTWALL* code, processing of each point consists of many small serial steps with interleaved control statements. Each of the steps involves a small amount of computation, which is executed by a thread block in the manual version. Even though each step can be executed in parallel by a thread block, the dependency between steps requires explicit synchronizations. Because the dependency exists only within a thread block, the manual version uses the *__syncthreads()* runtime library, which enforces synchronization within a thread block. However, in the OpenMP-to-CUDA translation system, the only way to express synchronization is to split kernels, and thus the modified version consists of many small kernels due to these splitting, leading to much more kernel invocation overhead contrary to the manual version, which contains only one kernel. The *HEARTWALL* case suggests

that for achieving the best performance on some applications, low-level APIs, such as hiCUDA [14], may be needed to express GPU-specific features.

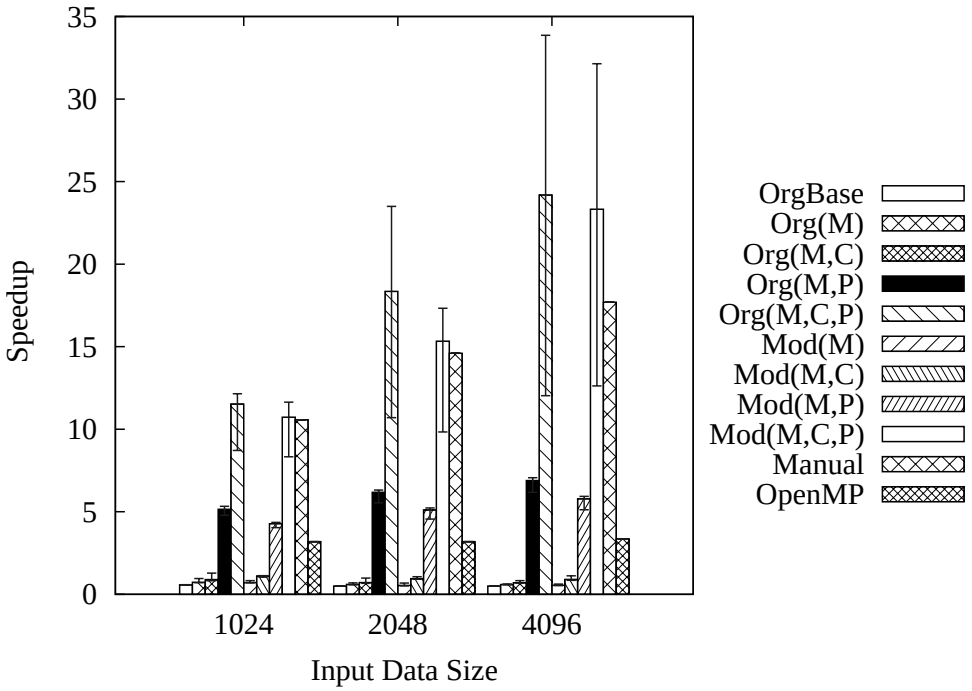


Fig. 2.31. Performance of SRAD (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, and *P* refers to *Parallel Loop-Swap* optimization. The bars in each box show the performance variations caused by different thread batchings.

2.6.10 Performance of SRAD

As explained in Section 2.6.9, *SRAD* kernel is another part of the *Heart Wall* application, and it performs a diffusion method called Speckle Reducing Anisotropic

Diffusion (SRAD) to remove locally correlated noise, known as speckles, in ultrasonic and radar imaging applications based on partial differential equations. In the *SRAD* kernel, *parallel loop-swap* optimization improves the performance successfully (*P* in Figure 2.31). When both *parallel loop-swap* and caching optimizations are combined, the automatically translated versions (*Org(M,C,P)*) outperform the hand-written CUDA versions (*Manual*). While the automatic versions rely on *parallel loop-swap* to enable coalesced accesses and use various caching optimizations such as registers and texture cache for temporal locality, the manual versions use the tiling transformation with shared memory for both coalesced access and temporal locality. However, two-dimensional tiling and caching on the shared memory incur more control flow divergences and additional synchronizations, and thus the overall performance improvement is less impressive than the automatic versions, which indicates that the optimal caching strategies can vary depending on locality patterns.

The manual version also applies an additional optimization to reduce the number of global memory accesses; the original OpenMP version uses subscript arrays to store index expressions for subscripted arrays, but the manual version calculates the index expressions directly without using the subscript arrays, which reside in the global memory. This optimization reduces the number of global memory accesses but increases the amount of computations and control flow divergences for calculating the index expressions. We applied the same optimization on the input OpenMP program, and the performance of the new version is shown as *Mod* in the figure. Comparison between the original versions (*Org*) and the modified versions (*Mod*) reveals that the performance gain by reducing global memory accesses are overwhelmed by the overhead of additional control flow divergences.

2.6.11 Performance of HOTSPOT

HOTSPOT is a tool to estimate processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively

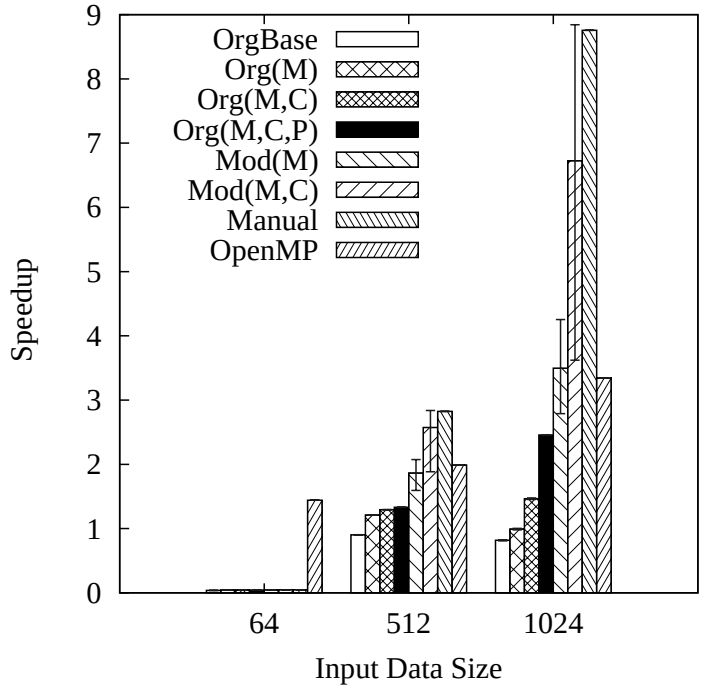


Fig. 2.32. Performance of HOTSPOT (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, and *P* refers to *Parallel Loop-Swap* optimization. The bars in each box show the performance variations caused by different thread batchings.

solves a series of differential equations. The original OpenMP program contains two parallel loops, and each loop has a nested inner loop, which is also parallelizable. The automatically translated versions (*Org* in Figure 2.32) assign each iteration of the outer parallel loops to a thread, as expressed in the original OpenMP program. However, this thread batching does not provide enough number of threads to hide the global memory latency; much of the global memory latency can be hidden by the

thread scheduling if there are sufficient independent threads that can be issued while waiting for the global memory access to complete.

The manual version (*Manual* in the figure) uses a two-dimensional partitioning scheme to increase the number of threads. It also use tiling transformation to exploit shared memory. Due to these differences, the manual versions show better performance than the automatic versions (*Org*). The current OpenMP-to-CUDA translator does not support multi-dimensional partitioning scheme, but we can get a similar effect by using the OpenMP *collapse* clause. *Mod* in the figure presents the performance when the *collapse* clauses are manually added to the nested parallel loops in the input OpenMP program. The current translator does not have a separate pass to automatically insert *collapse* clauses to general nested parallel loops. However, this may be automated since the underlying Cetus compiler has a tool to detect nested parallelism; this tool has already been used for the *parallel loop-swap* and *loop collapsing* techniques.

The results in Figure 2.32 show that the automatic versions augmented with OpenMP *collapse* clauses (*Mod*) perform slightly better the manual versions (*Manual*), where complex tiling optimizations are used.

2.6.12 Performance of KMEANS

K-Means (*KMEANS*) is a clustering algorithm used extensively in data-mining and many other areas. In K-means, a cluster of data objects are randomly divided into K sub-clusters. In each iteration, the algorithm associates each data object with its nearest center, based on some chosen distance metric. The new centroids are computed by taking the mean of all the data objects belonging to each sub-cluster respectively. The algorithm repeats the iteration until a given condition is satisfied. Calculating the new centroids has reduction patterns, but the original OpenMP version does not use OpenMP *reduction* clauses, since OpenMP does not support array reductions. Instead, the OpenMP version creates temporary arrays for

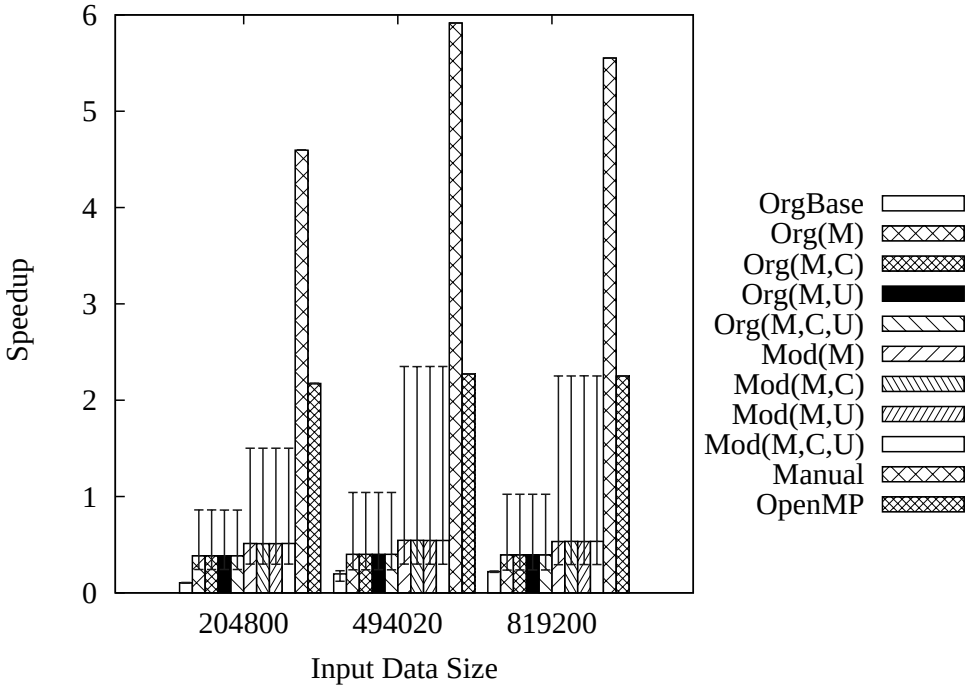


Fig. 2.33. Performance of KMEANS (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis, and *Mod* is another automatically translated and optimized version, but the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. In the parentheses, *M* means redundant memory transfer optimizations, *C* shows caching optimizations, and *U* refers to *Loop Unrolling on Reduction* optimization. The bars in each box show the performance variations caused by different thread batchings.

each thread to hold partial output, and after a parallel execution finishes, the main thread performs the array reduction by updating the output array with temporary arrays sequentially. Another equivalent, but still valid OpenMP version is to use OpenMP *critical* constructs. Because our translator can detect and transform array reduction patterns in the *critical* sections, we slightly modified the original OpenMP program such that the array reductions are expressed using the *critical* constructs.

The translated versions (both *Org* and *Mod* in Figure 2.33) use the modified OpenMP version, and thus all the reduction patterns were successfully transformed into the two-level tree reduction forms [11].

The hand-written CUDA code (*Manual* in the figure) also applies similar two-level reduction transformation techniques. As shown in Figure 2.33, however, the manual version performs much better than the automatic versions; this performance gap is mainly due to the differences in implementing the two-level reductions. The two-level tree reduction scheme consists of two steps: a local parallel reduction within each thread block followed by a host-side global reduction across thread blocks. To implement the in-block reduction, shared memory is usually used to hold the partial reduction outputs for each thread in a thread block. However, if a reduction variable is an array, keeping all the partial outputs may take too much space. In the *KMEANS* case, the shared memory usage for the partial reduction outputs exceeds the allowed limit. Therefore, our translator could not cache the partial outputs on the shared memory. To reduce the need for the temporary caching space during the in-block reduction step, the manual version applies a different technique; using a complex subscript manipulation, it could change the in-block reduction pattern so that each thread can update disjoint part of the in-block reduction outputs. To express these changes in the OpenMP code, special extensions to explicitly express the shared memory and GPU thread IDs will be needed.

2.6.13 Performance of LUD

LU decomposition (*LUD*) is a matrix decomposition tool, which writes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This decomposition is used in numerical analysis to solve systems of linear equations or calculate the determinant of a matrix. In the original OpenMP version, the main computation consists of only two simple parallel loops; one calculates an upper triangular part and the other computes a lower triangular part. However, it is known to

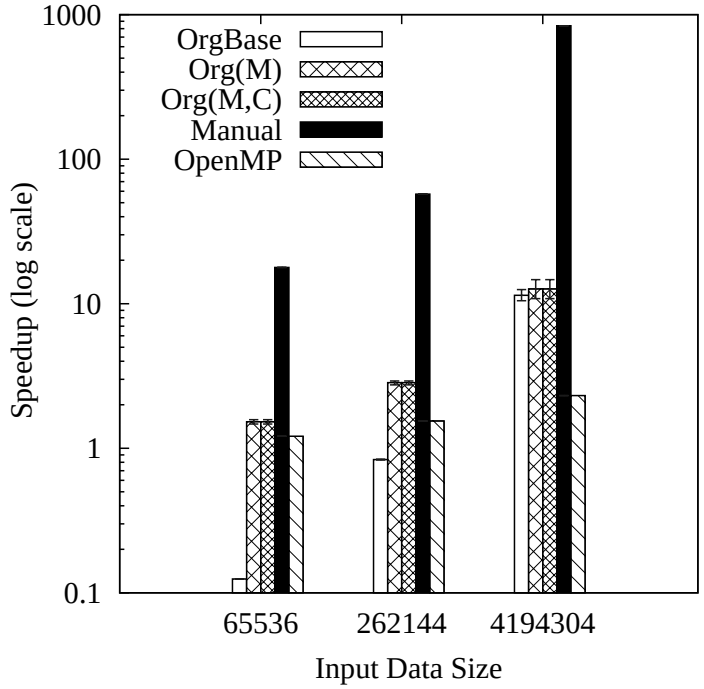


Fig. 2.34. Performance of LUD (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *Org-Base* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *M* means redundant memory transfer optimizations, and *C* shows caching optimizations. The bars in each box show the performance variations caused by different thread batchings.

be very difficult for compilers to analyze and generate efficient GPU code, due to its unique access patterns. The hand-written CUDA code uses a very complex partitioning algorithm; it partitions the whole matrix into many small blocks, and depending on the location of each block, it applies different computation patterns. The manual version also exploits shared memory by caching the tiled block onto shared memory. However, this tiling is also very complex, since each block may have different access patterns, requiring different tiling schemes. Due to these complex optimizations, the manual versions (*Manual*) could achieve very high speedups, as shown in Figure 2.34. Unfortunately, we could not express the complex manual optimizations

in the OpenMP code, and thus the automatic versions (*Org*) achieved relatively low speedups compared to the hand-written versions. This large performance gap motivates that for the best performance, our high-level translation system also needs to support an alternative way to express the CUDA programming and memory model in low level.

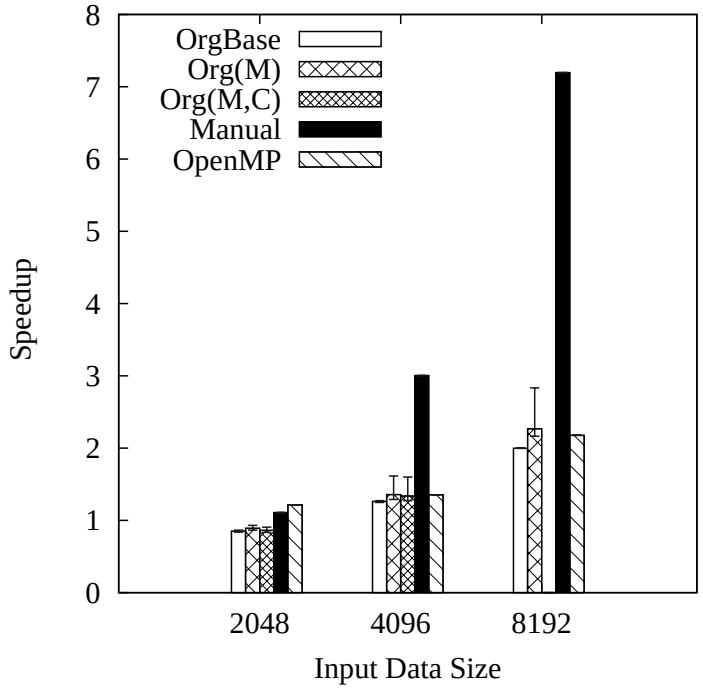


Fig. 2.35. Performance of NW (speedups are over serial on the CPU). Each box represents average speedups of the following versions: *OrgBase* is the baseline translation without optimizations, *Manual* is a hand-written CUDA version, and *OpenMP* is the OpenMP version run on the CPU with 4 threads. *Org* is the translation with various optimizations, which are shown in the parenthesis; *M* means redundant memory transfer optimizations, and *C* shows caching optimizations. The bars in each box show the performance variations caused by different thread batchings.

2.6.14 Performance of NW

Needleman-Wunsch (*NW*) is a nonlinear global optimization method for DNA sequence alignments. The algorithm consists of two steps; the first step fills a two-dimensional input matrix from top left to bottom right, step-by-step, and the second step traces back the matrix from bottom right. To achieve the optimal GPU performance, a tiling optimization using shared memory is essential. Due to the boundary access patterns, however, it may not be easy for the compilers to generate efficient tiling codes. Figure 2.35 shows the performance of both the automatically translated versions (*Org*) and the hand-written versions (*Manual*); the main reason for the performance gap is largely due to the lack of automatic tiling transformation capability in our translator.

2.7 Related Work

Prior to the advent of the CUDA programming model [5], programming GPUs was highly complex, requiring deep knowledge of the underlying hardware and graphics programming interfaces [15–17]. Although the CUDA programming model provides improved programmability, achieving high performance with CUDA programs is still challenging. Several studies have been conducted to optimize the performance of CUDA-based GPGPU applications: an optimization space pruning technique [18] has been proposed, using a Pareto-optimal curve, to find the optimal configuration for a GPGPU application. Also, an experimental study on general optimization strategies for programs on an NVIDIA CUDA-supported GPU has been presented [19]. In these contributions, optimizations were performed manually.

For the automatic optimization of CUDA programs, a compile-time transformation scheme [7] has been developed, which finds program transformations that can lead to efficient global memory access. The proposed compiler framework optimizes affine loop nests using a polyhedral compiler model. By contrast, our compiler framework optimizes irregular loops, as well as regular loops. Moreover, we have demon-

strated that our framework performs well on actual benchmarks as well as on kernels. CUDA-lite [20] is another translator, which generates codes for optimal tiling of global memory data. CUDA-lite relies on information that a programmer provides via annotations, to perform transformations. Our approach is similar to CUDA-lite in that we also support special annotations provided by a programmer. In our compiler framework, however, the necessary information is automatically extracted from OpenMP directives, and the annotations by a programmer are used for fine-grained tuning.

OpenMP [6] is an industry standard directive language, widely used for parallel programming on shared memory systems. Due to its well established model and convenience of incremental parallelization, the OpenMP programming model has been ported to a variety of platforms; compiler techniques to translate OpenMP applications into a form suitable for execution on a Software Distributed Shared Memory (DSM) system have been developed [21], and another compile-time translation scheme to convert OpenMP programs into MPI message-passing programs for execution on distributed memory systems has been proposed [22]. Recently, there have been several efforts to map OpenMP to Cell architectures [23–25]. Our approach is similar to the previous work in that OpenMP parallelism, specified by work-sharing constructs, is exploited to distribute work among participating threads or processes, and OpenMP data environment directives are used to map data into underlying memory systems. However, different memory architectures and execution models among the underlying platforms pose various challenges in mapping data and enforcing synchronization for each architecture, resulting in differences in optimization strategies. To our knowledge, the proposed work is the first to present an automatic OpenMP to GPGPU translation scheme and related compile-time techniques.

MCUDA [26] is an opposite approach, which maps the CUDA programming model onto a conventional shared-memory CPU architecture. MCUDA may be used as a tool to apply the CUDA programming model for developing data-parallel applications running on shared-memory parallel systems. By contrast, our motivation is to reduce the complexity residing in the CUDA programming model, with the help of OpenMP,

which we consider to be an easier model. In addition to the ease of creating CUDA programs with OpenMP, our system provides several compiler optimizations to reduce the performance gap between hand-optimized and auto-translated programs.

The compile-time transformations proposed in this chapter are not fundamentally new ones; vector systems use similar transformations [27–29]. However, the architectural differences between GPGPUs and vector systems pose different challenges in applying these techniques, leading to opposite directions; *parallel loop-swap* and *loop collapsing* transformations are enabling techniques to expose stride-one accesses in a program so that concurrent GPU threads can use the coalesced memory accesses to optimize the off-chip memory performance. On the other hand, *loop interchange* in vectorizing compilers is to enable vectorization of certain loops within a single thread.

2.8 Summary

In this chapter, we have described a compiler framework for translating standard OpenMP shared-memory programs into CUDA-based GPGPU programs. For an automatic source-to-source translation, several translation strategies have been developed, including a *kernel region* identifying algorithm. The proposed translation aims at offering an easier programming model for general computing on GPGPUs. By applying OpenMP as a front-end programming model, the proposed translator could convert the loop-level parallelism of the OpenMP programming model into the data parallelism of the CUDA programming model in a natural way; hence, OpenMP appears to be a good fit for GPGPUs. We have also identified several key transformation techniques to enable efficient GPU *global memory* access: *parallel loop-swap* and *matrix transpose* techniques for regular applications, and *loop collapsing* for irregular ones.

Experiments on both regular and irregular applications led to several findings. First, a baseline translation of existing OpenMP applications does not always yield good performance; hence, optimization techniques designed for traditional shared-

memory multiprocessors do not translate directly onto GPU architectures. This is because traditional optimizations focus on the intra-thread locality, while the inter-thread locality is the main issue in the GPU architectures. Second, efficient *global memory* access is one of the most important targets of GPU optimizations, but simple transformation techniques, such as the ones proposed in this chapter or the ones applied to the input OpenMP programs manually, are effective in optimizing global memory accesses. Third, to fully adapt certain types of applications, such as Rodinia Benchmark *LUD*, low-level APIs to directly express the underlying memory model and execution model may be needed. Fourth, complex interaction among limited hardware resources, which incurs large performance variations as shown in many of tested benchmarks, may require fine-grained tuning, which will be addressed in the following chapters.

3. ARTS: ADAPTIVE RUNTIME TUNING SYSTEM

3.1 Introduction

In the previous chapter, we have presented a compiler framework for an automatic translation of OpenMP applications into CUDA-based GPGPU applications. We have also identified several compile-time transformation techniques to achieve high performance. The evaluation of the proposed system shows that the translator and compile-time optimizations work well on both regular and irregular applications. However, the performance variations in the preliminary results also reveal that complex interactions among the hardware resources may necessitate fine-grained tuning for optimal performance.

With the increased complexity of current computing environments, writing high performance applications is very challenging. A variety of compile-time transformation techniques have been developed to optimize the programs. However, compiler optimizations are not always beneficial; the same program may behave differently depending on the underlying computing environments and complex interactions among optimizations may incur unexpected side effects. Therefore, achieving optimal performance requires combined efforts of both compilers and runtime system. For that purpose, this dissertation studies a compiler-driven runtime tuning system for dynamic adaptation of applications onto the underlying execution platforms. In preliminary work, we have proposed an adaptive runtime tuning system with emphasis on parallel irregular applications such as sparse matrix-vector (SpMV) multiplication kernels. The tuning system proposed in this chapter has been deployed to a distributed memory system with 32 nodes, but the tuning system is general enough to be applicable to both regular and irregular applications on diverse architectures, such as GPGPUs. An advantage of the proposed tuning system is that it does not require any prepro-

cessing steps, since the tuning system purely relies on runtime information. However, pure runtime tuning may incur large overhead if the tuning space is too large, and available information is also restricted to that accessible at runtime. Compile-time information can leverage the runtime tuning system by providing global information of applications. A framework, on top of which the compiler-assisted, combined runtime tuning system can be built, will be presented in Chapter 4. The rest of this chapter will explain the adaptive runtime tuning system, focused on parallel sparse matrix applications on distributed memory systems.

3.2 Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems

The importance of sparse matrix-vector (SpMV) multiplication as a computational kernel has led to a large number of research contributions to optimize its performance. Many contributions have improved SpMV multiplications on single-processor or shared memory systems [30, 31]. There has been a focus on architecture-oriented techniques, such as register blocking, cache blocking, and TLB blocking. While these techniques may be applied on distributed systems to tune individual nodes, they do not propose parallel distributed optimizations.

On distributed parallel SpMV multiplication, load balancing and communication cost reduction are two key issues. To address these issues, many graph-partitioning-based decomposition algorithms [32–35] have been proposed. Due to the complexity of these algorithms, they are generally applied as preprocessing steps, rather than as runtime optimizations. For runtime tuning, several decomposition heuristics have been suggested [36–40]. They generally aim at distributing non-zero elements as evenly as possible. However, these static allocation methods do not capture dynamic runtime factors affecting the performance. One approach proposed a framework for dynamic optimization of parallel SpMV operations, on top of which load-balancing

techniques can be added [41]. The approach aims at only high-level parallelism, however.

In this chapter, we propose runtime tuning mechanisms that can be applied dynamically to adapt distributed parallel SpMV multiplication kernels to the underlying environments. No preprocessing steps are necessary. Our adaptive *iteration-to-process mapping* system achieves load balance by re-assigning rows to processes, according to the measured execution-time workload of each process. In this way, our tuning system achieves better load balance than previous static allocation systems. Our adaptive system also selects, at runtime, from among several communication methods. We evaluated the proposed tuning system on 26 real sparse matrices from a wide variety of scientific and engineering applications. The experimental results show that our tuning system reduces execution time up to 68.8% (30.9% on average) over a base block-distributed parallel algorithm on a 32-node platform.

The rest of this chapter is organized as follows: Section 3.3 provides an overview of basic sequential and distributed parallel SpMV multiplication algorithms, and discusses previous approaches on parallel SpMV optimizations. Section 3.4 presents our runtime tuning system, which consists of an adaptive iteration-to-process mapping mechanism and a runtime communication selection system. Implementation methodology and experimental results on 26 sparse matrices are shown in Section 3.5 and Section 3.6, respectively. Section 3.7 presents the summary of this chapter.

3.3 Parallel SpMV Multiplication on Distributed Memory Systems

There are several sparse storage formats, which favor different parallelization strategies [42]. In this work, compressed row storage (CRS), which is the most widely used sparse-data format in scientific computing applications, is used to store a sparse matrix for SpMV computations. In CRS, rows are stored in consecutive order. A dense array, *val*, is used to store non-zero matrix elements in a row-wise fashion, and two other dense arrays are used to keep the positional information of each non-zero

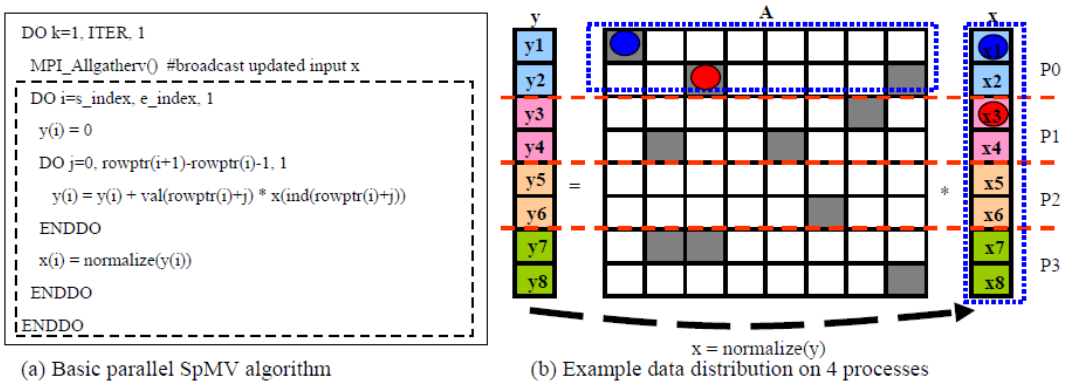
```
DO k = 1, ITER, 1
 DO i=1, N, 1
  y(i) = 0
  DO j=0, rowptr(i+1)-rowptr(i)-1, 1
   y(i) = y(i) + val(rowptr(i)+j) * x(ind(rowptr(i)+j))
  ENDDO
  x(i) = normalize(y(i))
 ENDDO
ENDDO
```

Fig. 3.1. Sequential algorithm for sparse matrix-vector multiplication, where N is the number of rows in matrix A

element: *ind*, which contains the column indices of each stored element, and *rowptr*, which contains pointers to the first non-zero element of each row in the array *val*. A sequential algorithm for sparse matrix-vector multiplication (y=Ax) is shown in Fig. 3.1.

For a base distributed parallel implementation, we followed the guidelines in [42]: one-dimensional data distribution and broadcast messages for all data communication. One-dimensional data distribution reduces the complexity of the load-balancing problem on heterogeneous clusters [43]. There are several algorithms for 2-D data partitioning [38, 40, 44]. However, due to their complexity, they are more suitable for single or shared memory systems than for distributed memory systems [36]. The use of broadcast messages for data communication is a natural choice, and it may perform optimally for common cluster interconnections such as Ethernet [42]. As we will see later in this chapter, however, different communication methods may be preferred depending on irregular parallel SpMV communication patterns.

In a parallel SpMV implementation for CRS format, the sparse matrix is block-distributed in a row-wise fashion, and the output vector y can also be block-distributed. On the other hand, the input vector x should be replicated to all processes because each process may need the entire input vector x for SpMV multiplication. At each outermost-loop iteration, each process computes its local matrix-vector multiplica-



(a) Basic parallel SpMV algorithm (b) Example data distribution on 4 processes

Fig. 3.2. Parallel algorithm and data distribution for sparse matrix-vector multiplication

tion part and broadcasts its newly updated input vector to all processes. The parallel algorithm and corresponding data distribution appear in Fig. 3.2.

In Fig. 3.2 (a), *s_index* and *e_index* determine the row blocks that each process will compute. For example, the process *P0* in Fig. 3.2 (b) calculates the matrix-vector product of the dotted region and generates new values *y(1:2)*. Before starting the next calculation, each process updates remotely computed values of x through MPI_Allgatherv() all-to-all communication.

In distributed parallel SpMV algorithms, irregular memory access patterns cause significant load imbalance in terms of both computation and communication. Resolving this issue at compile time is difficult because the data access patterns of the involved computation and communication may be known only at runtime. There exist extensive graph-partitioning-based decomposition algorithms [32–35] pursuing either minimum communication cost or computational load balancing. But due to their complexity or incurred large overhead, none of them are applicable as runtime techniques. Instead, these techniques are applied to sparse matrices as preprocessing steps. However, this means that users must preprocess every input sparse matrix before running SpMV kernels. Other research has proposed decomposition heuristics [36–39]; although less complex, they still incur non-negligible overhead. They are engaged as either preprocessing or static runtime allocation methods. Moreover, none

of the aforementioned techniques consider dynamic runtime factors such as computing power difference among the assigned nodes, the interconnection characteristics of deployed clusters, and interference from co-existing jobs.

3.4 Adaptive Runtime Tuning System

This section presents an adaptive runtime tuning system for distributed parallel SpMV multiplication kernels. The proposed tuning system consists of two steps: *normalized row-execution-time-based iteration-to-process mapping* and *runtime selection of communication algorithms*. The first step achieves computational load balance by mapping rows to processes dynamically, based on measured execution time; the second step finds the best possible communication method for message patterns generated by the first step. In contrast to previous approaches, which statically assign workload to each process according to non-zero element distributions, the proposed tuning system re-distributes the workload dynamically. In this way, our tuning system attains improved load balance. The proposed system assumes compressed row storage (CRS) format, but it can be applied to other storage formats, too. Detailed descriptions of the tuning system are presented in the following two subsections.

3.4.1 Normalized Row-Execution-Time-based Iteration-to-Process Mapping Algorithm

In distributed parallel SpMV multiplication, general decomposition algorithms [36–39] solve load-balancing problems by partitioning non-zero elements evenly among participating processes. In a one-dimensional data distribution case, the decomposition algorithms map rows to processes such that the numbers of non-zero elements assigned to each process are as even as possible. However, these mapping mechanisms do not consider other factors, such as loop overheads. For example, suppose that two processes are assigned the same number of non-zero elements to compute. If this assignment results in 10 out of 1000 rows being mapped to one process and the

remaining 990 rows to the other process, the two processes will incur different loop overheads, even though the amount of SpMV calculations is identical. Another problem of the existing decomposition algorithms is that they do not consider dynamic runtime-system performance. In batch-oriented large clusters, the computing nodes may consist of heterogeneous machines with different clock speeds or physical memory sizes. Moreover, if each node has multiple cores, depending on job configurations, some jobs may run on the same node, sharing its physical memory. The existing algorithms do not cover these dynamic factors affecting runtime performance.

To address these issues, we propose a dynamic runtime mapping mechanism called normalized row-execution-time-based iteration-to-process mapping. In the proposed algorithm, row-to-process mapping is performed at runtime, based on the execution time of each row. The optimization goal of our mapping algorithm is to find the best row-to-process mapping, such that measured SpMV computation times are as even as possible among processes. The basic approach to achieve this goal is as follows: initially rows are block-distributed evenly among processes and each process measures the execution time of each assigned row. The row-execution times are exchanged among processes in an all-to-all manner. The rows are block-distributed again, but inter-process boundaries are adjusted, such that each process has similar row-block-execution time, which is the sum of execution times of rows mapped to the same process. Due to the runtime performance difference, the execution times of the same row may differ on different processes. Therefore, the execution time of each row is measured again. This measure-and-map procedure is repeated until the difference of row-block-execution times are within some threshold (5% in our experiments).

Measuring and exchanging each row execution time may incur large measuring and communication overhead. To minimize the incurred overheads, we approximate each row execution time. At every outermost-loop iteration, which corresponds to one computation of SpMV multiplication, every process measures net computation time consumed to calculate row blocks assigned to the process. *Normalized row execution time (NRET)* is the measured time divided by the number of assigned rows.

$$NRET = \frac{exe.\,time\ for\ assigned\ rows\ computation}{the\ number\ of\ assigned\ rows}$$

In this approximation, each row assigned to one process has identical execution time regardless of the number of non-zeros contained in the row. To increase accuracy, the row execution time is recalculated whenever the row-to-process mapping is changed, and the newly estimated value is used for the next re-mapping. These repeated feedback steps have a grouping effect, so that the algorithm converges. In theory, convergence is not guaranteed, which can occur if there exist a few highly dense rows in small matrices.

To guarantee termination, our algorithm forces tuning to stop after a number of steps (20 in our experiments) and remain off for a *shelter period*. After that period (100 outermost-loop iterations in our experiments) the runtime environment is checked for changes; if unchanged, the algorithm repeats the shelter period. This process continues through the end of the program.

The overall procedure and a simple example are presented in Fig. 3.3. During the mapping phase, all calculations are redundantly executed by all processes to minimize the involved communication. As shown in the example, some row blocks should be migrated to a new process when the mapping is changed. However, the communication cost to migrate large row blocks may be high. To reduce the migration cost, our tuning system replicates the entire sparse matrix data rather than block-distributing the data among processes. This replication removes data migration complexity and the involved communication cost at the expense of increased memory pressure. Selecting between data replication and migration may be a further tuning target.

3.4.2 Runtime Selection of Communication Algorithms

On distributed memory systems, data communication is performed by explicit message passing. In this work, we use MPI [45] for data communication. MPI provides rich communication methods, offering variants and parameters that may be

Given that n = # of rows in sparse matrix A, m = # of involved processes

1. Each process, p, calculates average computation time (CT(p)) of the assigned SpMV comp. blocks

2. Each process collects CT(k) from all the other processes (k=1..m) through MPI communication (MPI_Allgather)

3. Each process calculates normalized row-execution time (NRET) and target computation time (tCT)

For k = 1..m

For each row, i, assigned to process k //s_index(k) and e_index(k) are row-block boundaries of process k

$$NRET(i) = \frac{CT(k)}{e\_index(k) - s\_index(k) + 1}$$

tCT = Average of {CT(k), k=1..m}

4. Each process re-maps rows to processes such that all processes have the same expected computation time (eCT)

For k = 1…m //Assume i was initialized to 1 at the beginning

eCT(k) = 0; s_index(k) = i; //Set a new start boundary for process k

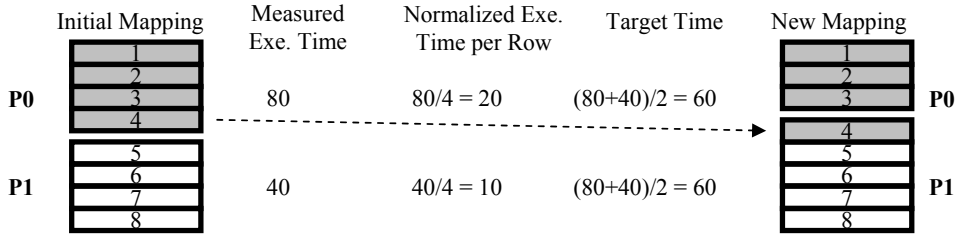
While ((eCT(k) < tCT) && (i<=n)) { eCT(k) += NRET(i); i++; }

e_index(k) = i-1; //Set a new end boundary for process k

e_index(m) = n; //Assign the remaining rows to the last process m if any exists

5. Repeat 1 ~ 4 until the difference of CT(k) among processes is within 5%

(a) Normalized row-execution time based iteration-to-process mapping algorithm



(b) Example case where 8 rows are mapped to 2 processes

Fig. 3.3. Normalized row-execution-time-based iteration-to-process mapping algorithm and an example

tuned for specific communication patterns. Choosing the right methods is important for high performance applications. An extensive study [46] on real MPI applications, performed on a Cray T3E, reveals that the most important performance bottleneck of MPI communication lies in the synchronization delay, which is the difference between the total time spent in the MPI communication call and the underlying data-transfer time. This synchronization delay depends on both computational and communicational load balances. Many existing approaches [47–49] only focus on individual collective-communication-call tuning, ignoring the synchronization delay caused by computational load imbalance. By contrast, our iteration-to-process mapping system addresses this delay. Given that the computational load-balancing problem is solved and there exist many solutions for individual-communication-call tuning, our

communication tuning system focuses on more high-level algorithmic approaches. In distributed parallel SpMV multiplication, each process updates remotely computed data at every outermost-loop iteration. This data exchange involves many-to-many communication. There are several options for this communication, as shown below.

Block broadcasting method (CM1): each process broadcasts locally written output blocks.

Block point-to-point exchange method (CM2): processes exchange bounding blocks containing needed elements through point-to-point communication.

Packed point-to-point exchange method (CM3): processes exchange exactly needed elements through point-to-point communication. This method involves explicit/implicit packing and unpacking.

The three methods perform differently depending on communication patterns. *CM1* may work well when large numbers of non-zero elements are distributed over an entire matrix. The rationale behind this method is that existing collective communications may be customized to all-to-all communication patterns with large data size. In contrast to *CM1*, *CM2* and *CM3* reduce the communication volume by calculating needed data for each process. *CM3* minimizes communication volume at the cost of additional computation and memory overheads for data packing and unpacking. Compared to *CM3*, *CM2* may include unnecessary elements in the bounding blocks. This does not affect correctness because these unnecessary elements are also the ones written by the sending process. In the proposed tuning system, the best communication method is selected at runtime, based on measured execution time. Since the performance of these methods is heavily dependent on a data distribution, the fastest method is re-selected whenever the adaptive mapping system changes the distribution. Measuring a global time is not simple in parallel executions because nodes have local timers, which are not synchronized. In the proposed tuning system, however, this problem is minimized by the balanced workload. Our selection mechanism simply uses the average measured communication time.

The idea of selecting the best variant among several high-level communication algorithms came from [50], where the best method is chosen at runtime, based on a performance model. One drawback of the previous approach is that the performance model is a simple latency-based point-to-point model, which does not capture complex communication behaviors. By contrast, our simple selection mechanism reflects dynamic runtime performance. Moreover, our system does not depend on prerequisites, such as measuring latencies between nodes to calculate model parameters [50].

3.5 Methodology

3.5.1 Implementation

For our experiments, we hand-coded an MPI-version of *spmul*, a common sequential SpMV multiplication kernel. We combined this base version of the distributed SpMV multiplication kernel with the described tuning system. As communication methods, we used MPI_Allgatherv() for *CM1* and non-blocking receive (MPI_Irecv()) combined with standard send (MPI_Send()) for *CM2* and *CM3*. For data packing in *CM3*, we used MPI_Type_indexed(), which provides implicit data packing and unpacking by the underlying MPI implementation.

3.5.2 Parallel Platforms

We used the Intel compiler 9.0 with option -O2 and MPICH2 1.2.7. We carried out the experiments on the Hamlet linux cluster at Purdue University. Hamlet consists of 308 IA-32 P4 nodes with two different processor speeds (3.06 GHz and 3.2 GHz) and two different physical memory sizes (2 GB and 4GB). We used 32 nodes connected with InfiniBand.

Table 3.1
Summary of sparse matrices used in evaluation

Name	Dim (NxN)	Non-zeros	Type	Description
af_shell10	1,508,065	27,090,195	diagonal	sheet metal forming
boneS10	914,898	28,191,660	diagonal	3D trabecular bone
rajat31	4,690,002	20,316,253	diagonal	circuit simulation matrix
Si41Ge41H72	185,639	7,598,452	diagonal	Real-space pseudo-potential method
SiO2	155,331	5,719,417	diagonal	Real-space pseudo-potential method
g7jac200sc	59,310	837,936	diagonal	Jacobian from CEPII
ldoor	952,203	23,737,339	even	INDEED test matrix
appu	14,000	1,853,104	even	fluid dynamics
ASIC_680ks	682,712	2,329,176	even	Xyce circuit simulation matrix
ASIC_680k	682,862	3,871,773	uneven	Xyce circuit simulation matrix
crankseg_2	63,838	7,106,348	uneven	OUTPUT4-Matrix
F1	343,791	13,590,452	uneven	Symmetric indefinite matrix
F2	71,505	2,682,895	uneven	AUDI engine piston rod
hood	220,542	5,494,489	uneven	INDEED Test Matrix (DC-mh)
nd6k	18,000	3,457,658	uneven	ND problem set
nd24k	72,000	14,393,817	uneven	ND problem set
ns3Da	20,414	1,679,599	uneven	3D Navier Stokes
poisson3Db	85,623	2,374,949	uneven	3D Poisson problem
sme3Db	29,067	2,081,063	uneven	3D structural mechanics problem
sme3Dc	42,930	3,148,656	uneven	3D structural mechanics problem
sparsine	50,000	799,494	uneven	structural optimization (CUTEr)
audikw_1	943,695	39,297,771	uneven	symmetric rb matrix
darcy003	389,874	1,167,685	uneven	discretization using mixed FE
inline_1	503,712	18,660,027	uneven	stiffness matrix
kkt_power	2,063,494	8,130,343	uneven	Optimal power flow
msdoor	415,863	10,328,399	uneven	medium size door

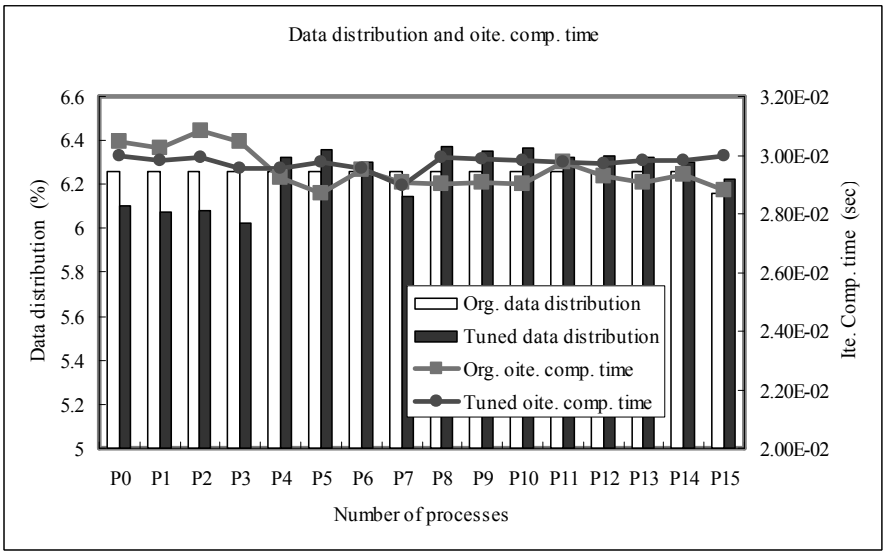
3.5.3 Evaluated Sparse Matrices

To evaluate the performance of our tuning system on the parallel SpMV multiplication kernel, we conducted experiments on 26 real sparse matrices in the UF Sparse Matrix Collection [13]. The matrices cover a wide range of application areas, such as finite element method, circuit simulation, and linear programming. A summary of the matrices appears in Table 3.1. Most of the matrices in the table are symmetric. For some of these cases, only one half of the matrix is stored, and we performed our experiments only on the stored information. While the overall execution time for these cases would be twice the numbers we obtained, the load-balancing results would be the same. The same algorithm would essentially be performed twice – once on the upper half and once on the lower half of the input matrix.

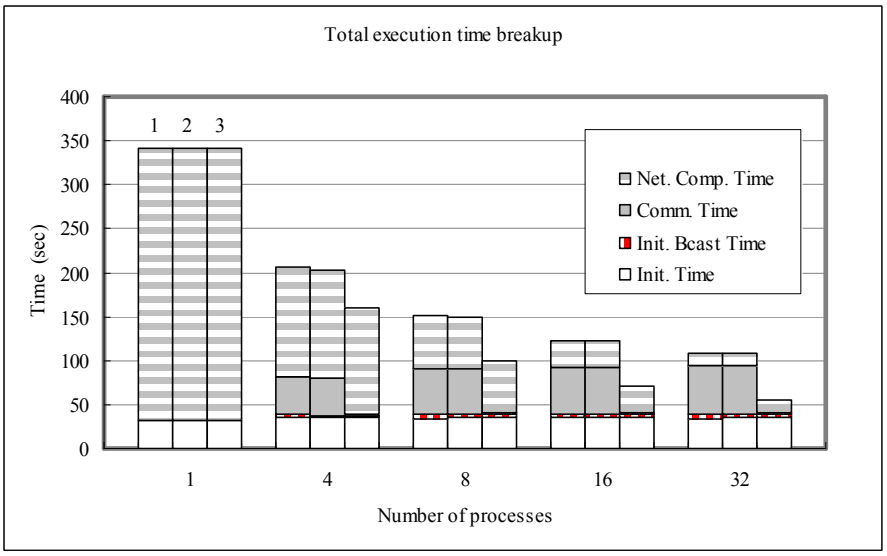
3.6 Experimental Results

This section presents the performance of the tuned parallel SpMV multiplication kernel on real matrices listed in Table 3.1. We compared our tuned version with the base parallel SpMV multiplication kernel described in Section 3.3. In the base implementation, rows were block-distributed evenly among processes. To separate the contributions of adaptive mapping and communication method selection, we also ran variants with only one of these techniques turned on. We implemented a static allocation method and compared it with our iteration-to-process mapping system. The overhead incurred by the tuning system was analyzed and the upper limit of the overhead was measured. Each kernel executed SpMV multiplication 1000 times and each test per input matrix was repeated three times. For time measurements, we used MPI_Wtime(), a built-in MPI function.

(a) Non-zero data distribution (graphical view)



(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)



(c) Execution time breakups for original, computation-tuning-only (CTuned), and tuned versions

Fig. 3.4. Non-zero data distribution and parallel performance of sparse matrix *af_shell*

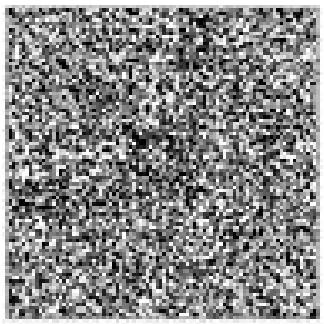
3.6.1 Parallel Performance

Performance impact of matrix structure: In SpMV computations, the data structure of the input matrices has a significant impact on the performance. In Table 3.1, matrices are classified into three types: diagonal, even, and uneven. In diagonal-type matrices, non-zero elements are allocated along and around the diagonal. These matrices are usually highly sparse and evenly distributed with respect to rows.

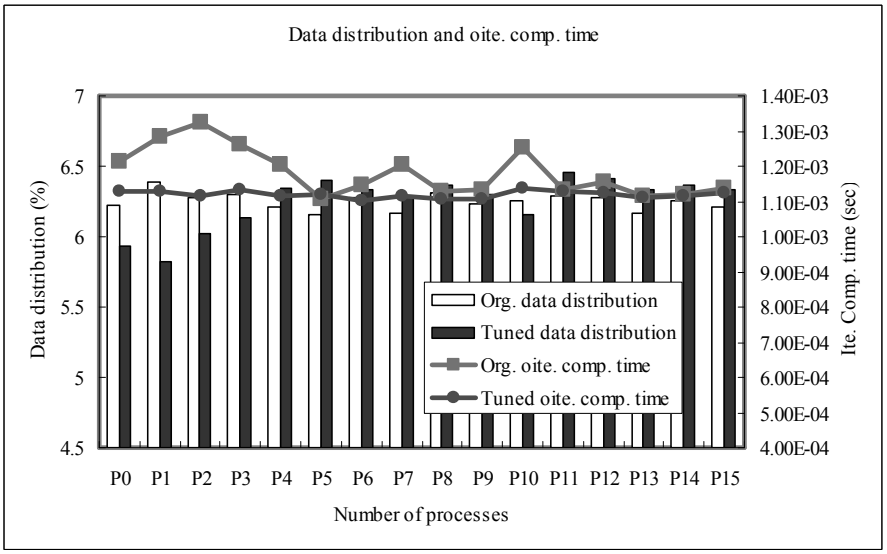
Fig. 3.4 shows an example of the diagonal-type matrices. Fig. 3.4 (b) displays non-zero element distribution and one outermost-loop iteration execution time (oite. comp. time), which is the time to execute SpMV computation blocks assigned to each process. The graph contains results of the original version (base parallel version) and the tuned version run on 16 nodes. In this case, the original data distribution is already fairly even. Nevertheless the effect of load balancing is evidenced in the figure.

Fig. 3.4 (c) shows execution time breakups for original, computation-tuning-only (*CTuned*), and tuned versions. The left bar in each group, marked as *1*, represents the original version, followed by the CTuned and tuned versions. From the graph, we can see that adaptive mapping does not improve significantly, but communication tuning is very effective (averaged total execution time reduction = 36%). In diagonal matrices such as *af_shell10*, each process has to exchange data only with its neighbors. In this case, point-to-point communications such as *CM2* or *CM3* are more suitable than heavy all-to-all communication (*CM1*). In real experiments on 4, 8, 16, and 32 nodes, *CM3* is selected for 8 out of 12 executions and *CM2* was selected for the remaining executions.

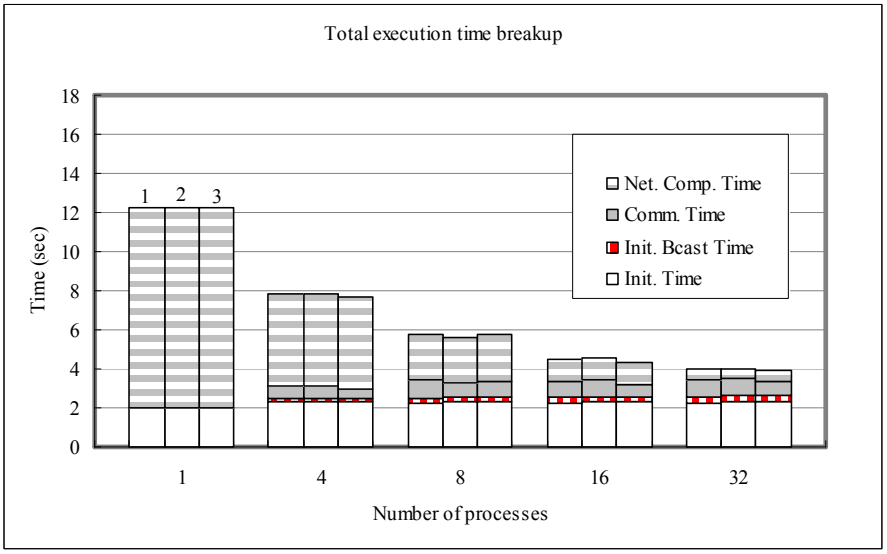
Fig. 3.5 represents the case for even-type matrices, whose non-zero elements are not allocated diagonally but still distributed evenly with respect to rows. As shown in Fig. 3.5 (a), the matrices of this type usually have non-zeros distributed randomly over the entire matrix. As the *af_shell10* case did, Fig. 3.5 (b) also shows that compu-



(a) Non-zero data distribution (graphical view)



(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)

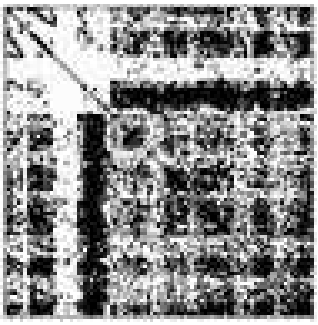


(c) Execution time breakups for original, computation-tuning only (CTuned), and tuned versions

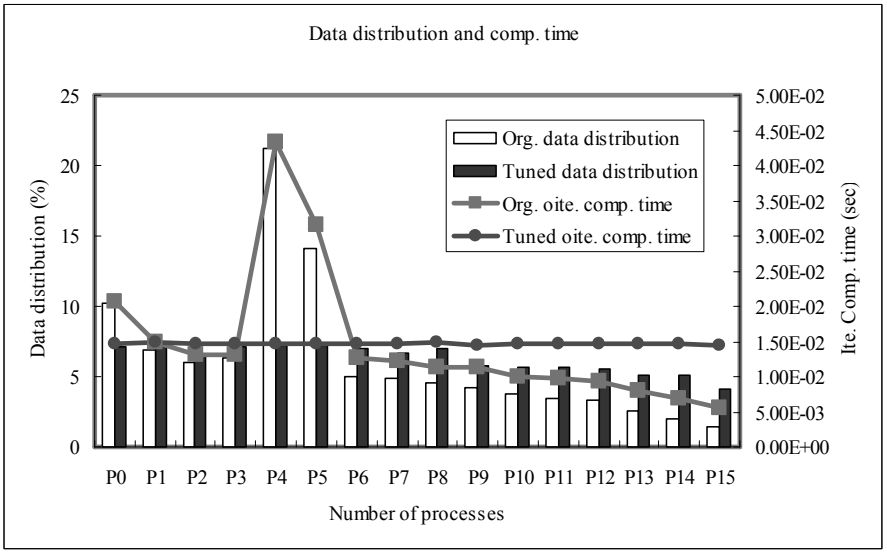
Fig. 3.5. Non-zero data distribution and parallel performance of sparse matrix *appu*

tational load balance can be achieved with uneven non-zero distribution, even though the performance benefits are negligible. In this case, both adaptive mapping and communication tuning are of little use. Communication tuning is not useful because the data distribution in this case requires all-to-all communication, where *CM1* may perform better than others, depending on the sparsity and communication volume. In our experiments, *CM3* was selected more often than *CM1*, but the results indicate that the performance of *CM3* is no better than *CM1*, in this case.

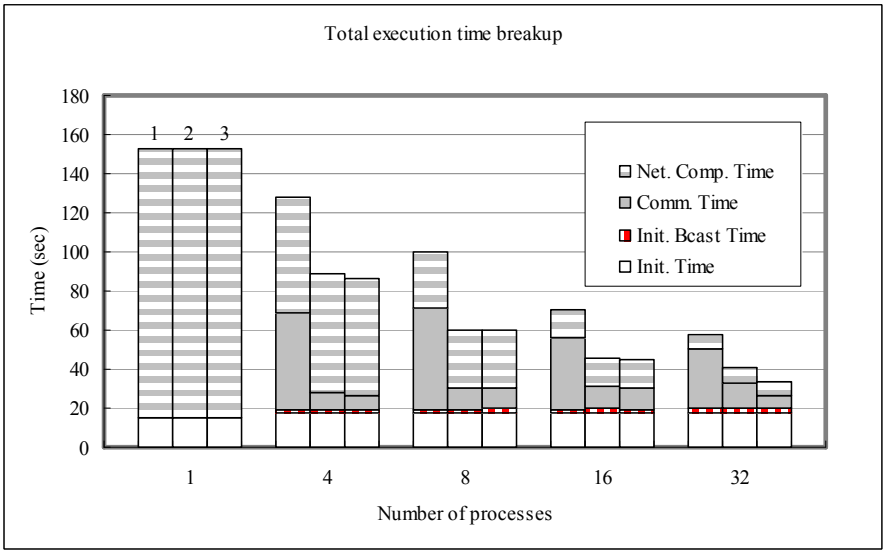
The matrix *F1* in Fig. 3.6 is highly uneven. Consequently, the data distribution is also uneven and its communication pattern is irregular. Fig. 3.6 (b) and (c) illustrate the power of our adaptive mapping system. In the *F1* input case, our mapping system achieves load balance with 5.6 tuning calls on average, and the incurred overhead is 0.45% on average. With this small overhead, our tuning system reduces the total execution time by 37%. The main execution time (the time to execute the main SpMV computation body excluding the initial input data distributing section,) is reduced by 49.6% on average. From Fig. 3.6 (c), we can see that computational load balancing reduces communication time. By balancing the workload among processes, the synchronization delay, mentioned in Section 3.4.2, can be reduced. As shown in Fig. 3.6 (a), *F1* has non-zero elements randomly distributed among the entire matrix. Therefore, communication tuning is not effective. On the other hand, in the 32-node case (rightmost group in Fig. 3.6 (c)), there is a noticeable time reduction between the middle bar (CTuned version) and the right bar (tuned version). This suggests that communication characteristics can be changed within the same input matrix, depending on the number of involved processes. In our experiments on the *F1* input, *CM1* or *CM2* were selected for the 4, 8, and 16 node cases, but the 32-node case preferred *CM3*. *CM3* minimizes the communication volume, but it requires complex data packing and unpacking. If large numbers of non-zero elements are located closely but not continuously, *CM3* will perform worse than *CM2*, when its packing overhead offsets the benefits from communication volume reduction. This happened on 4, 8, and 16 nodes. On 32 nodes, however, communication volume reduction by *CM3*



(a) Non-zero data distribution (graphical view)

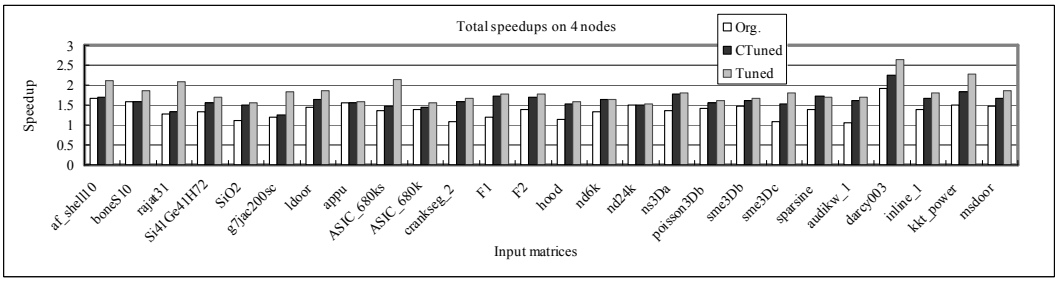


(b) Non-zero data distribution and outermost-loop iteration execution time (oite. comp. time)

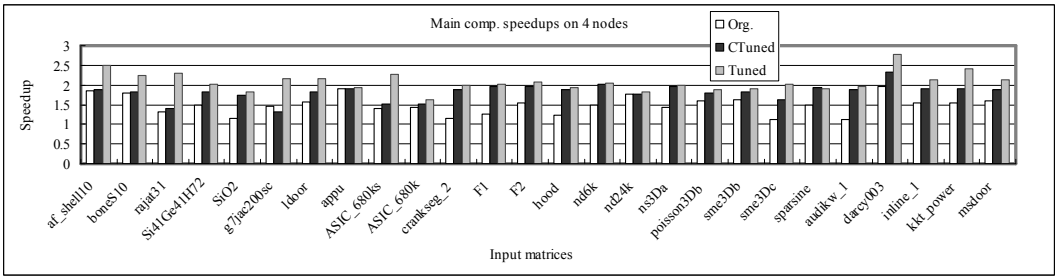


(c) Execution time breakups for original, computation-tuning only (CTuned), and tuned versions

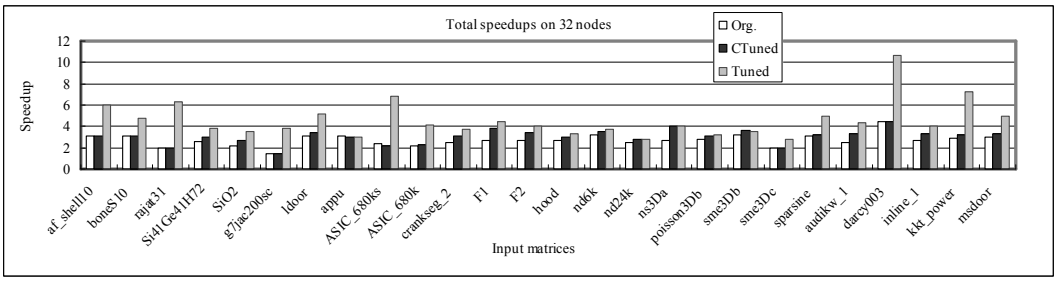
Fig. 3.6. Non-zero data distribution and parallel performance of sparse matrix *F1*



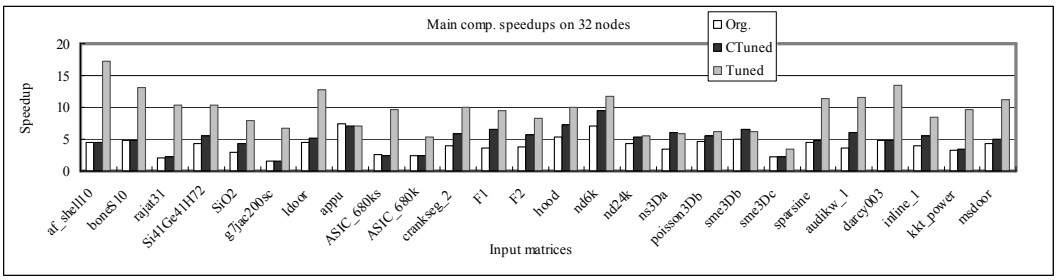
(a) Total execution time speedups on 4 nodes



(b) Main execution time speedups on 4 nodes



(c) Total execution time speedups on 32 nodes



(d) Main execution time speedups on 32 nodes

Fig. 3.7. Speedups of all 26 matrices on 4 and 32 nodes

outweighed the packing overhead. This case benefits from tuning even within the same input data and the same target architecture.

These three cases are representative of the performance behavior of our tuning system.

Overall performance on 26 matrices: Speedups of all 26 matrices on 4 and 32 nodes are presented in Fig. 3.7. In the figure, *total speedups* are the speedups in terms of total execution time and *main comp. speedups* show main execution speedups. The graphs reveal several interesting behaviors. First, in the 4-node cases, both adaptive mapping and communication tuning work well, but on 32 nodes, communication tuning is more important than adaptive mapping. This effect is caused by computational load balancing, which becomes less problematic as the number of involved processes increases. When computation is distributed to a large number of processes, less work is assigned to each process on average. Hence, the synchronization delay caused by load imbalance tends to be smaller. Another reason is that the benefit of using *CM2* or *CM3* becomes less on small numbers of nodes; communication volume tends to be large and it is more likely to be all-to-all communication. In such case, *CM1* may perform better than the others. In our experiments, *CM3* was chosen more often than *CM1* even on 4 nodes, but the performance benefit of *CM3* was much less than on 32 nodes.

Second, there is a gap between total execution speedup and main execution speedup and the gap increases as the number of nodes increases. In the current parallel implementation, input data are read by process *P0* and the data are broadcast to all the other processes. The broadcast message replicates input data to all involved processes, which adds non-negligible overhead. If we block-distribute the input data, the initial overhead may reduce, but the distribution may cause several input data migrations during the adaptive mapping phase. There is an opportunity to further develop efficient algorithms for data migration. Table 3.2 summarizes the execution time reductions measured for the 26 matrices.

Table 3.2
Execution time reduction by the proposed tuning system: In *A(B)* format, *A* represents the average of 26 matrices and *B* is the maximum value

	4 nodes	8 nodes	16 nodes	32 nodes	overall
Main time reduction (CTuned) (%)	17.2 (40.4)	22.1 (55.2)	20.7 (51.7)	16.3 (45)	19.1 (55.2)
Main time reduction (Tuned) (%)	27.8 (44.1)	38.8 (62.1)	46.1 (75.9)	53.2 (79.3)	41.5 (79.3)
Total time reduction (CTuned) (%)	14.9 (34.2)	16.9 (44.7)	14 (37.8)	9.6 (32)	13.8 (44.7)
Total time reduction (Tuned)(%)	23.9 (39.6)	30.6 (55.6)	33.3 (66.7)	36 (68.8)	30.9 (68.8)

3.6.2 Comparison of Static Allocation and Adaptive Iteration-to-Process Mapping

This section compares our adaptive iteration-to-process mapping system with a static allocation method, which maps rows to processes statically, such that non-zero elements are distributed evenly among processes. Fig. 3.8 contains the speedups of both our mapping algorithm (CTuned) and the static allocation algorithm (SA) on 4 and 16 nodes. The figure reveals that our adaptive mapping system performs equally or better than the static allocation method in most cases. *darcy003* and *kkt_power* are quite interesting cases; the static method gets hardly any speedups on both 4 and 16 nodes, while our adaptive mapping achieves reasonable ones. *darcy003* and *kkt_power* have a large number of consecutive rows that do not contain any non-zero elements. When the static method is applied, these rows are assigned to one process. Fig. 3.2 (a) shows that the rows containing no non-zero elements still have some computations, such as loop checking and normalization. If a great number of such rows are assigned to one process, as the static allocation does, these small computations build up significant workloads, causing severe load imbalance. By contrast, our adaptive mapping system considers effective workloads experienced by each process. Therefore, our mapping system causes no such load imbalance problem. These results indicate that non-zero-element-distribution-based load-balancing mechanisms may not be optimal, depending on input matrix characteristics.

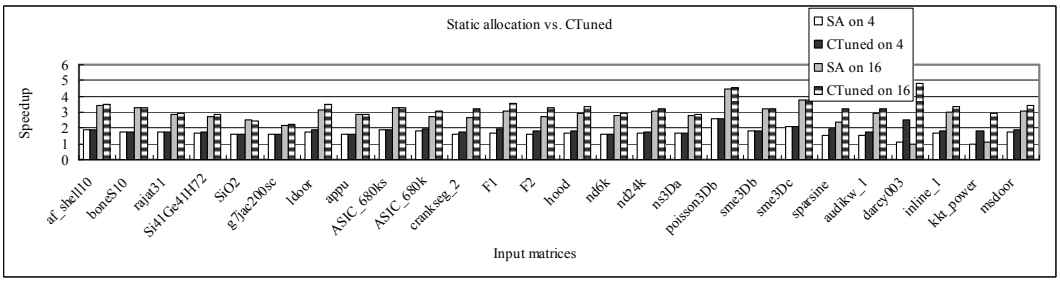


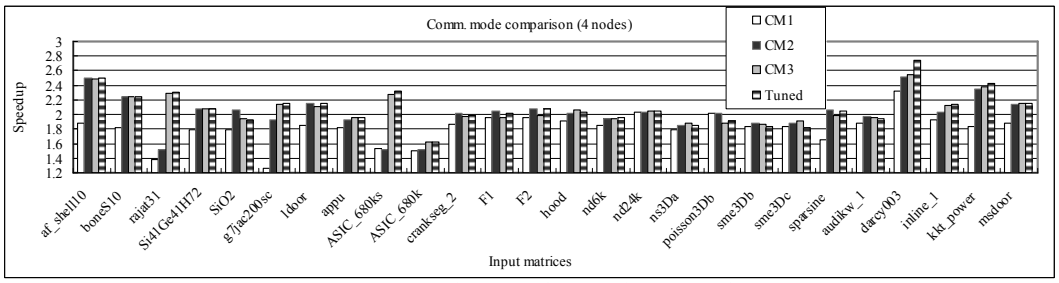
Fig. 3.8. Performance comparison of static allocation method (SA) vs. adaptive iteration-to-process mapping method (CTuned)

3.6.3 Comparison of Tuned and Fixed Communication

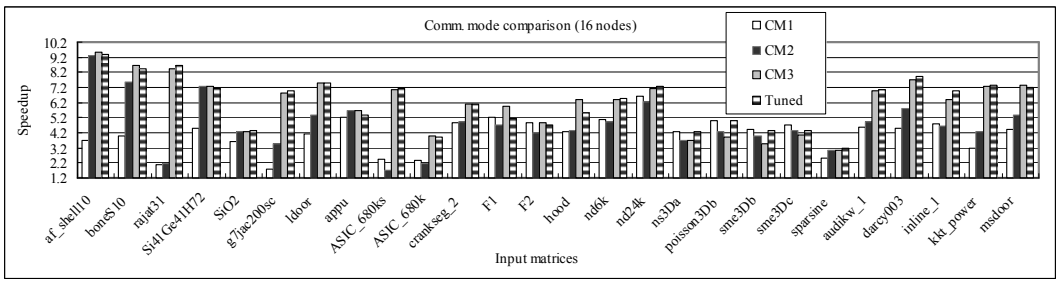
In this section, we study the effect of runtime selection of communication methods. For this study, we compared our tuned version with fixed-communication-method versions (*CM1*, *CM2*, and *CM3*). Adaptive mapping is applied to all versions to rule out the effect of load balancing. The speedups on 4 nodes and 16 nodes are shown in Fig. 3.9. The results indicate that *CM3* works well in most cases, except for *ns3Da*, *poisson3Db*, *sme3Db*, and *sme3Dc*, where *CM1* works best. This phenomenon occurs because we deal with sparse matrices. Table 3.1 shows that most of the matrices used in the experiments are very large but highly sparse at the same time. On these matrices, *CM3* may generate much smaller communication volume than *CM1* or *CM2*, such that the packing overhead can be overcome by reduced communication size. By comparing graphs in Fig. 3.9 and Table 3.1, we can find the trend that *CM1* or *CM2* is preferred to *CM3*, as input matrices become denser. Fig. 3.9 shows that our tuning system can capture the best communication method in all cases.

3.6.4 Tuning Overhead

Fig. 3.10 presents tuning overhead, which is represented by the percentage of tuning time in the total execution time. Fig. 3.10 (a) contains measured overheads on 4, 8, 16, and 32 nodes. From the graph, we can see that the overheads increase as the

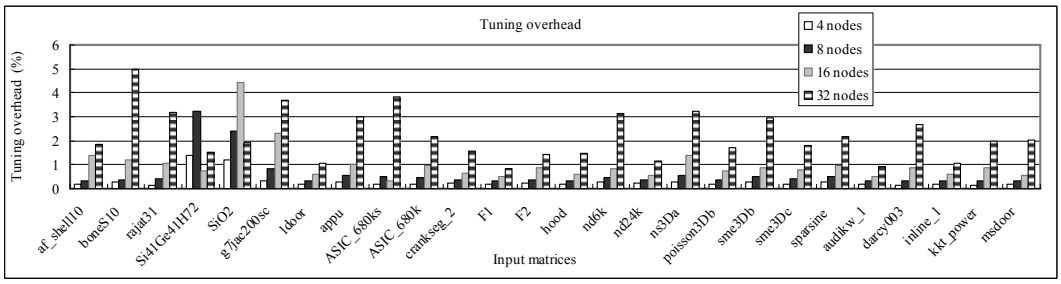


(a) Communication mode comparison on 4 nodes

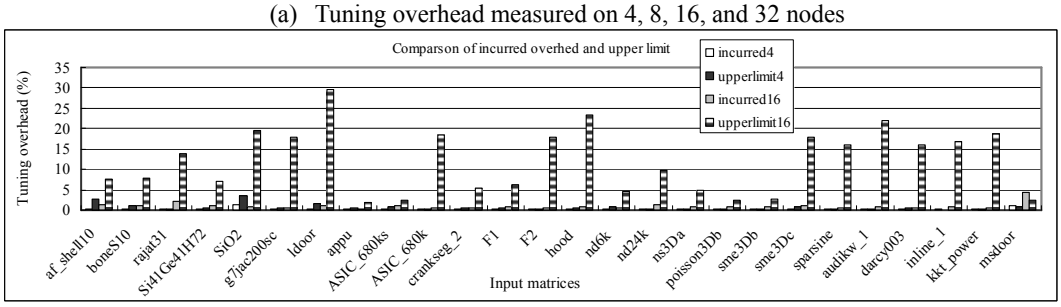


(b) Communication mode comparison on 16 nodes

Fig. 3.9. Performance comparison of fixed communication modes vs. tuned mode



(a) Tuning overhead measured on 4, 8, 16, and 32 nodes



(b) Comparison of incurred overhead and upper limit

Fig. 3.10. Tuning overhead (percentage of tuning time in the total execution time)

number of nodes increases. The proposed tuning system involves a small number of all-to-all communications for each process to collect timing information of the oth-

ers. Even though the required communication volume is small, these collective calls become heavier, as the number of nodes increases. Fig. 3.10 (b) compares incurred overhead and upper limit, which is the case where the tuning system is called continuously until the program finishes. The upper limit tests were conducted on 4 and 16 nodes. On 4 nodes, the upper limit is still small (0.72% on average), but on 16 nodes, it goes up to 12% on average. However, the actually incurred overhead remains small (0.3%, 0.6%, 1%, and 2.2% on 4, 8, 16, 32 nodes, respectively). This shows that our tuning system is efficient and converges quickly in most cases.

3.7 Summary

We have presented an adaptive runtime tuning system for distributed parallel SpMV multiplication kernels. Our adaptive mapping system solves load-balancing problems by dynamically re-assigning matrix rows to processes, according to measured real-time workloads. To minimize the incurred communication cost, our runtime selection system finds the best communication method for the data distribution tuned by our mapping mechanism. Experiments on 26 sparse matrices from various scientific and engineering applications led to several key findings. First, load-balancing mechanisms based on non-zero element distribution may not be suitable. Most previous work attempts to achieve load balance by distributing non-zero elements among processes as evenly as possible. However, our results show that load balancing may be lowest with uneven data distribution, when considering the underlying runtime environment. Second, computational load balancing can play an important role in minimizing communication cost, as it can reduce synchronization delay. Third, different data distributions favor different communication methods, even though point-to-point communication methods work well in many cases. This is because the sparsity of matrices is often very high. In these cases, it is likely that either communication volume is small or all-to-all communication is unnecessary. Fourth, initial input data transfer times add non-negligible overhead to the parallel execution time. To reduce

this overhead, more studies on various storage formats or data migration algorithms are needed.

Even though this chapter focuses on sparse matrix-vector multiplication tuning, our adaptive tuning system can be applied to general irregular applications such as N-body problems. The work presented in this chapter generalizes a compiler-driven adaptive tuning system, where a compiler identifies irregular loops showing repetitive computation and communication patterns and generates necessary inspector-executor codes. The performance improvement achieved in this chapter suggests that the compiler-driven, adaptive tuning system similar to the one presented in this chapter will be able to successfully tune the performance of the translation system presented in Chapter 2. An important infrastructure to enable the compiler-driven adaptive tuning system for the OpenMP-to-GPGPU translation system presented in Chapter 2 will be presented in the next chapter.

4. OPENMPC: EXTENDED OPENMP PROGRAMMING AND TUNING FOR GPUS

4.1 Introduction

This chapter focuses on the possibility of creating an integrated framework that combines the compiler framework in Chapter 2 and the adaptive runtime tuning system in Chapter 3, such that standard OpenMP programs are automatically translated, transformed, and tuned for the underlying GPGPU architectures.

In Chapter 2, we have proposed an automatic OpenMP-to-CUDA translation framework, which extends the ease of creating parallel applications with OpenMP to GPGPU architectures such as CUDA. It includes several optimizations that deal with the architectural differences between traditional shared memory systems, served by OpenMP, and stream architectures adopted by most GPUs.

However, developing efficient CUDA programs still remains difficult; complex interactions among hardware resources and the multi-layered software execution stack used for CUDA compilation and execution limit the compiler's ability to predict the performance effect of its optimizations [18,51]. In the OpenMP-to-CUDA translation framework, the CUDA programming model and memory model are transparent to users. However, this transparency comes at the cost of reduced control over fine-grained tuning. Achieving optimal performance with the generated programs may require additional, manual changes to the output CUDA code, which can be tedious and error-prone [14, 51, 52].

There has been extensive work on optimizing the performance of CUDA-based GPGPU programs. Studies on general optimization strategies found that the performance difference between well optimized GPU applications and poorly optimized ones can be orders of magnitude [7, 18, 19]. The studies also show that the level of

effort and expertise required to obtain optimal application performance on GPGPUs can be very high. Even though there are several efforts to automatically optimize and tune the performance of GPGPU programs, most of them are either application-specific [53–55], restricted to certain types of applications [7], or applied to only a small subset of optimization parameters [51]. Therefore, achieving maximum performance for general GPGPU applications is still a challenge and usually involves manual work.

To overcome this challenge, we propose *OpenMPC* – OpenMP extended for CUDA, which consists of a standard OpenMP API plus a new set of directives and environment variables to control important CUDA-related parameters and optimizations. OpenMPC provides programmers with a high-level abstraction of the CUDA programming model. It also provides a tuning environment that assists users in generating CUDA programs in many optimization variants without detailed knowledge of the programming and memory models.

We have developed a fully automatic, parameterized compilation and user-assisted tuning system supporting OpenMPC. In addition to a range of compiler transformations and optimizations, the system includes tuning capabilities for generating, pruning, and navigating the search space of compilation variants.

To evaluate the effectiveness of OpenMPC, a set of tuning tests were conducted using the reference compilation system and tuning tools. The experiments on fourteen OpenMP programs (two kernel benchmarks, three NAS Parallel Benchmarks, and nine Rodinia Benchmarks) demonstrate that tuning using OpenMPC with optional manual modification on the input OpenMP programs, which may be automated by an advanced compiler, can improve the performance up to 7.71 times (1.24 times on average) over un-tuned versions, which is 75% of the performance of hand-written CUDA versions. If we exclude one exceptional case (Rodinia Benchmark *LUD*), the average speedups will be 92% of those of the manual CUDA versions.

The rest of this chapter is organized as follows: Section 4.2 introduces OpenMPC, and Section 4.3 presents a reference compilation system and a prototype tuning

framework supporting OpenMPC. Experimental results are shown in Section 4.4, and related work and summary are presented in Section 4.5 and Section 4.6, respectively.

4.2 OpenMPC: Extended OpenMP for CUDA

OpenMPC extends the programming system described in Chapter 2 by adding new directives and environment variables that enable users and automatic tuning systems to apply CUDA-specific optimizations. The OpenMPC optimization system uses these directives to pass information generated by various analysis passes to the actual OpenMP-to-CUDA translator.

4.2.1 Directive Extension

The format of the OpenMPC directives is shown in Table 4.1.

Table 4.1
OpenMPC directive format

#pragma cuda gpurun [clause [,] clause]...]
#pragma cuda cpurun [clause [,] clause]...]
#pragma cuda nogpurun
#pragma cuda ainfo procname(pName) kernelid(kID)

The directives in the table are used to annotate OpenMP parallel regions using the syntax common in OpenMP. The first two directives (*gpurun* and *cpurun*) can be inserted either automatically by the compiler or manually by the user, while the third directive (*nogpurun*) is inserted mostly by the user, and the last directive (*ainfo*) is always inserted by the compiler. The *gpurun* directive specifies that the attached parallel region is eligible for kernel-region transformation. Clauses that may be used for this directive are shown in Table 4.2, Table 4.3, and Table 4.4. The *gpurun* directive can control the translation of each kernel region. The *cpurun* directive says that the

Table 4.2
Brief description of OpenMPC clauses, which control kernel-specific thread batchings and optimizations

Clause	Description
maxnumofblocks(N)	Set Maximum number of CUDA thread blocks for a kernel
threadblocksize(N)	Set CUDA thread block size for a kernel
noloopcollapse	Do not apply Loop Collapsing optimization
noploopswap	Do not apply Parallel Loop-Swap optimization
noreductionunroll	Do not apply loop unrolling for in-block reduction

Table 4.3
Brief description of OpenMPC clauses, which control kernel-specific data caching strategies

Clause	Description
registerRO(list)	Cache R/O variables in the list on GPU registers
registerRW(list)	Cache R/W variables in the list on GPU registers
noregister(list)	Set the list of variables not to be cached on GPU registers
sharedRO(list)	Cache R/O variables in the list on GPU shared memory
sharedRW(list)	Cache R/W variables in the list on GPU shared memory
noshared(list)	Set the list of variables not to be cached on GPU shared memory
texture(list)	Cache variables in the list on GPU texture memory
notexture(list)	Set the list of variables not to be cached on GPU texture memory
constant(list)	Cache variables in the list on GPU constant memory
noconstant(list)	Set the list of variables not to be cached on GPU constant memory

associated parallel region will be executed by the CPU. For this directive, the following four clauses from Table 4.4 can be used: *c2gmemtr*, *noc2gmemtr*, *g2cmemtr*, and *nog2cmemtr*. The third directive (*nogpurun*) prevents the translator from transforming the attached kernel region. In our system, the *gpurun* directive is usually added by the automatic translator; it can be overridden by a *nogpurun* directive inserted by a user or tuning system. The translator uses the *ainfo* directive to assign unique IDs to each kernel region. This allows programmers and tuning systems to provide addi-

Table 4.4
Brief description of OpenMPC clauses, which control data mapping or movement between CPU and GPU. These clauses are used either internally by a compiler framework or externally by a manual tuner.

Clause	Description
c2gmemtr(list)	Set the list of variables to be transferred from a CPU to a GPU
noc2gmemtr(list)	Set the list of variables not to be transferred from a CPU to a GPU
g2cmemtr(list)	Set the list of variables to be transferred from a GPU to a CPU
nog2cmemtr(list)	Set the list of variables not to be transferred from a GPU to a CPU
nocudamalloc(list)	Set the list of variables not to be CUDA-mallocated
nocudafree(list)	Set the list of variables not to be CUDA-freed

tional directives via a separate *user directive file*, rather than annotating the input OpenMP code. Directives provided in a user directive file have a similar syntax as in Table 4.1, but are prefixed by the procedure name and kernel ID they refer to.

4.2.2 Environment Variable Extension

The OpenMPC provides a rich set of environment variables, which control the program-level behavior of various optimizations or execution configurations for an output CUDA program. Table 4.5, Table 4.6, and Table 4.7 show the supported environment variables; variables in Table 4.5 control default behaviors of various optimizations and translations and set default thread batching for an input program. Those in Table 4.6 control program-level behaviors of data caching strategies, which may not be needed if each kernel region has been already annotated by data-caching clauses in Table 4.3. The ones in Table 4.7 are special variables used for setting various tuning configurations. Because directives have priority over environment variables, users or tuning systems can alter the program-level optimizations and configurations for each kernel region.

Table 4.5
Brief description of OpenMPC environment variables, which control program-level behaviors of various optimizations, thread batchings, and translation configurations.

Parameter	Description
maxNumOfCudaThreadBlocks=N	Set the maximum number of CUDA thread blocks
cudaThreadBlockSize=N	Set the default CUDA thread block size
useMatrixTranspose	Apply Matrix Transpose optimization
useLoopCollapse	Apply Loop Collapsing optimization
useParallelLoopSwap	Apply Parallel Loop-Swap optimization
useUnrollingOnReduction	Apply loop unrolling for in-block reduction
useMallocPitch	Use cudaMallocPitch() for 2-dimensional arrays
useGlobalGMalloc	Allocate GPU variables as global variables
globalGMallocOpt	Apply CUDA malloc optimization for globally allocated GPU variables
cudaMallocOptLevel=N	Set CUDA malloc optimization level for locally allocated GPU variables
cudaMemTrOptLevel=N	Set CUDA CPU-GPU memory transfer optimization level
assumeNonZeroTripLoops	Assume that all loops have non-zero iterations
convStatic2Global	Convert static variables in procedures except for main into global variables
disableCritical2ReductionConv	Disable critical-to-reduction conversion pass
UEPRemovalOptLevel=N	Set the level of upwardly exposed private variable removal optimization
MemTrOptOnLoops	Apply memory transfer promotion optimization
localRedVarConf=N	Configure how local reduction variables are generated for array-type variables
forceSyncKernelCall	Enforce explicit synchronization for each kernel call
addCudaErrorCheckingCode	Add CUDA-error-checking code right after each kernel call
addSafetyCheckingCode	Add GPU-memory-usage checking code just before each kernel call

Table 4.6
Brief description of OpenMPC environment variables, which control program-level behaviors of data caching strategies.

Parameter	Description
shrdSclrCachingOnReg	Cache shared scalar variables onto GPU registers
shrdArryElmtCachingOnReg	Cache shared array elements onto GPU registers
shrdSclrCachingOnSM	Cache shared scalar variables onto GPU shared memory
prvtArryCachingOnSM	Cache private array variables onto GPU shared memory
shrdArryCachingOnTM	Cache 1-dimensional, R/O shared array variables onto GPU texture memory
shrdSclrCachingOnConst	Cache R/O shared scalar variables onto GPU constant memory
shrdArryCachingOnConst	Cache R/O shared array variables onto GPU constant memory

Table 4.7
Brief description of OpenMPC environment variables, which control tuning-related configurations.

Parameter	Description
cudaUserDirectiveFile=filename	Set the name of file containing user directives
extractTuningParameters=filename	Extract tuning parameters applicable to a given input program
genTuningConfFiles=directory	Generate sets of tuning configuration files and user-directive files
defaultTuningConfFile=filename	Set the name of file containing default CUDA tuning configurations
tuningLevel=N	Set tuning level (0: Program-level tuning 1: Kernel-level tuning)

4.3 Compilation and Tuning System for OpenMPC

This section presents a reference compilation and tuning system supporting OpenMPC. To realize the compilation system, we have modified the compiler proposed in

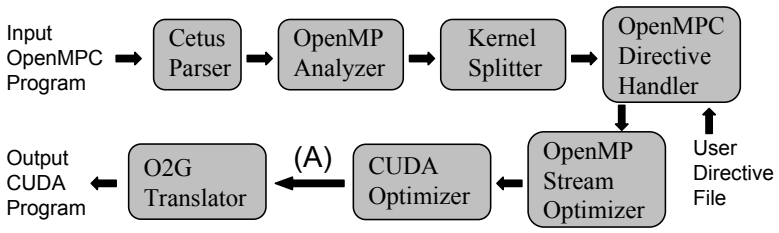


Fig. 4.1. Overall compilation flow. When the compilation system is used for automatic tuning, additional passes are invoked between *CUDA Optimizer* and *O2G Translator*, marked as **(A)** in the figure (See Figure 4.3)

chapter 2 by (1) separating the OpenMP-to-CUDA translator from the CUDA optimizer, (2) adding an OpenMPC directive handler, (3) implementing all the optimizations that had been applied manually, (4) implementing new transformation passes to perform the necessary code changes for each OpenMPC directive or environment variable, and (5) modifying existing optimization passes and the translator so they communicate with each other using the new directives.

The compiler also includes capabilities for tuning systems such as the one described in Section 4.3.3. These capabilities include a *search space pruner* and a *tuning configuration generator*.

Using the compilation system, we also created a prototype tuning system, which builds an optimization search space with applicable optimizations by analyzing the program and optional user settings. It then creates a path through the space and generates output CUDA code for each point in the search space. For our experiments, we have chosen a simple approach that visits each point in the space; that is, it exhaustively searches the space.

4.3.1 Overall Compilation Flow

Figure 4.1 shows the overall flow of the compilation. The *Cetus Parser* reads the input OpenMPC program and generates an internal representation (Cetus IR). The

OpenMP Analyzer recognizes standard OpenMP directives and analyzes the program to find all OpenMP *shared*, *threadprivate*, *private*, and *reduction* variables that are explicitly and implicitly used in each parallel region. The analyzer also identifies implicit barriers by OpenMP semantics and adds explicit barrier statements at each implicit synchronization point. The *Kernel Splitter* divides parallel regions at each synchronization point to enforce synchronization semantics under the CUDA programming model. The *OpenMPC-directive Handler* annotates each *kernel region* with an *ainfo* directive to assign a unique ID and parses a user directive file, if present. The handler also processes possible OpenMPC directives present in the input program. The *OpenMP Stream Optimizer* transforms traditional CPU-oriented OpenMP programs into OpenMP programs optimized for GPGPUs, and the *CUDA Optimizer* performs CUDA-specific optimizations. Both optimization passes express their results in the form of OpenMPC directives in the Cetus IR. In the last pass, the *O2G Translator* performs the actual code transformations according to the directives provided either by a user or by the optimization passes. Figure 4.2 shows a compilation example; Figure 4.2 (a) is an input OpenMP program, and Figure 4.2 (b) is an output OpenMP code after various optimization passes are executed (at a point *(A)* in Figure 4.1). The OpenMP code in Figure 4.2 (b) contains two kernel regions as a result of *Kernel Splitter*, and each kernel region has explicit OpenMP *shared* clause and *private* clause due to *OpenMP Analyzer*. Moreover, each kernel region is annotated with a set of OpenMPC clauses (line 3 through 5 and line 12 through 14 in Figure 4.2 (b)), which were added as outputs of various optimizations explained in Section 2.4.

4.3.2 Compiler Support for Tuning

The OpenMPC system supports a rich set of directives and environment variables controlling the automatic translation and optimization. This set can be used as the basis of a tuning system. The *search space pruning* and *tuning configuration*

```
1  int main (int argc, char *argv[]) {
2      int i; float sum = 0.0F;
3      #pragma omp parallel
4      {
5          #pragma omp for
6          for (i = 0; i < 4096; i++)
7              c[i] = a[i] + b [i];
8          #pragma omp for
9          for(i = 0; i < 2048; i++)
10             d[i] = c[i] + c[4095 - i];
11     }
12     ...
13 }
```

(a) Input OpenMP code

```
1  int main(int argc, char * argv[]) {
2      int i; float sum = 0.0F;
3      #pragma cuda ainfo kernelid(0) procname(main)
4      #pragma cuda gpurun nocudafree(a, b, c) nog2cmemtr(a, b, c) noc2gmemtr(c)
5      #pragma cuda gpurun constant(a, b)
6      #pragma omp parallel shared(a, b, c) private(i)
7      {
8          #pragma omp for nowait
9          for(i = 0; i < 4096; i++)
10             c[i] = a[i] + b[i];
11     }
12     #pragma cuda ainfo kernelid(1) procname(main)
13     #pragma cuda gpurun noc2gmemtr(c, d) nog2cmemtr(c) texture(c)
14     #pragma cuda gpurun nocudamalloc(c) nocudafree(c, d)
15     #pragma omp parallel shared(c, d) private(i)
16     {
17         #pragma omp for nowait
18         for(i = 0; i < 2048; i++)
19             d[i] = c[i] + c[4095 - i];
20     }
21     ...
22 }
```

(b) OpenMP code after optimization passes are applied

Fig. 4.2. Compilation example where kernel regions are annotated with OpenMPC clauses as results of various optimization passes

generation functions serve that purpose. The system allows user input for certain aggressive optimizations.

Search Space Pruning

Each of the new directives and environment variables that OpenMPC supports controls either an optimization or a thread batching for a kernel execution. A complete optimization search space consists of all possible combinations of values of these directives and environment variables. For automatic tuning, only the clauses in Table 4.2 and Table 4.3 and the environment variables in Table 4.5 and Table 4.6 are used as tuning parameters. Clauses in Table 4.4 have a predictable effect – they are used either by a user or by the translator internally, and variables in Table 4.7 are used to control various tuning-related configurations.

Because there are many directives and environment variables, the complete optimization space cannot be feasibly searched. Non-trivial CUDA programs contain many kernel functions, each of which can be controlled with the directive set individually. The *automatic search space pruning* function attempts to reduce this optimization space to a feasible size.

First, the *search space pruner* analyzes conditions necessary for applying each optimization and checks whether a given program has code sections satisfying the conditions. If no eligible code section is found, the optimization is removed from the optimization space. Second, the *pruner* suggests applicable caching methods for each variable that exhibits locality, based on the caching strategies described in Section 2.4.2.

The *search space pruner* may not be able to analyze the applicability of all parameters (e.g., *cudaMemTrOptLevel* and *assumeNonZeroTripLoops* in Table 4.5) because the analysis may be too complex or sensitive to runtime inputs (i.e., unsafe). The pruner reports these parameters. In response, a user may decide and express the validity of these parameters in the *optimization-space-setup*, described next.

Table 4.8
Optimization-space-setup file format

defaultGOptionSet([parameter [,] parameter]...])
excludeGOptionSet([parameter [,] parameter]...])
cudaMemTrOptLevel=N
cudaMallocOptLevel=N
UEPRemovalOptLevel=N
cudaThreadBlockSet([block_size [,] block_size]...])
maxnumofblockSet([num_of_blocks [,] num_of_blocks]...])

Tuning Configuration Generation

Once a search space is defined by the search space pruner, the *configuration generator* creates tuning configuration files for each point in the search space. The configuration files are fed to the O2G translator, one at a time, generating output CUDA code. By default, the configuration generator builds tuning configurations for *program-level tuning*. Using an OpenMPC environment variable (*tuningLevel* in Table 4.7), a user can choose the more exhaustive *kernel-level tuning*.

To further prune the search space, the user can provide an *optimization-space-setup* file containing parameters that should or should not be part of the optimization search space. These settings can direct the tuning system to choose aggressive optimizations, which otherwise might be unsafe. Additionally, the setup file may contain the value ranges of important parameters such as thread block size and the number of thread blocks. The format of the *optimization-space-setup* file is shown in Table 4.8, where a parameter can be one of OpenMPC environment variables in Table 4.5 and Table 4.6. The setup file may contain multiple *defaultGOptionSets* and *excludeGOptionSets*, and if the same parameter exists in both sets, *excludeGOptionSet* has a priority over the other.

4.3.3 Prototype Tuning System

Using the described search space functions, we have created a prototype tuning system, shown in Figure 4.3. The overall tuning process is as follows:

- The *search space pruner* analyzes an input OpenMPC program plus optional user settings, which exist as annotations in the input program, and suggests applicable tuning parameters.
- The *tuning configuration generator* builds a search space, further prunes the space using the *optimization space setup file* if user-provided, and generates tuning configuration files for the given search space.
- For each tuning configuration, the *O2G translator* generates an output CUDA program.
- The tuning engine produces executables from the generated CUDA programs and measures the performance of the CUDA programs by running the executables.
- The tuning engine decides a direction to the next search and requests the *configuration generator* to generate new configurations.
- The last three steps are repeated, as needed.

In the example tuning framework, a programmer can replace the tuning engine with any custom engine; all the other steps from finding tunable parameters to complex code changes for each tuning configuration are automatically handled by the proposed compilation system. In our prototype, we have developed a simple tuning engine, which performs exhaustive search. Tuning with an exhaustive search algorithm is feasible for our benchmarks, because the automatic search-space pruner can effectively reduce the optimization search. Our search engine simply consists of a

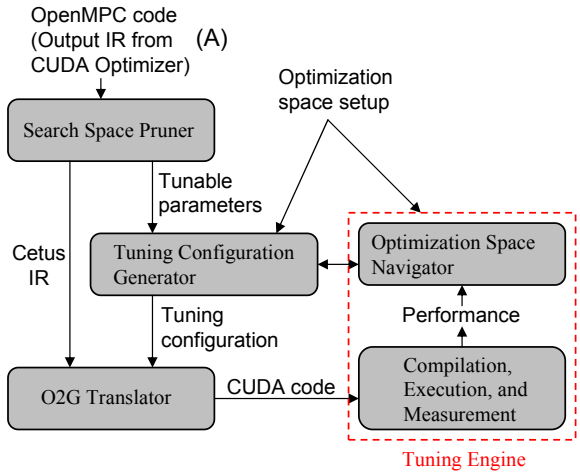


Fig. 4.3. Overall tuning framework. In the figure, input OpenMPC code is an output IR from CUDA Optimizer in the compilation system (See Figure 4.1)

script that compiles CUDA codes for each configuration, runs executables, and measures their performance. Several algorithms for more efficient search space navigation exist [18, 56]; they could replace exhaustive search in our system.

4.4 Evaluation

To demonstrate the effectiveness of OpenMPC, we have conducted two types of performance tuning experiments, using the prototype tuning framework: *profile-based tuning (ProfT)* and *user-assisted tuning (UAT)*. In profile-based tuning (*ProfT*), a target program is tuned with a *training* input data set – the smallest available set, in our case; the tuning system finds the best variant for the training input, and then the best variant is used to execute and measure the program with the actual data sets of interest (referred to as *production* data). The profile-based tuning is fully automatic.

User-assisted tuning (*UAT*) is used to obtain an upper performance bound of our tuning system. The programs have been tuned for each production data set. In addition, the user assists the tuning system by confirming the applicability of aggressive optimizations. The other tuning processes are performed automatically.

In these experiments, fourteen OpenMP programs (two kernel benchmarks (*JACOBI* and *SPMUL*), three NAS OpenMP Parallel Benchmarks (*EP*, *CG*, and *FT*), and nine Rodinia Benchmarks [9] (*BACKPROP*, *BFS*, *CFD*, *HEARTWALL*, *HOTSPOT*, *KMEANS*, *LUD*, *NW*, and *SRAD*)) were automatically transformed and tuned. Our system is able to handle a larger class of programs, with some limitations such as the one mentioned in Section 2.5.3. The translator produces appropriate warnings for unsupported program patterns.

For comparison, three types of code variants of the tested programs were also evaluated: *Base*, *AllOpt*, and *Manual* versions. *Base* means CUDA programs translated by the proposed system without any optimization, *AllOpt* refers to the code variants where all safe optimizations are applied automatically by the compiler, and *Manual* represents hand-written CUDA versions. In creating the manual CUDA versions of the tested programs that do not have corresponding CUDA versions (*JACOBI*, *SPMUL*, *EP*, and *CG*), we have also used OpenMPC; we have first annotated each OpenMP source program using the OpenMPC directives and generated CUDA programs with our translator. We have then applied additional manual transformations to the generated CUDA programs, as possible. Creating these hand-coded reference code versions consumed substantial time.

The tested GPU device is an NVIDIA Quadro FX 5600 GPU, which has 16 multiprocessors (SMs) clocked at 1.35 GHz and 1.5 GB of DRAM. Each SM consists of 8 SIMD processing units (SPs) and has 16 KB of shared memory. The host CPU is a 3-GHz AMD dual-core processor with 12 GB DRAM. The translated CUDA programs were compiled using the NVIDIA CUDA Compiler (NVCC) and the serial versions of the input OpenMP programs were compiled using the GCC compiler version 4.2.2, with option *-O3*.

Table 4.9 summarizes the performance improvements achieved by the tested tuning systems. Overall, we found the followings:

- User-assisted tuning using the described system increases the performance up to 4.23 times (1.19 times on average) over the un-tuned versions (*AllOpt*). When

Table 4.9
Overall tuning performance. *Translator Input* refers to the types of the translation input sources; *Modified OpenMP* means that the input OpenMP code is manually modified before fed to the translator. *All-Opt* versions are the ones where all safe optimizations are automatically applied by the compiler. In A(B) format, B refers to the performance when the results of *LUD*, which shows the most significant performance gap between tuned and manual versions, are excluded.

Translator Input	Performance Improvement over All-Opt Versions			Relative Performance over Manual Versions		
	MIN	MAX	AVG	MIN	MAX	AVG
Original OpenMP	1	4.23	1.19	0.02 (0.03)	1.92 (1.92)	0.5 (0.58)
Modified OpenMP	1	7.71	1.24	0.02 (0.33)	2.68 (2.68)	0.75 (0.92)

the input OpenMP programs were optionally modified by hand (we believe that most of changes can be done automatically by an advanced compiler), we could improve the performance up to 7.71 times (1.24 times on average) over the untuned versions, and the average performance gap between hand-written versions (*Manual*) and versions generated by our tuning system (*ModUAT*) becomes less than 25%. If we exclude one exceptional case (Rodinia Benchmark *LUD*), the average performance gap will be less than 8%.

- The proposed search-space pruner is able to reduce the optimization search space effectively (98.7% on average).

- In some programs, profile-based tuning is highly sensitive to input data, motivating future work in runtime tuning methods.

The detailed results are presented in the following sections.

4.4.1 Optimization Space Reduction

Table 4.10 lists the number of applicable tuning parameters suggested by the search-space pruner, and Table 4.11 shows the optimization search space reduction

Table 4.10
Number of parameters suggested by the search-space pruner and the number of kernel regions when the original OpenMP programs are translated. In $A/B/C$ format, A is the number of tunable program-level parameters, B is the number of parameters that the pruner suggests to be always beneficial, and C is the number of parameters that a user's approval is required.

Benchmark	Program-level Parameter	Kernel-level Parameter	# of kernel regions
JACOBI	3/4/1	1	2
SPMUL	4/3/2	4	2
EP	5/3/2	3	1
CG	8/3/2	5	19
FT	9/4/2	6	21
BACKPROP	6/4/2	4	4
BFS	6/3/2	4	2
CFD	8/3/2	5	5
HEARTWALL	8/4/2	5	1
SRAD	9/3/2	7	2
HOTSPOT	8/3/2	6	2
KMEANS	7/4/2	4	1
LUD	5/3/2	4	2
NW	4/3/2	3	2

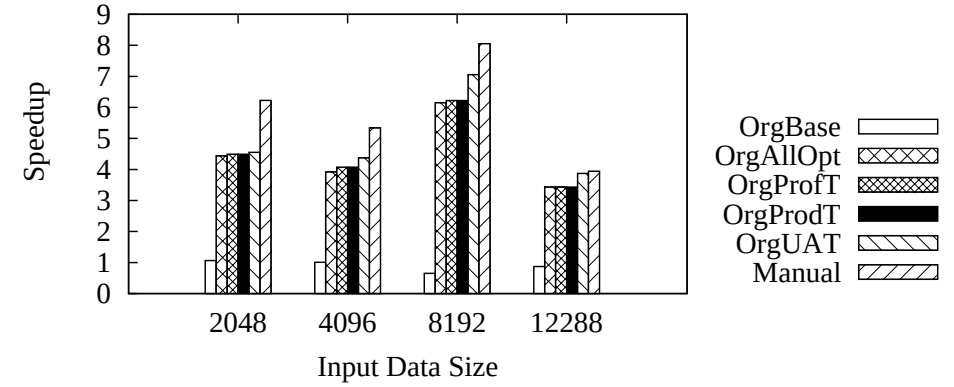
due to pruning. Aggressive parameters are pruned, unless the user confirms their validity. In all experiments, we have used program-level tuning. For programs with small number of kernels, kernel-level tuning would be feasible as well, despite our simple, exhaustive search engine. we have verified that the performance of both methods are nearly equal for those small programs. Applying kernel-level tuning in programs with many kernels, such as *CG* and *FT*, would increase the search space significantly, motivating future work in advanced search space navigations [18, 56].

Table 4.11
Optimization search space reduction by the search-space pruner for program-level tuning when the original OpenMP programs are translated

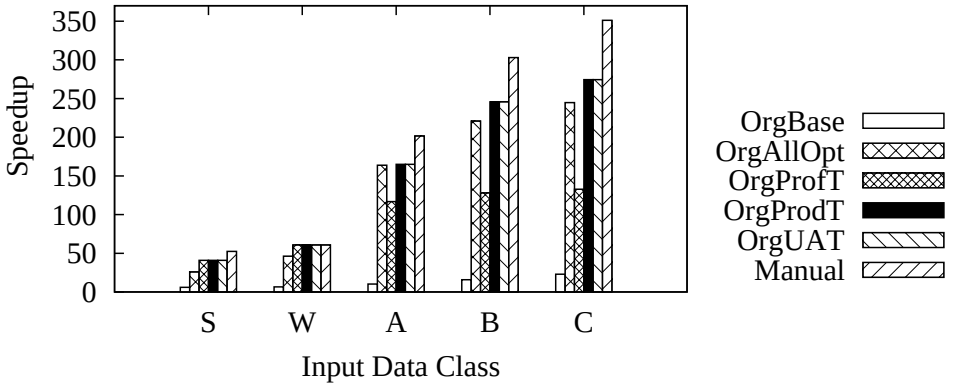
Benchmark	Number of Tuning Configurations		Search Space
	W/O pruning	W/ pruning	Reduction (%)
JACOBI	25600	100	99.61
SPMUL	16384	128	99.22
EP	21504	336	98.44
CG	6144	384	93.75
FT	16384	512	96.88
BACKPROP	65536	128	99.8
BFS	49152	96	99.8
CFD	49152	384	99.22
HEARTWALL	65536	512	99.22
SRAD	65536	1024	98.44
HOTSPOT	16384	256	98.44
KMEANS	65536	256	99.61
LUD	49152	48	99.9
NW	65536	32	99.95

4.4.2 Tuned Performance of JACOBI

JACOBI is a stencil computation kernel used in many regular scientific applications, such as partial differential equation solvers. Even though *JACOBI* has a simple, regular access pattern, the base-translated GPU code performs poorly due to un-coalesced global memory accesses (*OrgBase* in Figure 4.4(a)). Our translator changes the access patterns to coalesced ones (*OrgAllOpt* in Figure 4.4(a)). The results of profile-based tuning are shown as *OrgProfT* in Figure 4.4(a). User-assisted tuning (*OrgUAT* in the figure) shows the best performance that the proposed tuning system can achieve. To examine the effect of aggressive optimizations applied under the user's approval, the figure also includes the performance of production-



(a) JACOBI Kernel



(b) NAS Parallel Benchmark EP

Fig. 4.4. Performance of *JACOBI* and *EP* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pretuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *Manual* is the manually optimized version.

tuning-only versions (*OrgProdT* in the figure); the performance difference between *OrgProdT* and *OrgUAT* indicates the additional gain that can be achieved by applying unsafe, aggressive optimizations. Due to its simple structure, *JACOBI* does not have many tuning issues, except for thread batching. The performance gap between the manual version (*Manual*) and the system-tuned version (*OrgUAT*) is due

to a caching optimization using tiling transformation, which is not provided by our translation system.

4.4.3 Tuned Performance of EP

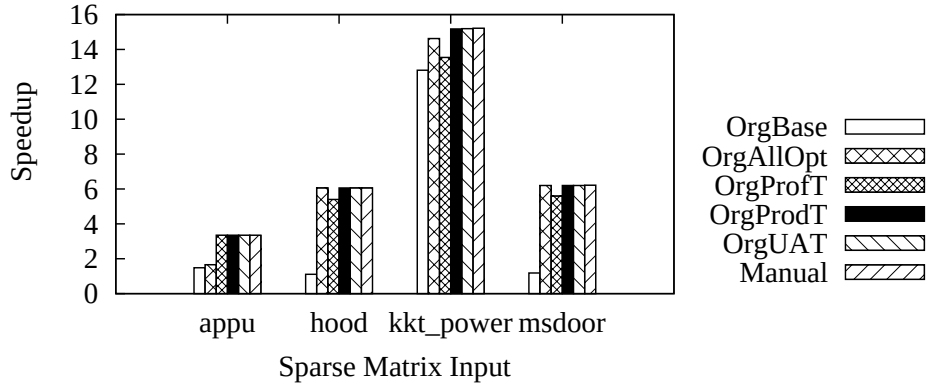
EP is a highly parallel application, which computes Gaussian deviates using pseudo-random numbers. Despite its parallelism, the base-translated version of EP performs poorly (*OrgBase* in Figure 4.4(b)), which again is due to un-coalesced global memory accesses (details in Section 2.6.2). As in *JACOBI*, our translator removes this limitation (*OrgAllOpt* in Figure 4.4(b)). In the case of *EP*, profile-based tuning is not effective.

Our results indicate that the performance of some GPU applications is highly sensitive to the input data. In such cases, input-sensitive tuning systems, such as G-ADAPT [51], will perform better than profile-based tuning systems.

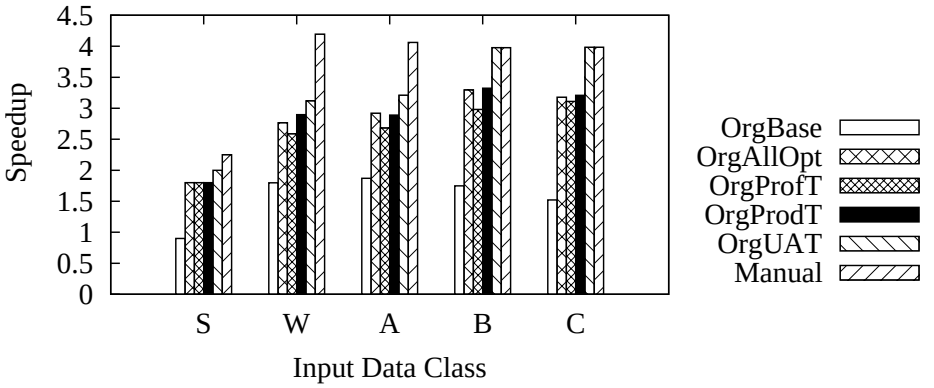
Our tuned programs (*OrgUAT* in Figure 4.4(b)) do not always include all caching optimizations. For example, the *private array caching* optimization allocates a private array in *shared memory* to reduce long latencies to the CUDA *local memory*. However, this optimization is implemented by expanding the private array in the *shared memory*, which puts pressure on this memory due to its small size. Our compiler could perform nearly same optimizations as the manual version (*Manual* in the figure), but due to the inefficiency in handling array reduction patterns, the system tuned versions (*OrgUAT*) show less speedups than the manual versions. An advanced array section analysis technique would be able to reduce this performance gap.

4.4.4 Tuned Performance of SPMUL

Sparse matrix computation is used in many scientific applications. *SPMUL* and *CG* are two important irregular programs performing sparse matrix computation. Sparse matrix computations tend to exhibit irregular computation and communication behavior; our results in Figure 4.5(a) show that profile-based tuning is not very



(a) SPMUL Kernel



(b) NAS Parallel Benchmark CG

Fig. 4.5. Performance of *SPMUL* and *CG* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pretuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *Manual* is the manually optimized version.

successful. One interesting point about *SPMUL* is that none of the tuned program variants for any input had *loop collapsing* applied (details in Section 2.6.3), even though this optimization was selected by most of the tuned variants of *CG*. *Loop collapsing* enables coalesced accesses to global memory by combining two nested sparse computation loops into one; additionally it caches shared data in the shared memory

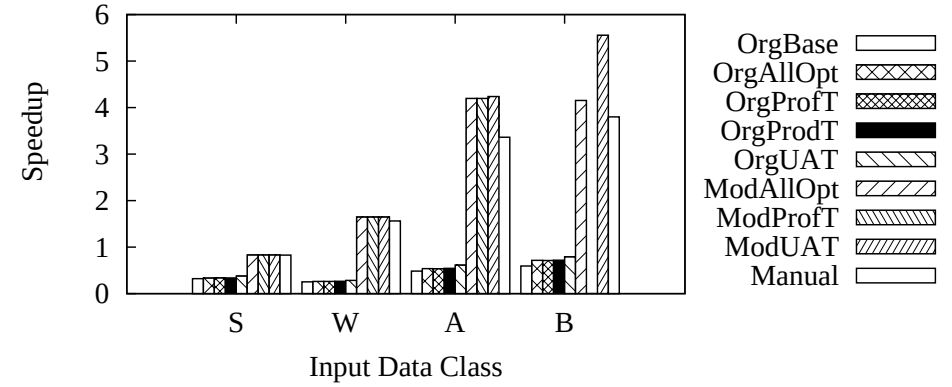
to reduce global memory accesses. However, the optimization increases the usage of shared memory and avoids exploiting the texture memory. Therefore, the overall benefit of the optimization is not statically predictable, making it amenable to tuning. In *SPMUL* case, the proposed translation system could perform the same optimizations as the manually optimized code (*Manual* in the figure), and thus the automatically tuned versions (*OrgUAT*) perform as equally as the manual version (*Manual*).

4.4.5 Tuned Performance of CG

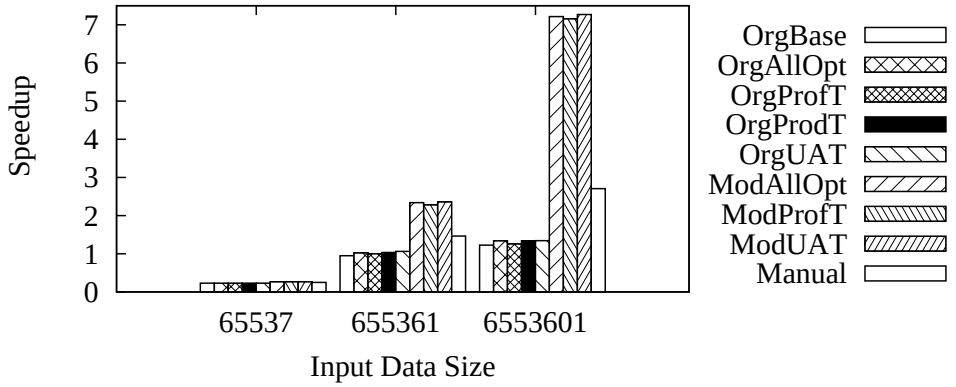
CG is a more challenging sparse matrix computation program. In *CG*, many kernel regions span across several procedures, resulting in complex memory transfer patterns between the CPU and the GPU. Interprocedural data flow analysis presented in Section 2.4.2 plays a key role in creating efficient memory transfer patterns (*OrgAllOpt* in Figure 4.5(b)). In *CG*, applying aggressive optimizations increases the overall performance (*OrgUAT*), since the aggressive optimizations augment the accuracy of CUDA memory-related optimizations. The GPU version of *CG* also shows input-sensitive performance behavior, and thus profile-based tuning was not effective (*OrgProfT* in Figure 4.5(b)). In *CG*, the manual version (*Manual*) applies more efficient GPU memory allocation and data-transfer schemes than the system-tuned version (*OrgUAT*), and the manual version also removes some of the implicit barriers, resulting in less kernel invocation overheads. This barrier removal is possible under the CUDA memory model, if two adjacent kernel regions are work-partitioned so that no two threads communicate with each other. The performance improvement by this manual overhead reduction is more pronounced for small input data sizes, as shown in Figure 4.5(b).

4.4.6 Tuned Performance of FT

FT solves a 3-D partial differential equation using the Fast Fourier Transform (FFT). The original OpenMP code is heavily optimized for the traditional, cache-



(a) NAS Parallel Benchmark FT



(b) Rodinia Benchmark BACKPROP

Fig. 4.6. Performance of *FT* and *BACKPROP* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pretuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *ModAllOpt*, *ModProfT*, and *ModUAT* apply the same techniques as *OrgAllOpt*, *OrgProfT*, and *OrgUAT* respectively, except that in *Mod*-versions, the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. *Manual* is the manually optimized version.

based architectures, and thus resulting memory access patterns do not allow enough opportunity for coalesced memory accesses when translated into GPU code (details in Section 2.6.5). Therefore, tuning the translation of the original OpenMP version does

not give any noticeable performance improvement, even with various tunable parameters, such as caching on registers, shared memory, texture cache, and constant cache (*OrgUAT* in Figure 4.6(a)). Changing the memory access patterns to GPU-friendly ones is difficult for the compiler to perform it automatically, due to its complexity.

ModAllOpt, *ModProfT*, and *ModUAT* in the figure refer to versions where the input OpenMP program is manually modified such that it has the same memory access patterns as the hand-written CUDA code (*Manual* in the figure). The modified versions allow several tuning opportunities for additional performance improvement; first, when the input data class is either *A* or *B*, thread batching affects the overall performance quite a lot (see Figure 2.26). Second, when the input data size is small, the best performance was achieved when all applicable caching optimizations were applied, even though the best thread batchings vary. As the input data size is getting bigger, however, not all caching optimizations were selected; when the input class is *B*, in-block-reduction-data caching optimization was not selected for the best configuration, even though caching of the in-block reduction data on shared memory is generally known to be beneficial. (More details on the in-block reduction caching can be found in Section 2.6.12.)

Another interesting thing in *FT* is that the profile-based tuning (*ModProfT*) does not work if the input data changes drastically; when the input class is *B*, compiling using the tuning configuration selected by the profile-based tuning failed, due to excessive memory usage. This shows another example where traditional, profile-based tuning does not work well.

4.4.7 Tuned Performance of BACKPROP

In *BACKPROP*, the main performance bottleneck is the uncoalesced memory access patterns caused by the layout of two-dimensional arrays. The uncoalesced access patterns may be fixed by *parallel loop-swap* technique proposed in Section 2.4.1, but the current compiler could not apply it automatically, due to the lack of advanced

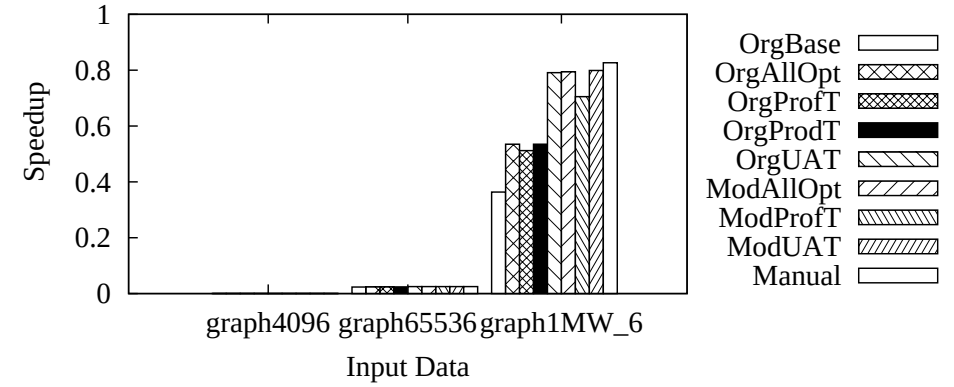
analysis technique. Figure 4.6(b) shows that fixing the uncoalesced access patterns manually (Mod in the figure) has significant performance impact. Even though the figure seems to say that both the profile-based tuning (*ProfT*) and user-assisted tuning (*UAT*) perform no better than the versions where all applicable optimizations are blindly applied (*AllOpt*), tuning is still necessary, since different input data demand different thread batching and optimizations for the best performance, and the performance of the *AllOpt* versions also varies according to the thread batching.

4.4.8 Tuned Performance of BFS

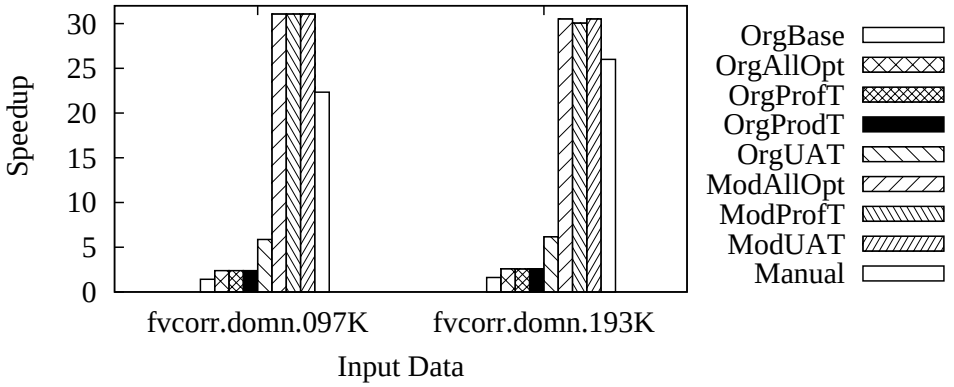
BFS performs a breadth-first search, which is one of commonly used graph algorithm. Even though it is a simple algorithm, it has irregular access patterns, and the amount of computation is small for the given input sets. Therefore, GPU memory allocation and memory transfer times are dominant, allowing little room for performance improvement through tuning. Figure 4.7(a) show the tuning performance; due to its irregular, memory-intensive nature, none of GPU versions performs better than the serial CPU versions.

4.4.9 Tuned Performance of CFD

CFD is an unstructured grid solver for the three-dimensional Euler equations for compressible flow. In *CFD*, applying aggressive optimizations to reduce redundant memory-transfers has noticeable performance effect (*OrgUAT* in Figure 4.7(b)). *CFD* allows various caching strategies, but thread batching is the most important tuning parameter, as shown in Figure 2.29. Figure 4.7(b) shows that there are big performance gap between the manual versions (*Manual*) and the automatically translated and tuned versions (*OrgUAT*). Uncoalesced memory accesses, which are difficult for the compiler to fix automatically, contributes to the performance gap mostly. The uncoalesced access problem can be fixed by modifying the input OpenMP program (*Mod* in the figure), and then the compiler-translated versions could perform better



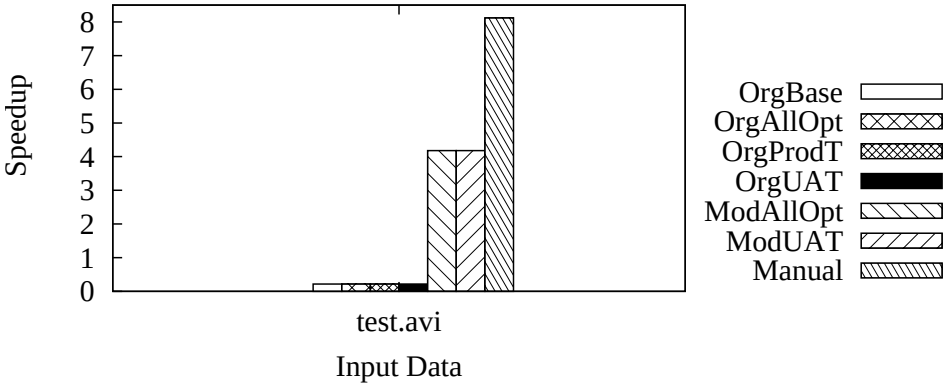
(a) Rodinia Benchmark BFS



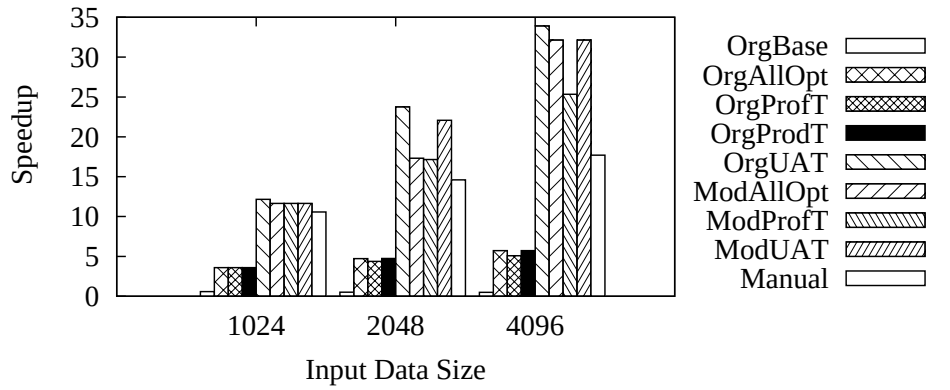
(b) Rodinia Benchmark CFD

Fig. 4.7. Performance of *BFS* and *CFD* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pretuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *ModAllOpt*, *ModProfT*, and *ModUAT* apply the same techniques as *OrgAllOpt*, *OrgProfT*, and *OrgUAT* respectively, except that in *Mod*-versions, the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. *Manual* is the manually optimized version.

than the hand-written CUDA codes, when the thread batching is properly tuned (*ModAllOpt*, *ModProfT*, and *ModUAT*).



(a) Rodinia Benchmark HEARTWALL



(b) Rodinia Benchmark SRAD

Fig. 4.8. Performance of *HEARTWALL* and *SRAD* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pre-tuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *ModAllOpt*, *ModProfT*, and *ModUAT* apply the same techniques as *OrgAllOpt*, *OrgProfT*, and *OrgUAT* respectively, except that in *Mod*-versions, the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. *Manual* is the manually optimized version.

4.4.10 Tuned Performance of HEARTWALL

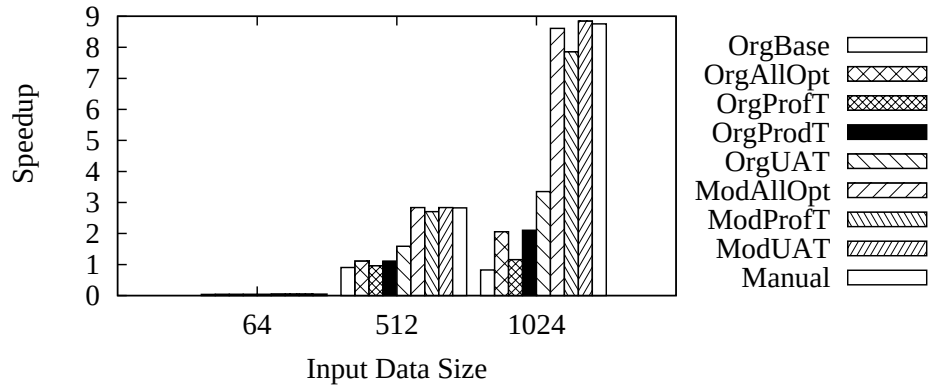
HEARTWALL is a program to track the movement of a mouse heart in response to the stimulus. The main part of the program consists of multiple nested loops,

and if we parallelize the outmost loop, as done in the original OpenMP version, the translated code suffers from control flow divergences and uncoalesced memory access problems (*OrgUAT* in Figure 4.8(a)). To exploit abundant GPU computing powers and to reduce uncoalesced memory access problems, the manual versions (*Manual* in Figure 4.8(a)) use a complex thread batching scheme, where each iteration of the outmost loop processing sample points is assigned to a thread block, and iterations in the inner loops computing each pixel are mapped to threads in a thread block. We could get the similar effect by using the OpenMP *collapse* clauses,and *Mod* in the figure presents the speedups when the modified OpenMP program is translated and tuned.

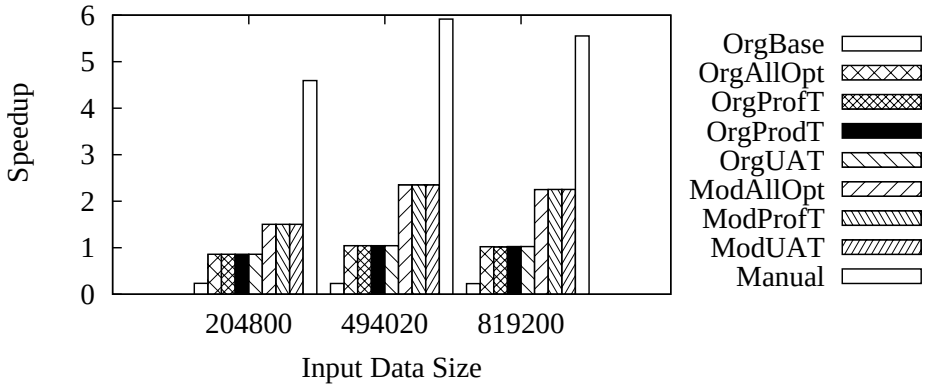
In the modified, translated versions (*Mod*), there are complex interactions among the limited hardware resources, and thus thread batching can change the overall performance drastically, depending on the applied optimizations (see Figure 2.30). The performance gap between *ModUAT* and *Manual* in the figure is largely due to the difference in handling synchronizations, and more details can be found in Section 2.6.9.

4.4.11 Tuned Performance of SRAD

SRAD is a diffusion algorithm based on partial differential equations and used to remove the speckles in ultrasonic and radar imaging applications. Figure 4.8(b) shows the tuning performance on *SRAD*. The performance of *OrgUAT* reveals that applying unsafe, aggressive optimizations under the user's approval can increase the performance quite a lot. *SRAD* is a representative case, which shows that automatic optimizations combined with tuning (*OrgUAT*) can beat the performance of the manually optimized versions (*Manual*). More detailed comparison between the automatic versions and the manual versions can be found in Section 2.6.10.



(a) Rodinia Benchmark HOTSPOT



(b) Rodinia Benchmark KMEANS

Fig. 4.9. Performance of *HOTSPOT* and *KMEANS* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pre-tuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *ModAllOpt*, *ModProfT*, and *ModUAT* apply the same techniques as *OrgAllOpt*, *OrgProfT*, and *OrgUAT* respectively, except that in *Mod*-versions, the input OpenMP program was manually modified in a GPU-friendly way before fed to the translator. *Manual* is the manually optimized version.

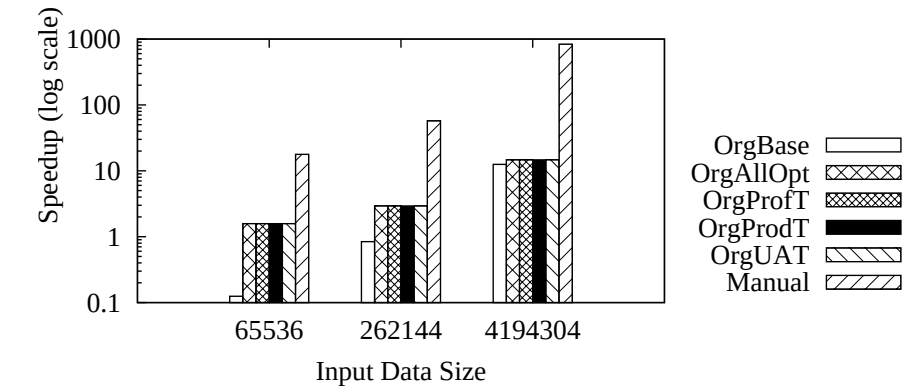
4.4.12 Tuned Performance of HOTSPOT

HOTSPOT is a thermal simulation tool used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The core part

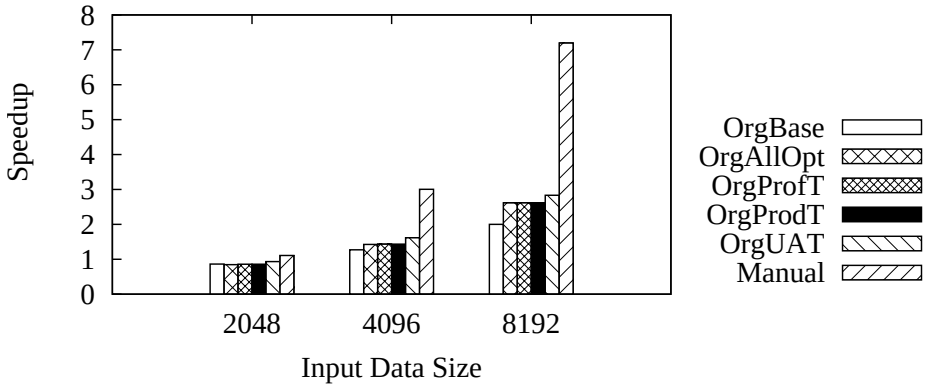
of *HOTSPOT* contains two nested loops, where the outmost loops are parallelized but the inner loops are also parallelizable. The automatically translated versions of the original OpenMP code (*Org* in Figure 4.9(a)) parallelize the outmost loops as OpenMP does, but the iteration spaces of the parallel loops are not big enough to create threads to hide the long global memory access latency. To increase the number of threads, the manual versions (*Manual* in the figure) apply a two-dimensional thread mapping to exploit the nested parallel loops. *Mod* in the figure shows the performance when we manually inserted the OpenMP *collapse* clauses to the nested parallel loops in the original OpenMP code, such that translated kernels can be executed by larger number of threads due to the increased iteration space. In tuning *HOTSPOT*, thread batching is an important tuning parameter; in the original versions where the outmost loops are parallelized, the best thread block size was either 32 or 64, but in the modified versions where the nested parallel loops are collapsed, bigger thread block sizes (64 or 128) were chosen. Moreover, due to the different thread batching preference depending on the input size, the profile-based tuning (*OrgProfT* and *ModProfT*) does not work well as the input size gets bigger.

4.4.13 Tuned Performance of KMEANS

KMEANS is a well-known clustering algorithm used for various data-mining applications. In *KMEANS*, thread batching is the most important tuning parameter, but other optimizations such as caching on texture cache or unrolling on the reduction are also selectively chosen depending on the inputs. Even with the user-assisted tuning (*ModUAT*), there exist big performance gaps between the manual versions (*Manual*) and the system-tuned versions. We attribute the performance gap to the differences in implementing array reductions (details in Section 2.6.12).



(a) Rodinia Benchmark LUD



(b) Rodinia Benchmark NW

Fig. 4.10. Performance of *LUD* and *NW* (Speedups are over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need a user's approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pretuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under the user's approval. *Manual* is the manually optimized version.

4.4.14 Tuned Performance of LUD

LUD is a simple matrix decomposition tools, whose main computation consists of two simple parallel loops. *LUD* does not have enough tuning issues; the performance effect of the thread batching is small, and there exist only a few applicable optimizations. However, *LUD* is a special case, since algorithmic changes special-

ized for the underlying GPU memory model (details in Section 2.6.13) can achieve surprisingly high speedup (*Manual* in Figure 4.10(a)), while there is little room for tuning. The large performance gap reveals that for some applications like *LUD*, customized algorithms considering the underlying architectures are essential for the optimal performance.

4.4.15 Tuned Performance of NW

Needleman-Wunsch (*NW*) is a global optimization method for DNA sequence alignment. Due to its simple structure, *NW* does not have enough tuning issues, and tuning experiments show that the same tuning configuration was selected for the best one regardless of the input size. Figure 4.10(b) presents the overall tuning performance; the performance gap between the manual versions (*Manual*) and the system-tuned versions (*OrgUAT*) is mainly caused by the lack of automatic tiling transformation technique in the automatic translation system.

4.5 Related Work

Several automatic translation techniques have been proposed with the goal of increasing the productivity of CUDA programming. In the hiCUDA directive-based language [14], a set of directives express CUDA computation and data attributes in a sequential program. This work is similar to ours in that it uses directives to provide abstractions of CUDA, and a compiler automatically generates CUDA code by interpreting these directives. However, hiCUDA uses the same programming paradigm as CUDA; even though it hides the CUDA language syntax, the complexity of the CUDA programming and memory model is directly exposed to programmers. OpenMPC is based on OpenMP, which is higher-level than hiCUDA, and thus our work provides better programmability than hiCUDA. Moreover, hiCUDA does not provide any optimization, whereas our framework supports various automatic performance optimizations. hiCUDA can complement our work, as its exposure to the CUDA

model offers the potential of finer-grain control over GPUs' performance. CUDA-lite [20] is another directive-based approach, which generates code for optimal tiling of global memory data. CUDA-lite is limited in that it supports automatic code translation only on existing CUDA programs. The approach can also complement our work by providing advanced tiling transformations for optimized global memory accesses; currently, the OpenMPC compiler performs tiling optimization only for work partitioning. Another approach [52] has developed an automatic code transformation system that generates CUDA code from sequential C source code, for affine programs. This approach uses a polyhedral compiler model to find affine transforms for optimizing data movement between CUDA off-chip and on-chip memories. By contrast, our compiler framework optimizes both regular and irregular programs and supports optimizations to minimize data movement between the CPU and the GPU, as well as the ones for efficient global memory accesses.

There have been many studies on optimizing the performance of CUDA-based GPGPU programs; Ryoo and his colleagues proposed optimization space pruning techniques [18]. Their techniques use a model-based approach and work well if global memory bandwidth is not a performance bottleneck. By contrast, our pruning algorithm reduces the search space by checking the applicability of each optimization. Their techniques can augment our framework by providing further pruning when the assumption holds, and our framework can also complement their work by automating their manual code conversions.

To automatically optimize the performance of CUDA programs, most previous work was application-specific; Datta et al. [54] developed a number of optimization strategies and an auto-tuning environment for stencil computations. Nukada et al. [53] presented an auto-tuning algorithm for 3-D FFT, and Volkov et al. [55] conducted an extensive study of dense linear algebra, using auto-tuning techniques to achieve the best performance. Unlike the previous contributions, G-ADAPT [51] uses a compiler-based, adaptive framework, which automatically searches the best optimizations for a general GPU program on different input data sets. This work is the closest to

ours; G-ADAPT performs program transformations and optimization space search automatically, and offers a set of directives for programmers to specify search criteria. However, the adaptive framework works on a small subset of the optimization space, and thus the framework performs automatic transformation in limited ways. Our work is complementary to this work in that our compiler framework can offer a richer set of transformations and optimizations, and also support a larger number of directives that provide control over translation and optimization parameters than G-ADAPT; by using our translator, G-ADAPT could extend its optimization space. G-ADAPT limitation of working only on existing GPU programs could be relaxed by adopting our OpenMPC API as a front-end programming model.

4.6 Summary

This chapter describes a new programming interface, called OpenMPC, which consists of standard OpenMP and a new set of compiler directives and environment variables, extended for CUDA. OpenMPC addresses two important issues on GPGPU programming: *programmability* and *tunability*. OpenMPC as a front-end programming model provides programmers with abstractions of the complex CUDA programming model and high-level control over various optimizations and CUDA-related parameters. We have developed a fully automatic compilation and user-assisted tuning system, which is able to suggest applicable tuning configurations for an input OpenMP program, generate CUDA code variants for each tuning configuration, and search the best optimizations for the generated CUDA program automatically. Experiments on both regular and irregular programs demonstrate that the proposed system achieves performance improvements comparable to hand-coded CUDA for various applications. However, large performance gap between the manual versions and the system-tuned versions in some applications also reveals that further extensions to express CUDA-specific execution model and memory model may be needed to fully exploit the abundant computing power of GPGPUs.

5. EPILOGUE

5.1 Conclusions

This dissertation has investigated compile-time and runtime techniques to improve programmability and to enable dynamic adaptation of parallel applications in modern, complex architectures such as GPGPUs.

First, this dissertation examines the feasibility of exploiting OpenMP shared memory programming model on stream architectures such as GPGPUs. We have described a compiler framework for translating standard OpenMP shared-memory applications into CUDA-based GPGPU applications. The goal of the proposed translation is to provide a better programming model for general computing on GPGPUs. By using OpenMP as a frond-end programming model, the proposed system provides enough abstraction to hide the complex programming and memory models of the underlying GPGPUs, while exploiting the abundant computing power by converting the loop-level parallelism of the OpenMP programming model into the data parallelism of the GPGPU programming model. To reduce the performance gaps caused by architectural differences between traditional shared-memory multiprocessors served by OpenMP and stream architectures adopted by most GPUs, several key transformation techniques, which enable efficient GPU global memory accesses, are developed. Experiments on fourteen OpenMP programs in diverse domains demonstrate that the described translator and compile-time optimizations work well on both regular and irregular applications, leading to performance improvements of up to 68X over the unoptimized translation (up to 245X over serial on the CPU). However, large performance variations according to the different execution configurations and optimizations also reveal that fine-grained tuning may be necessary to achieve optimal performance.

Second, this dissertation studies runtime tuning systems to address performance variation issues mentioned above. In preliminary work, an adaptive runtime tuning system with emphasis on parallel irregular applications has been proposed. The proposed system deals with two important tuning issues: computational load balancing and communicational load balancing. Our adaptive mapping system solves computational load-balancing problems by dynamically re-assigning iterations to processes, and our runtime selection system fixes the communicational load-balancing problem by choosing the best communication method for the given data distribution. The preliminary experiments on well-known sparse matrix-vector (SpMV) multiplication kernels, with 26 real sparse matrices from various scientific and engineering applications, show that the proposed tuning system reduces execution time up to 68.8% (30.9% on average) over a base parallel SpMV algorithm on distributed systems with 32 nodes.

Third, this dissertation studies the possibility of building an integrated framework where both the compiler framework and the runtime tuning system are synergistically combined, such that standard OpenMP applications are automatically translated, transformed, and tuned for the execution on the underlying GPGPU architectures. For this goal, a new API, called OpenMPC, has been proposed, which provides high-level control over important CUDA-related parameters and optimizations. We have developed a fully automatic, parameterized compilation and user-assisted tuning system supporting OpenMPC. Evaluation of the new system using the fourteen OpenMP programs presents that user-assisted tuning using OpenMPC with optional manual modification on the input OpenMP codes, which can be automated with more advanced compiler techniques, can improve the performance up to 7.71 times (1.24 times on average) over un-tuned versions, which is 75% of the performance of hand-written CUDA versions. The evaluation reveals that there are a few exceptional cases, where the performance gaps between the automatically translated and tuned versions and the manual versions are quite high. (If we exclude the most exceptional case, the average performance gap will be reduced from 25% to 8%.) To reduce the perfor-

mance gaps, the proposed API should be further extended to express CUDA-specific execution model and memory model directly. The evaluation also reveals that the performance of some applications are highly sensitive to input data, and thus traditional profile-based tuning does not work well on those applications. Possible approaches to address these issues are presented in the following, last section.

5.2 Future Work

While this dissertation provides an important infrastructure to study how high-level programming models such as OpenMP can be adapted onto new architectures such as GPGPUs, it opens up new possibilities for future research in this area. To conclude this dissertation, we discuss two interesting future directions.

5.2.1 High-level Extension to Support GPU-specific Execution Model and Memory Model

OpenMPC pursues two apparently conflicting goals: programmability and performance. By providing high-level control over important CUDA-related parameters and optimizations as well as supporting tuning tools, OpenMPC could achieve performance comparable to the hand-written CUDA codes on various applications. However, there exist some exceptional cases, such as Rodinia Benchmark *LUD*, *HEARTWALL*, and *KMEANS*, where OpenMPC performs much worse than the manual versions. The main reason for the large performance gap is because OpenMPC does not provide a direct way to express CUDA-specific execution model and memory model; to efficiently adapt these applications onto CUDA GPGPUs, minimizing synchronization overhead using a built-in CUDA synchronization library (*__syncthreads()*) and exploiting CUDA *shared memory* for coalesced global memory access are essential, but none of these CUDA-specific features are expressible in the OpenMP programming model, where OpenMPC is based. (Exploiting other CUDA-specific memories, such as *texture memory* and *constant memory*, can be efficiently derived from OpenMPC,

but exploiting CUDA shared memory is limited.) Multi-dimensional thread batching is also necessary for efficient work partitioning in those applications. Even though OpenMPC provides an indirect way to mimic the multi-dimensional work partitioning (refer to Section 2.6.9), high-level abstraction prevents complex manipulation of the thread-to-data mapping. Enabling these CUDA-specific features at high-level will extend the performance coverage of OpenMPC.

Table 5.1
Brief description of new OpenMPC clauses, which can be used to express multi-dimensional thread batching or explicit shared memory access

Clause	Description
threadbatching(numGridDims, numBlockDims)	Set the numbers of dimensions for a grid of thread blocks and each thread block
cudashared(list)	Set a list of variables to be allocated on CUDA shared memory
cudasyncthreads	Represent a CUDA *__syncthreads()* call

A possible OpenMPC extension and an example are shown in Table 5.1 and Figure 5.1, respectively. In Table 5.1, an OpenMPC clause *threadbatching* specifies the number of dimensions for a grid of thread blocks (*numGridDims*) and each thread block (*numBlockDims*) for an attached kernel region. The attached region should be a nested for-loop whose depth should be no less than the sum of the two dimension numbers (*numGridDims* + *numBlockDims*). The outmost *numGridDims* loops (line 5 and 6 in Figure 5.1) will be mapped to *numGridDims*-dimensional grid of thread blocks, and the next *numBlockDims* outmost loops (line 7 and 8 or line 13 and 14 in the figure) will be mapped to *numBlockDims*-dimensional thread blocks. As shown in the example (Figure 5.1), the attached nested loop does not have to be a perfectly nested loop, but grid-partitioning loops (line 5 and 6 in the figure) and thread-block-partitioning loops (line 7 and 8 or line 13 and 14) should be perfectly nested, respectively. A *cudashared* clause indicates that the variables in the list

```
1  float c[16][16];
2  ...
3  #pragma cuda gpurun threadbatching(2,2) cudashared(c)
4  #pragma omp parallel for shared(a,b) private(bx,by,tx,ty)
5  for(by=0; by<GDIMSIZEY; by++) {
6      for(bx=0; bx<GDIMSIZEX; bx++) {
7          for(ty=0; ty<BDIMSIZEY; ty++) {
8              for(tx=0; tx<BDIMSIZEX; tx++) {
9                  c[ty][tx] = ...
10             }
11         }
12         #pragma cuda syncthreads
13         for(ty=0; ty<BDIMSIZEY; ty++) {
14             for(tx=0; tx<BDIMSIZEX; tx++) {
15                 ...
16             }
17         }
18     }
19 }
```

Fig. 5.1. Example OpenMP code where an omp-parallel-for loop is annotated with the new OpenMPC clauses to express multi-dimensional thread batching and explicit shared memory usage

will be allocated on the CUDA shared memory when translated into CUDA codes. As OpenMP *private* variables are, *cudashared* variables are undefined on entry to a kernel region (line 6 in Figure 5.1) and on exit from the kernel region (line 18). A *cudasyncthreads* clause is used to indicate in-block synchronization using CUDA *__syncthreads()* runtime library call. OpenMPC directive containing *cudasyncthreads* should reside in kernel regions.

Using new OpenMPC extensions such as the ones explained above, programmers will be able to express complex multi-dimensional data access patterns and exploit CUDA shared memory more aggressively, still in high-level.

5.2.2 Compiler-assisted, Input-adaptive Tuning

OpenMPC provides a high-level infrastructure where various tuning systems can be built, without the detailed knowledge of the complex CUDA programming and memory models. As a reference, we have created a simple exhaustive tuning system, and the evaluation on various applications identified that the performance of some applications is very sensitive to input data. To address this input-sensitivity issue, input-adaptive tuning will be needed. The OpenMPC compilation framework can be used to build a compiler-assisted, input-adaptive tuning system. Overall steps of the new input-adaptive tuning system will be as follows:

- The compiler analyzes the input OpenMP program and identifies input-context variables, which are the input variables that control the execution behavior of the program, such as loop iteration size and variables that decide other control flow directions.

- OpenMPC *search space pruner* creates a search space only with applicable tuning parameters.

- For each of representative training input set, find the best tuning configuration using the OpenMPC reference tuning system. If the pruned search space is still too large for exhaustive search, tuning parameters in the search space can be grouped using clustering techniques such as K-Means algorithm [57], and then the best tuning configuration can be searched against representative tuning parameters of each group.

- For each representative training input, find the relation between its input-context variables and the corresponding best tuning configuration using statistical learning techniques such as Regression Trees [58].

- The best tuning parameter of the input program with an arbitrary input is estimated using the constructed relation information (e.g. regression trees).

Two open issues in this approach are (1) to find efficient clustering algorithm to classify the tuning parameters in the pruned search space into smaller number of groups feasible for exhaustive search and (2) to find effective learning algorithm to identify the relation between the input-context variables and the best tuning configurations.

LIST OF REFERENCES

LIST OF REFERENCES

[1] "Sh: A metaprogramming language for programmable GPUs. [online]. available: http://www.libsh.org."

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 777–786, ACM, 2004.

[3] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for brook streaming applications on multiprocessors," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, (Washington, DC, USA), pp. 196–207, IEEE Computer Society, 2006.

[4] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for gpus," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, (New York, NY, USA), p. 184, ACM, 2006.

[5] "NVIDIA CUDA [online]. available: http://developer.nvidia.com/object/cuda_home.html."

[6] "OpenMP [Online]. Available: http://openmp.org/wp/."

[7] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," *ACM International Conference on Supercomputing (ICS)*, 2008.

[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," *International Conference for High Performance Computing, Networking, Storage and Analysys (SC)*, 2006.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[10] S. Lee, T. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[11] "NVIDIA CUDA SDK - Data-Parallel Algorithms: Parallel Reduction [online]. available: http://developer.download.nvidia.com/compute/ cuda/1_1/Website/Data-Parallel_Algorithms.html."

[12] L. L. Pilla, "Hpcgpu Project [Online]. Available: http://hpcgpu.codeplex.com/."

[13] T. Davis, "University of Florida Sparse Matrix Collection [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/."

[14] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, (New York, NY, USA), pp. 52–61, ACM, 2009.

[15] T. Jansen, B. von Rymon-Lipinski, and E. Keeve, "Fourier Volume Rendering on the GPU using a Split-Stream-FFT," *Vison, Modeling, and Visualization (VMV)*, 2004.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 917–924, ACM, 2003.

[17] J. Kruger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," *ACM Transactions on Graphics*, vol. 22, pp. 908–916, 2003.

[18] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu, "Program optimization space pruning for a multithreaded GPU," *International Symposium on Code Generation and Optimization (CGO)*, 2008.

[19] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 73–82, 2008.

[20] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu, "CUDA-lite: Reducing GPU programming complexity," *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[21] S.-J. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on software distributed shared memory systems," *International Journel of Parallel Programming (IJPP)*, vol. 31, pp. 225–249, June 2003.

[22] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," *ACM International Conference on Supercomputing (ICS)*, pp. 189–198, 2005.

[23] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *International Journel of Parallel Programming (IJPP)*, vol. 36, pp. 289–311, June 2008.

[24] H. Wei and J. Yu, "Mapping OpenMP to Cell: A effective compiler framework for heterogeneous multi-core chip," *International Workshop on OpenMP (IWOMP)*, 2007.

[25] J. S. Park, J. G. Park, and H. J. Song, "Implementation of OpenMP workshares on Cell broadband engine," *International Workshop on OpenMP (IWOMP)*, 2007.

[26] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[27] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 491–542, Oct. 1987.

[28] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[29] D. Levine, D. Callahan, and J. Dongarra, "A comparative study of automatic vectorizing compilers," *Parallel Computing*, vol. 17, 1991.

[30] E. J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applicatins*, vol. 18, no. 1, pp. 135–158, 2004.

[31] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Proceedings of Supercomputing (SC)*, 2007.

[32] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication," *Electronic Transactions on Numerical Analysis*, vol. 21, pp. 47–65, 2005.

[33] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. on Parallel and Distributed systems*, vol. 10, July 1999.

[34] E. G. Boman and U. Catalyurek, "Constrained fine-grain parallel sparse matrix distribution," *SIAM Workshop on Combinatorial Scientific Computing*, 2007.

[35] C.-W. Qu and S. Ranka, "Parallel incremental graph partitioning," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 884–896, 1997.

[36] S. G. Nastea and O. Frieder, "Load-balancing in sparse matrix-vector multiplication," *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, p. 218, 1996.

[37] L. H. Ziantz, C. C. Ozturan, and B. K. Szymanski, "Run-time optimization of sparse matrix-vector multiplication on SIMD machines," *Int. Conf. Parallel Architecture and Languages*, vol. 817, pp. 313–322, July 1994.

[38] A. T. Ogielski and W. Aiello, "Sparse matrix computations on parallel processor arrays," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 519–530, 1993.

[39] J. I. Aliaga and V. Hernandez, "Symmetric sparse matrix-vector product on distributed memory multiprocessors," *Conf. on Parallel Computing and Transputer Applications*, 1992.

[40] E. Rothberg and R. Schreiber, "Improved load balancing in parallel sparse Choleski factorization," *Proceedings of Supercomputing (SC)*, 1994.

[41] D. Jin and S. G. Ziavras, "A super-programming technique for large sparse matrix multiplication on PC clusters," *IEICE Trans. Inf. & Syst.*, vol. E87-D, July 2004.

[42] R. Shahnaz, A. Usman, and I. R. Chughtai, "Implementation and evaluation of parallel sparse matrix-vector products on distributed memory parallel computers," *IEEE Int. Conf. on Cluster Computing*, pp. 1–6, 2006.

[43] B. Hendrickson and T. G. Kolda, "Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing," *SIAM J. on Scientific Computing*, vol. 21, no. 6, pp. 2048–2072, 2000.

[44] B. Vastenhouw and R. H. Bisseling, "A two-dimmensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[45] "The message passing interface (MPI) standard [online]. available: http://www-unix.mcs.anl.gov/mpi/."

[46] R. Rabenseifner, "Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512," *Euro PVM/MPI*, pp. 35–42, 1999.

[47] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to Open MPI," *Euro PVM/MPI*, 2006.

[48] S. S. Vahdiyar, G. E. Fagg, and J. J. Dongarra, "Automatically tuned collective communications," *Proceedings of Supercomputing (SC)*, p. 46, 2000.

[49] A. Faraj and X. Yuan, "Automatic generation and tuning of MPI collective communication routines," *Int. Conf. on Supercomputing*, pp. 393–402, 2005.

[50] J. Su and K. Yelick, "Automatic support for irregular computations in a high-level language," *Proc. of 19th IPDPS*, p. 53, 2005.

[51] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," *2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–10, 2009.

[52] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," *International Conference on Compiler Construction (CC)*, vol. Volume6011/2010, pp. 244–263, March 2010.

[53] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 1–10, ACM, 2009.

[54] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2008.

[55] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–11, IEEE Press, 2008.

[56] Z. Pan and R. Eigenmann, "PEAK—a fast and effective performance tuning system via compiler optimization orchestration," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–43, 2008.

[57] J. A. Hartigan and M. A. Wong, "A K-Means Clustering Algorithm," *Applied Statistics*, vol. 28, no. 1, pp. 100–108, 1979.

[58] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning," *Springer*, 2001.

VITA

VITA

Seyong Lee was born in May 1976 in Cheongju, South Korea. He received his BS degree in Electrical Engineering from Seoul National University at Seoul, South Korea, in February 1999. From January 1999 to July 2002, Seyong Lee worked at Xeline at Seoul, South Korea as an engineer to design powerline communication modems. He received MS degree in Computer Engineering at the School of Electrical and Computer Engineering, Purdue University, West Lafayette in Indiana, USA, in May 2004, and he continued to pursue a PhD at the same school with the same major. He successfully defended his thesis in April 2011 and received his PhD degree in May 2011.