AUTOMATIC TRANSFORMATION AND OPTIMIZATION OF APPLICATIONS ON GPUS AND GPU CLUSTERS

DISSERTATION

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

By
Wenjing Ma
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
mawe@cse.ohio-state.edu, BE/MS
Graduate Program in Computer Science and Engineering

The Ohio State University
2011

Dissertation Committee:
Dr. Gagan Agrawal, Advisor
Dr. Feng Qin
Dr. Atanas Rountev
ABSTRACT

Modern accelerators and multi-core architectures offer significant computing power at a very modest cost. With this trend, an important research issue at the software end is how to make the best use of these computing devices, and how to enable high performance without the users having to put too much effort into learning the architecture and the programming model.

Our goal is to address the above problem by developing automatic code generation systems, particularly for GPUs and GPU clusters. We believe that by focusing on specific application classes, the task of automatic code generation can be significantly simplified. Thus, we made efforts in providing code generation and optimization systems for two classes of applications: data-intensive applications with generalized reductions, and tensor contraction functions. First, we focused on a class of data-intensive applications, whose processing structure is of generalized reductions. In the code generation systems we have built, the user input are algorithms written in high-level languages, specifically, C or MATLAB. Program analysis and code generation is performed to generate code for a single GPU, or a GPU cluster. The three specific systems we have built are GREENRIDE, a code generation system to provide GPU support for C programs; GMAT-DM, which translates MATLAB code into GPU executable program; and AUTO-GC, which provides GPU support for clusters, by incorporating code generation for FREERIDE, which is a middleware supporting parallel execution for data mining.

For tensor contractions, we evaluated the automatically generated code on different GPUs, and made investigation in the algorithm optimization for each card. It led to an auto-tuning framework which selects algorithms and parameters according to some cost model and thresholds extracted from simple micro-benchmarks. We also developed a loop
transformation system in the environment of multi-level memory hierarchy. By focusing on the dominating factors of the computation, we were able to remove a large portion of extra data movement between memory hierarchies.

In future, we plan to extend our work in the following directions. The code generation system for data intensive applications with reduction patterns could be applied and optimized for other classes of applications. The integer programming model could also be used for other architectures, including future accelerators. We would like to consider heterogeneous systems for the loop transformation approach. The auto-tuning framework will be extended to include more parameters, enabling better performance gain.
ACKNOWLEDGMENTS

First of all, I want to give sincere gratitude to my advisor, Dr Gagan Agrawal. Ever since I came to OSU, he has been leading me in every step, showing me the right way to do research, helping me to learn about work and life. Not only has he given me great ideas for my work, but he is also trying to lead me onto the track of thinking about research. When I was in the low point of study and research, he never lost confidence on me, and kept encouraging me and giving me practical instructions to get out of difficult situation. He has been showing great patience and eagerness to help when I made mistakes again and again. But for his efforts, I would never have overcome the obstacles in my nearly 6 years’ PhD process. Not only did he help me in every aspect of research and study, but he has established an excellent model for me, with his attitude toward work, the way he treats people, his humour and optimism, and being responsible for his students.

I am also very grateful to Dr Sriram Krishnamoorthy, my mentor in PNNL, when I was a student intern there. Joining his research provided a new and promising direction for my work. He has been helping me in every detail, with the code we developed, the calculation we made, and the paper we wrote. His kind, frank, and patient character is also a model for me. And I am very glad to be able to join his group as a postdoc RA after graduation.

I want to thank Dr Atanas Rountev, for the help in directing my research from the compiler aspect. His ideas and suggestions enabled me to build my work in the right direction, and to write papers more accurately. Thanks to Dr Feng Qin and Dr Radu Teodorescu, we have been having a great time collaborating, from which I learned a lot and I really appreciate your patient and careful advisory while we were working together. Thanks to Dr Raghu Machiraju, for sitting as my committee member when I took my candidacy exam, and provided me opportunities to go to the presentations of his students.
Also, thanks to Dr Firdaus Janoos, for the help he gave me on some software of MATLAB. Thanks to Dr Hanwei Shen, who gave me the chance to listen to his students’ presentation about CUDA, which is my first taste about GPGPU.

I would like to thank Dr Oreste Villa and Dr Karol Kowalski in PNNL, for the help on my research in PNNL. I will also give my special thanks to Dr Leonid Glimcher, the senior student who helped me to learn about the former research of our group in every detail.

At last, I want to thank all the students in my lab (Qian, Fan, David, Wei, Vignesh, Tantan, Tekin, Bin, Xin, Smita, Sai, Jiedan, Yu, Yi, Vignesh. S, Ashish, etc), for the nice environment we have constructed, and for the great help they have given me through these years. There are too many people who gave me important help in the last 5 years. Though not all of their names are listed here, I will never forget the help I got from them. I feel that my life is getting really rich in this great school, and I believe the things I got and learned from here will benefit me forever.
Dedicated to my mother, Rufeng Xing, and my father, Biaocheng Ma
VITA

1981 .................................. Born in Tianjin, China

2004 .................................. B.E. in Computer Science and Technology,
                                  Nankai University, China

2009 .................................. M.S. in Computer Science and Engineering,
                                  The Ohio State University

2004-2005 ............................... Software Engineer, Fankai Science and Trade
                                     Company, Ltd, Tianjin, China

2006-Present ............................ Graduate Teaching/Research Associate, The
                                     Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Specialization: Parallel Computing and Compiling
# TABLE OF CONTENTS

Abstract ................................................................. ii  
Acknowledgments ....................................................... iv  
Dedication ................................................................. vi  
Vita .................................................................. vii  
List of Figures .......................................................... xii  

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
</table>
| 1  | Introduction ...................................................... 1  
  1.1 Popularity of accelerators, GPGPU and GPU Clusters ................ 1  
  1.2 Difficulty of GPU Programming .................................. 2  
  1.3 Our Approach for Applications on GPU and GPU Cluster .......... 3  
  1.4 Current Research .................................................. 5  
  1.5 Outline ............................................................... 7  |
| 2  | Related Work .......................................................... 8  
  2.1 General Programming on GPUs and GPU Clusters .................. 8  
  2.2 Parallel Data Mining and Data Mining with GPU .................. 9  
  2.3 Parallelization of MATLAB ........................................ 10  
  2.4 Arranging Shared Memory with ILP ............................... 11  
  2.5 Optimizing Tensor Contraction on Modern GPUs .................. 12  
  2.6 Auto-tuning for CPUs and GPUs ................................... 12  
  2.7 Loop Fusion for Tensor Contractions in Multi-level Memory Hierarchy . 13  |
| 3  | Translation System for Enabling Data Mining Applications on GPUs .......... 16  
  3.1 CUDA and NVIDIA GPU Architecture .............................. 17  
  3.2 System Design ....................................................... 19  
    3.2.1 Parallel Data Mining ........................................... 19  
    3.2.2 System API ...................................................... 20  
  3.3 System Implementation ............................................ 22  
    3.3.1 Code and Variable Analysis .................................... 22  
    3.3.2 Mapping to GPGPU ............................................... 25  
  3.4 Optimizations ........................................................ 26  
    3.4.1 Dealing with Shared Memory ................................... 27  
    3.4.2 Reducing Memory Allocation and Copy Overheads .......... 27  |
3.5 Experiments and Evaluation ........................................ 28
  3.5.1 K-means Clustering ............................................ 29
  3.5.2 EM Clustering Algorithm ...................................... 32
  3.5.3 Principal Component Analysis ................................. 34
3.6 Summary ............................................................. 35

4 Automatic Translation of Data Mining Applications from MATLAB to GPUs 37
  4.1 System Design .................................................... 38
  4.2 C Code Generation and Optimization from MATLAB ............ 40
    4.2.1 A Motivating Example: K-means Clustering ............... 40
    4.2.2 Problem Formulation ....................................... 42
    4.2.3 Solution Approach ......................................... 42
    4.2.4 Other Optimizations ....................................... 45
  4.3 Experiments and Results ........................................ 46
    4.3.1 Results from K-means Clustering .......................... 47
    4.3.2 Results from EM ........................................... 47
    4.3.3 Comparison with GPU version of MATLAB ................ 48
  4.4 Summary .......................................................... 48

5 Automatic Code Generation for FREERIDE with GPU Support .... 53
  5.1 System Design .................................................... 54
    5.1.1 System components ......................................... 54
    5.1.2 System API ................................................ 55
    5.1.3 Program Analysis .......................................... 57
    5.1.4 Code Generation for FREERIDE .................. 58
  5.2 Experiments ....................................................... 60
    5.2.1 K-means Clustering ..................................... 60
    5.2.2 Principal Component Analysis ............................ 62
  5.3 Summary .......................................................... 62

6 An Integer Programming Framework for Optimizing Shared Memory Usage on GPUs ........................................ 64
  6.1 System Design .................................................... 66
    6.1.1 Assumptions and Basic Ideas ................................ 66
    6.1.2 Detailed Integer Programming Model .................... 68
    6.1.3 Code Generation ........................................... 73
    6.1.4 Improving Shared Memory Use with Loop Transformations 74
  6.2 Experiments ....................................................... 75
  6.3 Summary .......................................................... 79

7 Evaluation of Algorithm for Tensor Contractions on different GPUs .... 85
  7.1 Background .......................................................... 87
  7.2 System Design .................................................... 88
    7.2.1 Tensor Contraction and Initial GPU Implementation .... 88
    7.2.2 Algorithmic Modifications and Performance Tuning for Fermi 92
  7.3 Experimental Evaluation ........................................ 96
    7.3.1 Overall Speedups on Fermi .................................. 98
    7.3.2 Impact of Restructuring Optimizations .................... 102
    7.3.3 Benefits from Other Optimizations ...................... 105
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>GPU Architecture</td>
</tr>
<tr>
<td>3.2</td>
<td>Overall System Design: User Input is Shown as Shaded Boxes</td>
</tr>
<tr>
<td>3.3</td>
<td>Generalized Reduction Processing Structure of Common Datamining Algorithms</td>
</tr>
<tr>
<td>3.4</td>
<td>Format of the User Input</td>
</tr>
<tr>
<td>3.5</td>
<td>User Input for k-means</td>
</tr>
<tr>
<td>3.6</td>
<td>Classification of Variables for K-means Reduction Function</td>
</tr>
<tr>
<td>3.7</td>
<td>Speedup of K-means on GeForce 8800GTX</td>
</tr>
<tr>
<td>3.8</td>
<td>Speedup of K-means on GeForce 9800GX2</td>
</tr>
<tr>
<td>3.9</td>
<td>Scalability of EM Application</td>
</tr>
<tr>
<td>3.10</td>
<td>Comparison of E and M Phase computing time among the 3 Shared Memory Layout Strategies</td>
</tr>
<tr>
<td>3.11</td>
<td>Comparison of E and M Phase memory copy time between Normal and Optimized Memory Allocation Strategies</td>
</tr>
<tr>
<td>3.12</td>
<td>Speedup of PCA: With and Without User-provided Optimized Combination Function</td>
</tr>
<tr>
<td>3.13</td>
<td>Sequential code for the Expectation Maximization Algorithm</td>
</tr>
<tr>
<td>4.1</td>
<td>GMAT-DM Design</td>
</tr>
<tr>
<td>4.2</td>
<td>K-means in C and MATLAB</td>
</tr>
<tr>
<td>4.3</td>
<td>Generated C code for k-means with Naive Algorithm</td>
</tr>
<tr>
<td>4.4</td>
<td>M phase in EM Clustering</td>
</tr>
<tr>
<td>4.5</td>
<td>K-means: Speedup of Different Versions (Copy time Excluded)</td>
</tr>
<tr>
<td>4.6</td>
<td>K-means: Speedup of Different Versions - Copy Time Included</td>
</tr>
<tr>
<td>4.7</td>
<td>EM: Speedups of E Phase</td>
</tr>
</tbody>
</table>
7.8 Boundary checking – reversed order ........................................ 115
7.9 Speedup of the various versions with respect to the CPU. x-axis - the different
sd2 functions. y-axis - speedup ........................................ 116
7.10 Execution time of CCSD(T) part of NWChem with GPU on Uracil and NiO2
molecules ................................................................. 116
8.1 Parametrized Micro-benchmarking based Auto-tuning .................... 122
8.2 Memory Access Costs (Main Cases) on Fermi .......................... 126
8.3 Memory Access Costs (Main Cases) on T10 ............................ 127
8.4 Computation of N Tasks With and Without Consolidation ............ 130
8.5 Comparison of Consolidation of Micro-benchmark with Copy on Fermi .. 131
8.6 Procedure to Select Index Order for a Tensor Contraction ............ 133
8.7 Comparison of Consolidation for Expression (I), (II), (III) with Copy on Fermi140
8.8 Comparison between Baseline and Auto-Tuned Versions ................ 141
9.1 Loop structures for a single contraction on a two-level memory hierarchy.
(a) Loop structure if $N_y \leq N_x$(b) Loop structure if $N_x \leq N_y$(c) Simplified
loop structure for discussion (d) Example code for the contraction $C = A \times B$ 150
9.2 Tensor contraction expressions used as benchmarks ........................ 160
9.3 Execution time with (left) and without (right) fusion at global memory level 162
9.4 Disk I/O time with (left) and without (right) fusion at disk level ........ 162
9.5 Execution time with (left) and without (right) fusion at global memory level
for expression D - different dimension sizes .......................... 164
9.6 Disk I/O time with and without fusion at disk level for expression D - dif-
ferent dimension sizes ................................................... 164
9.7 Execution time with (left) and without (right) fusion at global memory for
expression A on different numbers of nodes .......................... 165
9.8 Disk I/O time with and without fusion for expression A on different numbers
of nodes ................................................................. 165
CHAPTER 1
INTRODUCTION

In this dissertation, I will summarize automatic code generation and optimization on accelerators and multi-core architectures. The motivation of our work is the potential of using accelerators to speedup general purpose computation, and the computing power of multi-core architectures. Nowadays, GPUs, FPGAs, the Cell processor, and other accelerators are being investigated extensively for general purpose computing. Thus, from the software point of view, how to make the best use of these computing hardware becomes a challenging problem. Moreover, it is important to provide high performance to the users without requiring much efforts in learning the architecture details and programming model. Therefore, we solve this problem by starting from the specific classes of applications that demand large scale computing, and extending the approaches to a larger range of computations. In this chapter, we will introduce the motivation and background of the work, with an overview of our current research.

1.1 Popularity of accelerators, GPGPU and GPU Clusters

Accelerators including GPUs and FPGAs are getting popular in high performance computing research. Many of them offer an excellent performance to price ratio for scaling applications. Among these devices, GPUs are being used extensively, because of their parallel nature and the potential to speedup SIMD computation [152, 63, 144, 133, 148, 29]. In reality, users with a single desktop usually have a powerful GPU to support their graphics applications. Such users can speedup their programs (especially computing intensive
applications) with this GPU. In other scenarios, a cluster may be available for supporting large-scale data processing. Such clusters often need to have visualization capabilities, which means that each node has a powerful graphics card. Thus, to make good use of these resources, graphic device vendors have been making efforts in enabling and providing interfaces for general purpose computing on GPUs. AMD introduced the concept GPGPU (General Purpose computing with GPUs) in 2006 [10]. In the top 500 supercomputer list released in November 2010, three of the four fastest supercomputers were based on GPUs. NVIDIA developed CUDA (Compute Unified Device Architecture), which is applied to their latest series of graphic cards [128]. AMD also has the Stream Technology for GPGPU [9]. IBM Cell/BE is another architecture under intensive research which provides computing power for general applications [90]. Even prior to these developments, there had been a growing interest in the use of GPUs for non-graphics applications [28, 35, 44, 56, 63, 67, 148], as also documented in the GPGPU web-site\(^1\). Combining the usage of GPU, multi-core architectures and heterogeneous systems, OpenCL (Open Computing Language) is emerging as an open and cross-vendor standard for GPU and multi-core programming [157].

### 1.2 Difficulty of GPU Programming

Even though modern GPGPU languages including CUDA and OpenCL are accelerating the use of GPUs for general purpose applications, several challenges still remain in programming on GPUs. For example, CUDA involves explicit parallel programming, and explicit management of its complex memory hierarchy. In addition, allocating device memory, data movement between CPU and device memory, data movement between memory hierarchies, and specification of thread grid configurations is explicit. This implies a significant learning curve for programmers who want to improve the performance of their applications using the GPUs. Thus, it will clearly be desirable to be able to program GPUs using a higher-level interface.

On the other hand, as programmer productivity is becoming extremely important, there is a growing trend towards very high-level languages. Languages like MATLAB, Python,

\(^1\)www.gpgpu.org
Perl, and Ruby are often simpler to learn (especially, for programmers in certain domains), and result in much more concise code. But, because of their interpreted nature and the use of high-level constructs, they also often result in poor performance, besides being not able to exploit parallelism on multi-cores and GPUs. As we also try to explore the computing power of GPUs on clusters, making the best usage of these devices adds more challenge to the programmers. Clearly, it will be very desirable if translators can be built to automatically or semi-automatically translate programs written in high-level languages for scalable execution on GPUs, either for PC or for a parallel computer.

As new architectures of GPUs keep coming out, using the same algorithms on different devices cannot fully explore the computing power of each device. It is highly demanded that not only careful investigation should be make on the algorithm modification for each architecture, but also some approach should be used to generate and/or select suitable algorithm and parameters for a certain problem instance running on a certain card.

1.3 Our Approach for Applications on GPU and GPU Cluster

Considering the difficulty of programming on GPUs, and the big performance advantage we can get by using them, it is worth of making efforts to construct systems in which the user can write programs using the popular high level languages, while gaining high performance from efficient usage of accelerators.

On the other hand, it is very hard to provide a “universal compiler”, which can generate optimized code on GPU for any applications. Therefore, we selected two classes of applications which require high performance and are widely used. The first category is data mining and data analyzing applications, and the second category is tensor contraction functions. The first class of applications raise from the previous work in our research group, which led to the observation that a common processing structure fits a large number of popular data mining applications [86, 87]. Parallelization can be done by dividing the data instances (or records or transactions) among the nodes or threads. The computation on each node involves reading the data instances in an arbitrary order, processing each data
instance, and performing a local reduction. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a global reduction is performed. By making use of this program structure, we can provide automatically generated parallel code and GPU code for sequential data mining and data analysis applications that follow this pattern. As addressed in Section 1.2, we aim at enabling high performance on GPU and GPU clusters, and maintaining good productivity at the same time. Therefore, our system is an automatic code generation system that transforms sequential applications written in high level languages into code that can run on GPU or GPU clusters. The framework includes code analysis, code generation and optimization at each level. Focusing on this particular category of applications enables us to do the translation automatically by formulating the code in a common structure.

The second class of applications, tensor contraction, are widely used in chemistry calculation and is usually time-consuming. Even though it is simple in essence — the basic computation tasks are just matrix multiplication — the high dimensional feature and the architecture of GPUs make optimizing these functions on GPU a challenging task. Besides, various devices require different optimizations for the functions. Our evaluation on two different GPUs showed that the performance on GPU is affected by quite a few algorithm factors. So we proposed an auto-tuning framework to provide optimized code for tensor contraction functions on a particular device, with a certain expression, and for a certain range of tile size. The auto-tuning approach is based on micro-benchmarks, which are very simple kernels that run in a limited parameter space. Partial cost models and thresholds are extracted from the execution of micro-benchmarks, and are used to direct implementation selection for the real application.

Besides generating code on GPUs for this class of applications, we also made an effort to optimize the code in a larger scale of parallel computing. Because of the large amount of data being processed, and the unique access patterns of those functions, one possible and efficient optimization is reducing the data movement between memory hierarchies. We developed a framework to determine fusion patterns of tensor contraction functions on a multi-level memory hierarchy, based on a simple cost model.
1.4 Current Research

We have constructed the main framework of the code translation and optimization system, including the following five aspects.

**Automatic Transformation for Data Mining Applications to GPU:** In this code transformation system which we refer to as GREENRIDE, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the variables. We use program analysis and code generation to map the applications to a GPU. Several additional optimizations are also performed when the code is being generated. In addition, we allow the programmers to provide other functions and annotation, which can help achieve better performance. Overall, our work shows that very simple program analysis and code generation techniques can allow us to support a class of applications on GPUs with a higher-level interface than CUDA and OpenCL [110].

**Compiling System for Data Mining Applications in MATLAB with GPU:** As programmers in certain domains often prefer to use interpreting languages like MATLAB to implement data mining and data analysis algorithms, we developed a system GMAT-DM (Gpgpu from MATLAB for Data Mining applications), which is an automatic translation tool to convert data mining applications written in MATLAB to CUDA, for execution on GPUs. The key technical challenge we need to address is achieving memory efficiency. MATLAB provides a large number of operations and functions that make data mining algorithm implementation easier. However, this convenience often leads to inefficient usage of memory. Thus, we have developed a number of optimizations to achieve memory efficiency. First, the MATLAB code is translated into C code with the optimizations oriented to GPU programming. Then, the generated C code is converted into CUDA by the code transformation system introduced in the previous paragraph.

**Parallelization of Data Mining Applications on GPU Clusters:** Since our previous work FREERIDE was a middleware for parallel data mining [86, 87], we made an effort to enable code generation for FREERIDE, with GPU support. Thus, AUTO-GC is developed recently for this purpose. As in GREENRIDE, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the variables. We
use program analysis and code generation to map the applications to distributed memory cluster, with the support of a GPU on each node.

**Automatic Arrangement of Shared Memory on GPUs:** Efficient usage of shared memory on GPUs is a crucial issue in improving performance. We developed a framework for automatic arranging shared memory on GPUs, leveraging an integer programming solver. Within this framework, each variable (a scalar, an array, or a section of an array) in a kernel function is either in shared memory or not in a certain basic block. The boolean values of “in_shared_memory” for each variable are solved as the unknown variables of an integer linear programming problem. The objective function is finding the minimum data access to device memory, and the constraint is the limit of the shared memory. With this approach, we were also able to suggest loop transformations which enable better usage of shared memory.

**Auto-tuning for Tensor Contraction Applications on GPUs:** Since GPUs have various architectures and features, it is worth doing to use different algorithms for different devices. By evaluating the latest cards, we were able to find out the different algorithmic aspects that affect performance for each device. For a certain GPU, there are other factors that should be taken into account when selecting algorithms. Therefore, we constructed an auto-tuning framework to enable optimization to the GPU code for a certain tensor contraction function, according to the hardware being used, the pattern of the expression, and the tile size of the data. The auto-tuning approach is based on information extracted from running micro-benchmarks, which turned out to be efficient and effective.

**Loop Fusion for Tensor Contractions on Multi-level Memory Architectures:** For the second class of applications on GPU, the tensor contraction functions, we proposed an approach to automatically determine fusion patterns for loops over multiple levels of memory. By focusing on the dominant component of data movement, we were able to select loops for fusion with simple cost models. Our results showed that fusion decisions made by the framework provided significant improvement to performance.
1.5 Outline

The rest of the thesis is organized as follows. In Chapter 3, I will introduce the CUDA code generation system for data mining and data processing applications written in C. In Chapter 4, I give an overview of the GMAT-DM system, which transforms data processing applications in MATLAB into CUDA. Description of our automatic code generation system AUTO-GC is presented in Chapter 5. I introduce the automatic data arrangement on shared memory with ILP in Chapter 6. Then, in Chapter 7, I will show the algorithm modification for optimization on different GPU architectures, especially a recent card Fermi. In Chapter 8 I will elaborate on the auto-tuning system for tensor contractions on different GPUs. Chapter 9 describes the framework of automatically determining loop fusion for tensor contraction sequences on multi-level memory hierarchies. Conclusion and future directions are shown in Chapter 10.
CHAPTER 2

RELATED WORK

In this section, we briefly introduce some available research related to our work.

2.1 General Programming on GPUs and GPU Clusters

Exploring the computing power of GPU has been a topic of much investigation. In our research, we use NVIDIA’s CUDA [128] as our target language for GPU computing. Before the development of CUDA, Peercy et al. [133] proposed a computing model for GPU very similar to CUDA. Data representation and features of operations for GPU computing were also explored by Trancoso et al. [156]. Brook was a language developed to provide operations for data stream processing on GPUs [28]. Tarditi et al. [152] developed techniques to compile C# with an Accelerator, which evaluates the parallel part of the programs on a GPU, and the other parts on the CPU. While their system has many similarities with our work, they do not support complex reductions on parallel collections, which are common in data mining operations.

At UIUC, CUDA-lite [15] has been developed with the goal being to alleviate the need for explicit GPU memory hierarchy management by the programmers. The user input to our system is at a higher-level, in the sense that they do not need to write parallel code. However, our system is limited to a specific class of applications. The same group also investigated the optimization to CUDA programming, some of which are similar to the approaches we incorporated in our work [146]. MCUDA [150] is a compiler effort which takes CUDA code as input, and maps it to multi-core machines. Baskaran et al. use the polyhedral model for converting C code into CUDA automatically [21]. Their system is
focusing on affine loops, and provide a general high performance solution. Since we limit our target to a certain class of applications, we handle irregular reductions as well. A version of Python with support of CUDA, Pycuda, has also been developed, by wrapping the CUDA functions and operations into classes that are easy to use [96]. Some recent work has also made progress in translating OpenMP into CUDA [101]. The reported results are from simple stencil computations, and their is no support for handling complex reductions. Another group made an effort in scheduling and separating operators according to the input data set size [151].

Exploring the computing power of GPU in clusters has become a new focus of research in recent years. Center For Visual Computing and Department of Computer Science in Stony Brook University is among the first institutes that work on GPU clusters. They implemented parallel LBM computation on the GPU cluster, getting speedup of nearly 7 on 1 node, and about 5 on more than 4 nodes [56]. Other attempts are also made to use GPU as a coprocessor in a cluster. For example, Göddeke el at. implemented the multigrid solver on a GPU cluster, in a pattern similar to our approach. The computing intensive part is executed on GPUs, to overcome the extra time of memory copy [61]. The same group also implemented an FEM application on GPU cluster, and got good performance [60].

2.2 Parallel Data Mining and Data Mining with GPU

The main operations of data mining and data analysis applications can be generalized as reduction operations. Analysis and code generation for reduction operations has been studied by a number of distributed memory compilation projects [4, 17, 64, 76, 98, 168] as well as shared memory parallelization projects [24, 68, 70, 106, 108, 136, 166]. More recently, reductions on emerging multi-cores have also been studied [104]. Our work has many similarities, but is specific to the features of GPUs.

Map-reduce is a widely used parallel computing tool developed by Google, and there is already a CUDA version of map-reduce called Mars [75]. The map-reduce API typically results in high overheads for more compute-intensive data mining applications, because of the need for sorting reduction elements. Our system also supports a higher-level (almost sequential) API for these applications.
One of the popular data mining algorithms, k-Nearest Neighbor search, has been studied on GPUs by several groups [30], [137], [59] and [59]. Hall and Hart [67] ported different versions of k-means to GPU using Cg. Che et al. [37] did an analysis of CUDA computing model and a comparison with other architectures. Particularly, they tested k-means in CUDA, and got a speedup of about 70 over sequential code. This report, is based on a more advanced GPU (GeForce 260 GTX). Since our system is more general, it is to be expected that we can optimize a single application to the same extent. However, their work does form a basis for additional optimizations we can perform in our system in the future.

2.3 Parallelization of MATLAB

Several projects in the last 15 years have considered compilation of MATLAB. The main issue to be considered here is type, rank, and size analysis [145]. Other issues include JIT compilation [8] and parallelization [36]. A group at Northwestern University has done extensive work in compiling MATLAB programs, analyzing the array storage and its optimization of MATLAB. Most of the optimizations we perform are similar, expect that we focus on GPUs. We also simplify the analysis considerably by focusing only on matrix operations. The same group has also developed a tool, MAGICA, for type and shape inference [89]. Menon and Pingali made a case for source-level code transformations for MATLAB [119].

Optimizations for memory and storage requirements have been considered outside of MATLAB compilation as well. Song et al. developed a method to achieve memory reduction by combining loops [149]. We are considering this problem in the context of MATLAB. By focusing on matrix operations, we simplify the analysis, and do not require any dependence analysis. Zhang and Gupta proposed an approach for compacting dynamically allocated structures [167].

Recently released commercial products that provide GPU support for GPUs include GPumat and JACKET. GPumat is a convenient tool to leverage the computing power of GPUs, but it only supports single expressions and element functions [165]. JACKET uses GPU in a more extensive way, including for loops [107]. However, it requires that the programmers explicitly declare the data structures for GPU, and does not perform
advanced transformations. Our experiment showed a more than 50 times speedup over the execution with JACKET.

2.4 Arranging Shared Memory with ILP

In various application studies using GPUs, optimizing shared memory usage has been a major topic. Some of the prominent applications where shared memory use made a very substantial performance difference include N-Body [130], Three-Dimensional Cahn-Hilliard Field Equation Simulation [134], and FFT [66]. The methods used in these and other similar efforts were specific to the applications, and cannot be used as part of a high-level GPU programming system.

Baskaran et al. have provided an approach for automatically arranging shared memory on NVIDIA GPU by using the polyhedral model for affine loops [20]. They also focus on enabling more data reuse on shared memory, while reducing data movement. The main difference is that we consider non-uniform accesses to variables, and can analyze a full function, and not just a loop nest. Moazeni et al. have adapted approaches for register allocation to manage shared memory on GPU [121]. They used graph coloring to solve the problem. However, a major limitation of their approach is that only one variable is allowed in the shared memory at any given time.

Another set of efforts have focused on the use of scratch-pad memory, mainly in embedded systems. Diouf et al. also used ILP to solve the problem, though with more restricting assumptions [53]. Li et al proposed interval coloring approach for arranging scratch-pad memory, utilizing the observation that live ranges of two arrays should not interfere or one should not contain the other [102]. None of the work has considered non-uniform access or a multi-thread programming model. Udayakumaran et al. address the scratch-pad memory allocation in a dynamic way, using a greedy heuristic as the cost model [158].

A problem somewhat similar to the shared memory arrangement problem is of register allocation. Traditionally, register allocation is performed using graph coloring, as initially suggested by Chaitin et al. [33, 34]. In some of the more recent work, Briggs et al. improved this method using delayed coloring and rematerialization [26], Gupta et al reduced the space usage for register allocation [65], and Poletto et al. used linear scan to solve the
problem [135]. Integer linear programming, which is widely used in many scheduling and routing application [1], is also used for register allocation [62, 18].

2.5 Optimizing Tensor Contraction on Modern GPUs

Fermi is the latest graphic card that supports CUDA [129, 127]. It came into production in early 2010. Some studies on application development on Fermi are already available. Anzt et al. evaluated basic kernels including dot-products, vector updates, scalar-products, matrix-vector, and matrix-matrix operations, a CG solver, and a mixed precision iterative refinement solvers [13]. Nath et al. developed a new algorithm for matrix-matrix multiplication particularly for the Fermi architecture [124]. They also used the idea of register tiling, which has been applied in our work as well.

Optimization for tensor contractions have been studied on many platforms. We have successfully accelerated the execution of CCSD(T) computations on a heterogeneous architecture with both GPUs and multicore CPUs [114]. The GPU used in this work was a Tesla T10, and the algorithm developed in this work was the starting point for the Fermi-specific optimizations reported in Chapter 7. Other research in this field has focused on optimizations of loop structure and cache utilization [74, 23, 73]. Similar work has been done with single precision matrix multiplication for CCSD(T) computations at the Argonne National Lab [69], where the focus was on overlapping of different levels of communication.

2.6 Auto-tuning for CPUs and GPUs

Auto-tuning, as a widely used tool to gain good performance adaptively, was used for improving the performance for many applications. Typical tuning methods such as empirical search [12, 155, 160] and analytical model [164] have gained effective optimization for compiler optimization. In Belter et al.’s work, auto-tuning approach was used to evaluate the profitability of each optimization [22]. Chen et al.’s work selects code patterns that benefits certain applications with certain problem sizes by analyzing footprint and cache reuse [39]. In Lu et al.’s work [109], a combined approach with analytical model and empirical search
is used to explore architectural features such as memory hierarchy, memory misalignment, and vector instruction sets, to get good performance for matrix transposition.

Efforts are also made to apply auto-tuning tool to GPGPU. Early research was done for matrix multiplication with BrookGPU language [85], utilizing the low level hardware features of GPUs, and code selection is done with empirical search at compile time. Analytical models of applications on GPU were used for evaluating performance [78, 16] and power consumption [79]. Detailed study has been done to investigate hardware features such as memory access pattern on GPU [159]. A simple auto-tuning approach shows its effectiveness for DGEMM [103]. The target application that we focused on is more complicated, and by using the micro-benchmark, we still kept our model simple. Also, our tuning for kernel-consolidation is essentially an algorithm factor that combines GPU and host functions. Choi et al. investigated auto-tuning approach for sparse matrix-vector multiplication [41]. Different from their models, we used micro-benchmarks that only focusing on the main factors that affect performance, therefore, the expressions we extracted are simple, and can be evaluated using the basic architecture parameter and the problem size. Some research on auto-tuning for FFT application on GPU has also been successful [54]. However, they need to run the selected schedules at run time. In our approach, the overhead of running micro-benchmarks only happens at compile-time. Datta et al. developed an auto-tuning system for stencil computation on a variety of platforms, including Intel Clovertown, AMD Barcelona, Sun Victoria Falls, IBM QS22 PowerXCell 8i, and NVIDIA GTX280 [49]. In our future work, we also plan to provide support to more platforms than GPUs.

2.7 Loop Fusion for Tensor Contractions in Multi-level Memory Hierarchy

The previous work on optimizing tensor contractions focused on efficient exploration of the search space of memory hierarchy transformations [147, 58]. Sahoo et al. pruned the search space while combining permutation, loop fusion, tiling, and placement of disk I/O operations [147]. Gao et al. extended this work further to minimize disk I/O costs [58]. This
work only considered data movement across one level of memory hierarchy. In comparison, our work is the first in offering solutions with data movement across more than two levels. We have also presented more efficient algorithms for determining candidates for loop fusion.

Outside the specific context of tensor contractions, there is a very large volume of work on optimizing for memory hierarchies [117, 118]. Most of this work has focused on improving the utilization of a processor cache, which is not explicitly programmer controlled. One key difference between the work in this area and our work is in the metrics used for transformations. Cache transformations are based on metrics like reuse distance [52] or stack distance [32]. In comparison, we target programmer controlled memory hierarchy levels, and applications where we primarily see capacity misses. Also, because we are focusing on a limited class of applications, it is possible to develop a more accurate cost model. Darte has studied the general complexity of loop fusion [47] and has presented an approach for optimized memory allocation [48].

Significant work has been done on optimizing out-of-core programs. Particularly, Brown et al. have used compiler analysis to optimize prefetching of data in out-of-core programs [27]. Kandemir et al. have considered several compiler optimizations for out-of-core parallel programs [92] and other I/O intensive applications [91], building on earlier work at Syracuse/Northwestern and Rice [25, 154]. The key difference in our work is the consideration of data movement across more than two levels of programmer controlled memory hierarchy.

Prior to interest in accelerating computations using GPUs, several researchers have studied deeper memory hierarchies. Mitchell et al. considered multi-level caches, and further included considerations for instruction-level parallelism and TLB in their work [120]. For multi-level caches, they observed that single-level tiling is sufficient. We have considered a different architecture, and our conclusions are different. Rivera and Tseng examined loop transformations for multi-level caches, and finding that all performance gains can be achieved by simply focusing on L1 cache [143]. Clearly, as we have considered architectures with a different type of multi-level memory hierarchy, our conclusions are different. More recent work has considered multi-level tiling, but specific to stencil computations [141].
perform tiling for a different class of applications. Qaseem et al. have used complex modeling and a direct search to find transformation parameters [138]. Ren et al. presented an auto-tuning framework for software-managed memory hierarchies [140]. We, in comparison, employ an analytical approach.

A very similar problem is optimization for scratch-pad memory in embedded systems. Diouf et al. used integer linear programming to solve the problem [53]. Li et al. proposed interval coloring approach for arranging scratchpad memory, utilizing the observation that live ranges of two arrays should not interfere or one should not contain the other [102]. Udayakumaran et al. address the scratch-pad memory allocation in a dynamic way, using a greedy heuristic as the cost model [158]. None of these efforts have considered multi-level data movement. Most of these efforts are also not suitable for applications like tensor contractions, where only a fraction of a single array can fit in the memory.
CHAPTER 3
TRANSLATION SYSTEM FOR ENABLING DATA MINING APPLICATIONS ON GPUS

CUDA (Compute Unified Device Architecture), released by NVIDIA, is a very significant development in GPGPU. CUDA allows GPU programming with C language-like features, thus easing the development of non-graphics applications on a GPU. However, several challenges still remain in programming the GPUs. This is because CUDA involves explicit parallel programming, and explicit management of its complex memory hierarchy. In addition, allocating device memory, data movement between CPU and device memory, data movement between memory hierarchies, and specification of thread grid configurations is explicit. Thus, it will clearly be desirable to program GPUs using a higher-level interface.

To solve this problem, we offer a solution that is driven by the observation that a common processing structure fits a large number of popular data mining applications. In previous work of our group, it was observed that parallel versions of several well-known data mining techniques share a relatively similar structure [86, 87]. Parallel versions of several data mining applications were carefully studied, including apriori association mining [6], Bayesian network for classification [38], k-means clustering [84], k-nearest neighbor classifier [72], artificial neural networks [72], and decision tree classifiers [123]. In each of these methods, parallelization can be done by dividing the data instances (or records or transactions) among the nodes or threads. Each node processes part of the data, and performs a local reduction. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a global reduction is performed. Thus, we can expect similarities in how they can be ported on GPUs.
In our solution, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the parameters. We use program analysis and code generation to map the applications to a GPU. Several additional optimizations are also performed by the code generator. In addition, we allow the programmers to provide other functions and annotation, which can help achieve better performance. Overall, our work shows that very simple program analysis and code generation techniques can allow us to support a class of applications on GPUs with a higher-level interface than CUDA.

We have evaluated our system using three popular data mining applications, k-means clustering, EM clustering, and Principal Component Analysis (PCA). The main observations from our experiments are as follows. The speedup that each of these applications achieve over a sequential CPU version ranges between 20 and 50. The automatically generated version did not have any noticeable overheads compared to hand written codes. Finally, the optimizations performed in the system resulted in significant performance improvements.

### 3.1 CUDA and NVIDIA GPU Architecture

Among all the GPGPU products, NVIDIA’s CUDA is a successful one, which has been studied and used by many research communities and users who need high performance at low cost. Most of our research on GPUs are based on CUDA, but the ideas and technology can be easily ported to or implemented on other systems. Our work has used GeForce 8800 GTX and 9800 GX2 graphics cards, which are typical and widely used graphic cards that provide good performance/price ratio. In this section, we give a brief description of the architecture and programming model of GeForce 8800 GTX card, which is also common to many other newer cards.

This particular device has 16 multiprocessors, as shown in Figure 3.1. Each multiprocessor has a 575 MHz core clock and 16 KB of shared memory. The device memory totals 768 MB, with memory bandwidth of 86.4 GB/sec and 384-bit memory interface. Starting with the 8 Series GeForce, NVIDIA has started supporting high-level programming of the GPUs through CUDA. The computation to be performed by the device can be written as in normal C, with some predefined parameters and functions. Critical parameters of the
computation, such as the configuration of the thread grid and size of shared memory to be used, have to be supplied by the developer explicitly.

The kernel function is executed by the GPU in a SIMD manner, with threads executing on the device organized as a grid of thread blocks. Threads in one block have access to the same shared memory, which is a small piece of memory with high access speed. A mechanism for thread synchronization within one block is provided [128]. Each thread block is executed by one multiprocessor, and the threads within a block are launched in warps. Warps of threads are picked by the multiprocessor for execution, the exact order is undefined. The number of threads in a warp is fixed for a particular architecture. In the GeForce 8800 GTX model, 32 threads are launched in every warp. The number of thread blocks, however, can be varied by the developer based on requirements of computation or other preferences, with the maximum number being 65536 in one grid.

![Figure 3.1: GPU Architecture](image-url)
3.2 System Design

Our system is designed to ease GPU programming for a specific class of applications. Besides a C program to be executed on CPUs, the only required input from programmers is explicit recognition of reduction functions to be parallelized on GPUs, with additional information about the variables. Given such user input, the system can generate CUDA functions that execute these reduction functions in parallel, and the host functions invoking them.

The architecture of the system is shown in Figure 3.2. There are four components in the user input. The first three are analyzed by the system, they are: variable information, reduction function(s), and additional optional functions. The fourth component is the host program. The system itself has three components: code analyzer, which obtains variable access patterns and combination operations, variable analyzer, and the code generator. By analyzing the variables and the sequential reduction function(s), the system generates the kernel functions, grid configuration, and other necessary code. By compiling these functions with the user-specified host program, an executable file is generated. We used LLVM as the framework for program analysis [99]. We particularly benefited from the clear structure of its Intermediate Representation (IR).

3.2.1 Parallel Data Mining

Our system exploits a common structure underlying most data-intensive applications and data mining algorithms, which is shown in Figure 3.3. Each computing node or thread performs local reduction, by applying \( op \) on each data element or transaction, which is irregular in the sense that which elements of the reduction objects are updated depends upon the results of the processing of an element. Then, the reduction objects \( \text{Reduce} \) are combined and distributed to each node at the end of every iteration. For example, in k-means clustering, each iteration involves processing each point in the dataset. For each point, we determine the closest \( \text{center} \) to this point, and compute how this \( \text{center} \) should be updated.

This generalized reduction structure identified from data mining algorithms leads to the introduction of FREERIDE, a framework for parallel data mining [86, 87]. FREERIDE has some similarities with the map-reduce paradigm that Google has developed [50], but
it focuses more on the performance of the system, and has more optimized memory usage. There are also some differences in the generalized reductions.

For algorithms following such a generalized reduction structure, parallelization can be done by dividing the data instances (or records or transactions) among the processing threads. The computation performed by each thread will be iterative and will involve reading the data instances in an arbitrary order, processing each data instance, and performing a local reduction. Our system targets GPU-based parallelization of only the functions that follow this structure. By targeting a limited class of functions, we can simplify program analysis and automatic generation of GPGPU programs, while still offering a simple and high-level interface for the programmers.

### 3.2.2 System API

Using the common generalized reduction structure of our target applications, we provide a convenient API for a programmer. The format of input for a reduction function is shown in Figure 3.4. If there are multiple reduction functions, for example, the E phase and M phase in EM clustering, the user can define more than one section by specifying labels for each
While () {
    /* Outer Sequential Loop */
    Foreach (element e) {
        /* Reduction Loop */
        (i, val) = process(e);
        Reduc(i) = Reduc(i) op val;
    }
    /* operation on the combined Reduc */
    Finalize();
}

Figure 3.3: Generalized Reduction Processing Structure of Common Datamining Algorithms

one. A host program, not shown in Figure 3.4, invokes these reduction functions. Besides the label and the host program, the other components are as follows.

Variables for Computing: As shown in Figure 3.4, the declaration of each variable follows the following format:

   name, type, length[value]

   name is the name of the variable, type can be either a numeric type like int or pointer type like int*, which indicates an array. If this is a pointer, length is the size of the array, which can be a list of numbers and/or integer variables, and the size of the array is the multiplication of these terms. Otherwise, this field denotes a default value. We require all pointers to be one-dimensional, which means the user should marshal the multi-dimensional arrays and structures into 1-D arrays.

Sequential Reduction Function: The user can write the sequential code for the main loop of the reduction operation in C. Any variable declared inside the reduction function should also appear in the variable list as shown in Figure 3.4, and memory allocation for these variables is not needed.

Optional Initialization and Combination Functions from the User: Normally, the initialization and combination for the reduction objects and other variables is done by the code generator component of the system. However, if the user is familiar with
CUDA programming, they can provide their own combination and initialization functions, potentially improving the performance.

An example of user input for the k-means clustering algorithm is shown in Figure 3.5. The first line is the number of reduction functions, which is 1 here. The second line is the label `kmeans`. The following 5 lines are variable descriptions. Then, a sequential reduction function is provided.

3.3 System Implementation

This section describes the implementation of our system.

3.3.1 Code and Variable Analysis

The program analysis part comprises of three components. The first of these components is obtaining variable access information from a reduction function.

**Obtaining Variable Access Features:** We classify each variable as one of input, output and temporary. An input variable is input to the reduction function, which is not updated...
\begin{verbatim}
k int
n int
data float* n 3
update float* 5 k
cluster float* 3 k

void device_reduc(float* data, float* cluster, float* update, int k, int n) {
    for(int i=0; i<n; i++) {
        float min=65536*65, dis;
        float* mydata=data+i*DIM;
        int min_index=0;
        for (int i=0; i<k; i++) {
            dis = sqrt(
                (mydata[0]-cluster[i*DIM])* (mydata[0]-cluster[i*DIM])+
                (mydata[1]-cluster[i*DIM+1])* (mydata[1]-cluster[i*DIM+1])+
                (mydata[2]-cluster[i*DIM+2])* (mydata[2]-cluster[i*DIM+2])
            );
            if (dis<min) {
                min=dis; min_index=i;
            }
        }
        update[5*min_index] += mydata[0];
        update[5*min_index+1] += mydata[1];
        update[5*min_index+2] += mydata[2];
        update[5*min_index+3] += 1;
        update[5*min_index+4] += min;
    }
}
\end{verbatim}

Figure 3.5: User Input for k-means

in the function, and does not need to be returned. An output variable is to be returned from
the reduction function, as it is updated in the function. A temporary variable is declared
inside the reduction function for temporary storage. Thus, an input variable is read-only,
and output and temporary variables are read-write. Variables with different access patterns
are treated differently in declaration, result combination, and memory allocation strategies
described in the rest of this section.

Such information can usually be obtained from simple inspection of a function. However,
since we are supporting C language, complications can arise because of the use of pointers
and aliasing. In our implementation, first an Intermediate Representation (IR) is generated for the sequential reduction function with LLVM. Second, we used Anderson’s point-to analysis [11] to obtain the point-to set for each variable in the function’s argument list. Finally, we trace the entire function. When a store operation is found, if the destination of the store belongs to a points-to set of any variable in the function’s argument list, and the source is not in the same set, we conclude that it is an output variable. All the other variables in the argument list are denoted as input variables, and all the variables that do not appear in the argument list are considered temporary variables.

```
data input
update output
k input
n input
cluster input
```

Figure 3.6: Classification of Variables for K-means Reduction Function

As an example, let us consider the user input for k-means that we had shown earlier in Figure 3.5. The output obtained by analyzing the IR generated by LLVM for the reduction function is shown in Figure 3.6.

**Variable Analysis and Parallelization:** The variable analysis phase focuses on identifying how the reduction loop should be parallelized and if variables should be distributed or replicated.

We proceed by mapping the structure of the loop being analyzed to the canonical reduction loop we had shown earlier in Figure 3.3. We focus on the main outer loop and extract the loop variable. We also identify (symbolically) the number of iterations in the loop, and denote it as num_iter. If there are nested loops, for simplicity, we only parallelize the outer loop.
Next, we focus on the variables accessed in the loop. If a variable is only accessed with an affine subscript of the loop variable, it is denoted as a loop variable. Note that this variable could be an input, output, or temporary variable. The significance of a loop variable is that it can be distributed among the threads. All other variables need to be replicated, if they are written in the loop.

**Extracting the Combination Operations:** After local reduction is done by each thread, we need to combine their output variables, which are then copied to the host memory. Because we are focusing on reduction functions where output variables are updated with associative and commutative functions only (see Figure 3.3), the output variables updated by different threads can be correctly combined in the end. However, we need to identify the particular associative and commutative operator that is being used.

Earlier, we had generated the point-to sets for each parameter of the reduction function. We now conduct a new scan on the IR to find the reduction operator for each output variable. In the combination function, the values for a particular output parameter from each thread is combined using this function.

### 3.3.2 Mapping to GPGPU

Using the user input and the information extracted by the variable and code analyzer, the system next generates corresponding CUDA code and the host functions invoking CUDA-based parallel reductions.

**Grid Configuration and Kernel Invocation:** The host reduction function `host_reduc()` which invokes the kernel on device has 3 parts:

- **Declare and Copy:** We allocate device memory for variables to be used by the computing function on the GPU. We copy the ones that are needed to be read from host memory to device memory. Currently, we allocate memory for all variables except the temporary variables that are going to use shared memory. As we described earlier, loop variables are distributed across threads, depending upon how they are accessed across iterations. The read-write variables not denoted as loop might be updated simultaneously by multiple threads, so we create a copy for each thread. Again, because of the nature of the loops we
are focusing on, we can assume that a combination function can produce the correct final value of these variables.

**Compute:** We configure the thread grid on the device, and invoke the kernel function. Different thread grid configurations can be used for different reduction functions in one application. For example, in EM clustering, E phase and M phase can use different configurations. Currently, we configure the thread grid manually. In our future work, we hope to develop cost models that allow us to configure thread grids automatically.

**Copy updates:** We copy the variables needed by the host function. We perform the global combination for output variables which are not loop variables.

**Generating Kernel Code:** This task includes generating global function reduc() and device function device_reduc(), as well as device functions init() and combine(), if necessary. reduc() is the global function to be invoked by the host reduction function. It performs the initialization for the variables involves. The device main loop function device_reduc() is then invoked. Finally, one thread will execute combine() which performs the global combination. Between invocation of each function and at the end of reduc(), a _syncthreads() is inserted.

**Generating Local Reduction Function:** device_reduc() is the main loop to be executed on the GPU. This function is generated by rewriting the original sequential code in the user input, according to the information generated by the code and variable analyzer phases. The modifications include: 1) Dividing the loop to be parallelized by the number of blocks and number of threads in each block. 2) Rewriting the index of the array which are distributed. For example, we have an access to data[i], it is changed to data[i+index.n], where index.n is the offset for each thread in the entire grid. 3) Optimizing the use of shared memory, which we will discuss later.

### 3.4 Optimizations

We now describe two key optimizations that are implemented in our system.
3.4.1 Dealing with Shared Memory

Shared memory is a fast but very small read-write memory on the GPU. By making effective use of this memory, the performance of GPU applications can be improved dramatically. In various application studies that have been reported on GPUs, users have obtained significantly better performance with effective use of shared memory. However, because of its very small size, deciding which variables to put into shared memory is quite challenging.

Now we will describe the mechanisms we have developed in our system, to make the use of shared memory transparent to the programmers. First, the amount of shared memory that each array needs is calculated with the following expression:

\[ Size = length \times \text{sizeof(type)} \times \text{thread\_info} \]

Here, \( length \) is the length of this variable, \( type \) is one of \texttt{char}, \texttt{int}, \texttt{float}. The last factor \( \text{thread\_info} \) is 1 if \( \text{input} \) or \( \text{loop} \) is true, and \( n\_\text{threads} \) otherwise. It implies that if an array is read-write and not distributed over all threads, we need \( n\_\text{threads} \) copies of it.

To keep our system simple, we have focused on techniques that do not require advanced program analysis. The three schemes we have developed are as follows:

- **No sorting:** In this intuitive approach, the variable declarations are examined one by one. We simply allocate variables to shared memory as long as the memory requirements of all variables allocated do not exceed the total size of the shared memory.

- **Greedy-sorting:** Thus, in this approach, all the arrays are sorted with increasing order of their size. We select the variables to allocate onto shared memory from the beginning of this sorted array list, until the size of data on shared memory exceeds its limit.

- **Write-first sorting:** We found a non-intuitive optimization for the shared memory. By allocating variables that are updated in the reduction function at the lowest addresses in the shared memory, we can further improve performance. Thus, our write-first sorting is a variant of the greedy-sort strategy, where we insert variables that are written at the beginning of the sorted list.

3.4.2 Reducing Memory Allocation and Copy Overheads

Memory allocation and data movement overheads can be significant on GPUs. To enable optimization of these costs, we allow users to specify additional directives. Particularly, in
applications where a reduction function is invoked repeatedly, or where multiple reduction functions are invoked, user directives can help reduce memory allocation and data movement overheads.

As part of the input file, a user can use two directives, common and extern, to indicate the features of certain variables. When a variable is denoted as common, we allocate memory for this variable only in the first invocation of the reduction function, and not in subsequent iterations. Similarly, when a variable is declared as extern, it implies that the variable neither needs to be allocated in memory nor to be copied from host memory for this particular reduction function. This means that we expect a valid copy of this variables from the invocation of an earlier reduction function. For example, in our experiments with EM clustering, some of the variables can be declared as common for the E phase, and extern for the M phase. This is because an allocation and copy is needed only for the first invocation of the E phase reduction function, and not for M phase reduction functions, or subsequent invocation of E phase reduction functions. In the future, we will like to use inter-procedural analysis between the host function and various reduction functions to automate the identification of extern and common variables.

3.5 Experiments and Evaluation

This section reports on three data mining applications we ported on GPUs with our system. We also present a detailed evaluation study. Specifically, we had the following three goals in our experiments:

- Evaluating the overall performance of the system generated programs, measured as their speedup over a single threaded program executed on a CPU.

- Comparison of our system or automatic generated code with a manual version, to understand performance advantages or disadvantages of our approach.

- Evaluation of the benefits from a number of optimizations we have implemented in our system.
The sequential baseline executions were obtained on a Dell Dimension 9200 PC. It is equipped with Intel\texttrademark\(\textsuperscript{TM}\) Core\textsuperscript{TM} 2 E6420 Duo Processor with 2.13 GHz clock rate, 1GB Dual Channel DDR2 SDRAM memory at 667 MHz, a 4MB L2 cache and a 1066 MHz front side bus. The GPU versions used the same CPU, and a 768MB NVIDIA GeForce 8800 GTX, with 16 multiprocessors and 16KB shared memory on each multiprocessor. Some of our experiments were also performed using the GeForce 9800 GX2 card.

3.5.1 K-means Clustering

Clustering is one of the key data mining problems and k-means [84] is one of the most popular algorithms. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with \(k\) given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the \(k\) centroids from the points assigned to the corresponding center, and 4) repeat this process until the assignment of points to cluster does not change.

![Figure 3.7: Speedup of K-means on GeForce 8800 GTX](image-url)
The user input was shown earlier in Figure 3.5. In the variable description, $k$ is the number of clusters, $n$ is the length of the data block, \texttt{data} is the input data, and \texttt{update} stores reduction objects.

The performance of automatically generated programs on a 384 MB dataset is shown in Figure 3.7. All results are reported as a speedup over a sequential version execution on the 2.13 GHz CPU. Since the execution time does not change over iterations, we only show the execution time of the first 2 iterations. On the X scale, $n \times m$ implies executions with $m$ block and $n$ threads per block. The execution time on GPUs had two distinct components: the computation time, and the time spent moving data and results between the CPU and the GPU. We report two different speedup numbers. The computing speedups show the ratio between the execution time on the CPU and the computing time on the GPU. The computing with copy speedups show the ratio between the execution time on the CPU and the total execution time (including data movement time) using the GPU.

We also repeated the same experiment using GeForce 9800 GX2. The results are shown in Figure 3.8. The speedups are somewhat lower than that on GeForce 8800 GTX. This is because the memory bandwidth on 1 GPU of GeForce 9800 GX2 is lower than that on GeForce 8800 GTX. As there was only a small difference in the performance between these two cards, we only report results from 8800GTX card in the rest of this section.
Figure 3.9: Scalability of EM Application

The best speedups are nearly a factor of 50 over the CPU version. However, when the data movement times are included, the speedup decreases to nearly 20. Another observation is that the execution times of automatic versions are almost identical to the hand-coded version, showing that automatically generated code does not introduce any overheads. In fact, the only observable difference is with 1 block and 64 threads, and in this case, the automatic version is actually faster. This is because with a smaller number of threads, all replicated copies of centroids to be updated (the variable \texttt{update}) fit into the shared memory. The code generation system detected this feature and benefited from using shared memory. The manual version was designed to execute on all configurations, and because replicated copies of this variable cannot fit in shared memory with larger number of threads, this optimization was not performed.

The best performance is obtained with 256 thread per block and 16 or 32 blocks. More threads per block allows more concurrency. The maximum threads we can use in a block is 512, but this configuration does not obtain the best speedup, because of the larger amount of time that is spent on global combination. As there are 16 multiprocessors, best speedups are obtained with 16 or 32 blocks. Using a larger number of blocks only increases contention for resources, and does not allow any more parallelism.
The second data mining algorithm we have considered is also for clustering. Expectation Maximization (EM) is another popular clustering algorithm. The EM algorithm was first introduced in the seminar paper [51]. EM is a distance-based algorithm that assumes the data set can be modeled as a linear combination of multivariate normal distributions. There are several advantages to using EM for clustering data: it has a strong statistical
basis, it is robust to noisy data, it can accept the desired number of clusters as input, it provides a cluster membership probability per point, it can handle high dimensionality and it converges fast given a good initialization [131]. The goal of the EM algorithm is to estimate the means $C$, the covariances $R$ and the mixture weights $W$ of a Gaussian probability function [131]. The algorithm works by successfully improving the solution found so far. The algorithm stops when the quality of the current solution becomes stable, as measured by a monotonically increasing statistical quantity called loglikelihood. The sequential algorithm is shown in Figure 3.13.

We performed a scalability study, similar to the one we reported earlier for k-means, and the results are shown in Figure 3.9. We used a 12 MB dataset. All execution times are for 11 iterations. The best speedups are nearly 19, though when the data movement costs are included, they reduce to 12. The speedups are lower than what we obtained for k-means, because of a larger number of memory operations, and, relatively, less computation.

Earlier in Section 3.4, we had described several schemes for effectively using the shared memory. The automatically generated version whose performance we have reported is based on the use of scheme that performed the best, which is the write-first strategy. This also turns out to be the strategy that the manual version used. Overall, the two versions are almost identical in the compute time, but the manual version is slightly faster in the data copying time.
Next, we focus on examining the impact that different shared memory utilization schemes have on performance. The EM algorithm involves a number of distinct variables that are accessed with different patterns. Thus, unlike k-means and PCA, we notice significant differences from different strategies. The computing time of E and M phases using the 3 strategies are displayed in Figure 3.10. In the E phase, no-sorting is slower than the other two. This is because the other two strategies copied more variables onto shared memory. In the M phase, no-sorting again did not do well, but further, write-first strategy outperformed the greedy-sort strategy. The reason for this seems to be that this particular chip achieves better performance when data to be updated is stored at lower addresses in the shared memory. Overall, the total speedup in the computing time between the version that best uses the shared memory, and a version that does not use shared memory at all, is 40.

We also used the EM application to study the benefits from using optimized copying schemes. The results are shown in Figure 3.11. We can see that the execution time for both E and M phases is reduced by eliminating unnecessary memory operations. Particularly, the M phase copying costs are reduced to almost zero, as the input data block could be declared as extern, and their values can be reused from the values at the end of the E phase reduction function.

### 3.5.3 Principal Component Analysis

Principal Components Analysis is a popular dimensionality reduction method. This method was developed by Pearson in 1901. Our experiments are conducted on a modified version of F. Murtagh’s code\(^1\). Since it has many steps which are not quite compute-intensive, we only converted the creation of the correlation matrix to CUDA. Though we did not have a manual version for comparison, we did create a version with manually written combination function. This function was more efficient than the version automatically generated by our system.

The speedups on an input matrix of 256K rows, 80 columns are shown in Figure 3.12.

\(^1\)http://www.mirrorservice.org/sites/lib.stat.cmu.edu/multi/pca.c
Unlike the previous two applications, the best performance was achieved with 128 threads per block. This, in turn, was because of the increasing overhead of global reductions with a larger number of threads.

The optimized versions are the one with user-provided combination function. These versions are faster by nearly 20%, and show a limitation of the current program analysis and code generation performed by our system. The best speedups are nearly 24, though including the data movement costs, they reduce to 20. The speedups are higher than what we obtained from EM, but not as high as those from k-means. This is because of a higher fraction of memory accesses, and a relatively larger cost of global combination. This is also the reason that performance decreases rapidly when more than 16 blocks are used.

3.6 Summary

GPUs have been under research for its usage in general computation. However, because of the complication and hardware constraint, it is demanding to program in GPU. By focusing on data mining and data analyzing applications falling into certain structures, we were able to develop a transformation system which provides GPU support for programs written in sequential C. By using GPU, we got speedups between 20 and 50 over sequential versions, and between 12 and 20 when data movement cost is included. Those experiments showed the potential of our system for providing good performance while maintaining productivity.
Input: \( k \), \# of clusters, 
\( Y = \{y_1 \ldots y_n\} \), set of \( n \) \( p \)-dimensional points, 
\( \epsilon \), a tolerance for loglikelihood, 
\( \text{maxiterations} \), maximum number of iterations.

Output: \( C \), \( R \), \( W \), the matrices containing the updated mixture parameters. 
\( X \), a matrix with cluster membership probabilities.

Initialize: Set initial values for \( C \), \( R \), and \( W \) (random or approximate solutions)

WHILE: \( \delta(\text{llh}) > \epsilon \) and \( \text{maxiterations} \) has not been reached

DO E and M steps

E step
\[
C' = R' = W' = \text{llh} = 0 \\
\text{for } i = 1 \text{ to } n \\
\quad sump_i = 0 \\
\quad \text{for } j = 1 \text{ to } k \\
\quad\quad \delta_{ij} = (y_i - C_j)^t R^{-1} (y_i - C_j) \\
\quad\quad p_{ij} = \frac{w_i}{(2\pi)^{p/2}|R|^{1/2}} \exp\left(-\frac{1}{2} \delta_{ij}\right) \\
\quad\quad sump_i = sump_i + p_{ij} \\
\quad\text{endfor} \\
\quad x_i = p_i/sump_i, \quad \text{llh} = \text{llh} + \ln(sump_i) \\
\quad C' = C' + y_i x_i^t, \quad W' = W' + x_i \\
\text{endfor} \\

M step
\text{for } j = 1 \text{ to } k \\
\quad C_j = C_j'/W_j \\
\text{for } i = 1 \text{ to } n \\
\quad R' = R'+(y_i - C_j)x_{ij}(y_i - C_j)^t \\
\text{endfor} \\
\text{endfor} \\
\quad R = R'/n, \quad W = W'/n

Figure 3.13: Sequential code for the Expectation Maximization Algorithm
CHAPTER 4
AUTOMATIC TRANSLATION OF DATA MINING APPLICATIONS FROM MATLAB TO GPUS

In previous chapter, we introduced a system to convert data mining applications written in C into CUDA. In reality, when creating data mining implementations, scientists and engineers will clearly prefer to use programming tools they most widely use. As a result, MATLAB has been popular for teaching\(^1\), developing\(^2,\,^3\) and distributing\(^4\) data mining algorithms. Ahalt \textit{et al}. have stated that “The mathematical and interactive nature of many of the data mining algorithms, makes it natural to use a language like MATLAB both to design algorithms and for post-processing of the results”\(^5\).

On the other hand, MATLAB based data mining implementations today are only useful for small data sets that easily fit in memory, and can be processed interactively on a single processor. Even though there has been a lot of work on parallel MATLAB\(^{[42]}\), there is no transparent MATLAB system which can support development of data mining algorithms in parallel, and on large data sets. A recently released commercial product, JACKET, targets the use of GPUs on MATLAB \(^{[107]}\). This system requires that the programmers explicitly

\(^{1}\)http://www.ncrg.aston.ac.uk/netlab/
\(^{2}\)http://eprints.usq.edu.au/58
\(^{4}\)http://www.spatial.cs.umn.edu/sdm.html
\(^{5}\)http://www.ll.mit.edu/HPEC/agendas/proc03/abstracts/khot_abstract.pdf
declare the data structures for GPU, and does not perform any advanced transformations. Earlier, there had been a lot of work on compiling and optimizing MATLAB, including type, rank, and size analysis [145], JIT compilation [8], storage optimizations [88], and parallelization [36]. But, advanced transformation to automatically generate efficient GPU programs have not been considered.

This chapter reports on a system we refer to as GMAT-DM (Gpgpu from MATLAB for Data Mining applications), which is an automatic translation tool to convert data mining applications written in MATLAB to CUDA, for execution on GPUs. The key technical challenge we need to address is achieving memory efficiency. MATLAB provides a large number of operations and functions that make data mining algorithm implementation easier. However, this convenience often leads to inefficient usage of memory. We have developed and implemented a number of optimizations to achieve memory efficiency. Our tool GMAT-DM builds on the system introduced in Chapter 3, which converts C programs with additional annotations to CUDA [110].

We evaluated the GMAT-DM system using two applications, k-means clustering and EM clustering. The main observations from the experiments are as follows. We are able to achieve performance close to that of C programs automatically translated to CUDA, our optimizations help improve the performance by up to a factor of 3, and automatically generated CUDA code can achieve speedups between 8 and 38 over sequential CPU-based C codes.

4.1 System Design

This section gives an overview of our translation system. The main optimization framework implemented in our system is described in the next Section.

There have been quite a few efforts on parallelizing MATLAB programs. In an extensive survey by Choy, Edelman and Moler on this topic [43], parallel MATLAB systems are categorized into four classes: Embarrassing parallel, Message passing, Back-end support, and MATLAB compilers. Our system is a compilation based system. We translate MATLAB
code into C programs in a way that supports processing on GPUs well, and apply optimizations that are suitable for GPU computing. Particularly, we have developed methods for translating matrix operations and functions in MATLAB to memory-efficient C code.

A high-level overview of GMAT-DM is shown in Figure 4.1. We use OCTAVE as the framework to process MATLAB code [55]. OCTAVE is an open source software capable of interpreting most common MATLAB programs. During the parsing of MATLAB code, the code generating functions we have inserted get invoked. Actual processing of MATLAB programs is not performed in the system. As we can see from Figure 4.1, the C code is translated into CUDA using GREENRIDE, the code transformation system we described in Chapter 3 [110]. In the following sections, we will mainly explain how the C code is generated, and optimized for the usage of GPU.

Shape, type, and rank analysis are important problems in translating MATLAB code, and they have been well studied by earlier MATLAB compilation projects (for example, [145]). OCTAVE has its own type inference mechanism, which is based on using the C++ type dispatch. However, since evaluation is removed from OCTAVE in our framework, we can not use this strategy. So, our system takes a simple approach to this problem, based on adapting the prior work. This simple implementation turns out to be sufficient for the programs we are considering.
4.2 C Code Generation and Optimization from MATLAB

This section focuses on the main transformations and optimizations that our framework performs. Initially, we use an example to motivate the transformation framework we have implemented.

4.2.1 A Motivating Example: K-means Clustering

```c
for(int outi=0;outi<n;outi++)
{
    float min=65536*65, dis;
    int min_index=0;
    for (int i=0;i<k;i++)
    {
        for(int m=0;m<3;m++)
            dis +=(data[outi][m]-cluster[i][m])*(data[outi][m]-cluster[i][m]);
        dis=sqrt(dis);
        if (dis<min)
            {
                min=dis; min_index=i; }
    }
    update[5*min_index] += data[outi][0];
    update[5*min_index+1] += data[outi][1];
    update[5*min_index+2] += data[outi][2];
    update[5*min_index+3] += 1;
    update[5*min_index+4] += min;
}
```

(a) C-based Reduction Function for k-means

```matlab
for i = 1:n
    for j = 1:k
        distances = sum((mydata(i,:) - cluster(j,:)).^2, 2);
        d(i,j) = sqrt(distances);
    end
end
[z,g]=min(d,[]); % MATLAB
```

(b) Reduction Function for k-means in MATLAB

Figure 4.2: K-means in C and MATLAB
Clustering is one of the key data mining problems and k-means [84] is one of the most popular algorithms. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with \( k \) given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the \( k \) centroids from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change.

Figure 4.2, sub-figure (a) shows the main reduction function expressed in C. As we have shown in our previous work [110], such a function can be parallelized in CUDA by creating a copy of the \texttt{update} array for each thread, dividing the iterations of the main loop among the threads, and introducing a global reduction function.

Figure 4.2, sub-figure (b) shows the same loop expressed in MATLAB. This implementation of k-means in MATLAB is based on a tutorial by Toknomo\(^6\). Comparing sub-figures (a) and (b), the main difference in the two implementations is that distances between each pair of the input point \texttt{mydata} and clusters \texttt{cluster} is now computed and stored in a two dimensional matrix \texttt{distances}. The C-based implementation, in comparison, uses a single scalar \texttt{dis}. Because matrix operations are a key aspect of MATLAB, it is convenient for programmers to use them. However, as we have seen in this example, such a use of matrices increases the memory requirements of the program very significantly. Depending upon the number of dimensions in the data and the number of clusters, the array \texttt{distances} here could be even larger than the size of the input data.

While this difference between C and MATLAB is not specific to GPUs, it is a key challenge in mapping MATLAB-based programs to GPUs. This is because the GPUs have very high processing power, but the memory tends to be limited. For example, the NVIDIA Fermi card has 14 multiprocessors, each of which has 32 processing cores, but it has only 3 GB of device memory. Furthermore, one important optimization in CUDA is making use

\(^6\)http://people.revoledu.com/kardi/tutorial/kMean/matlab_kMeans.htm
of shared memory, a software-controlled cache. Though shared memory can be accessed faster, its size is extremely small. Thus, translating matrix operations to achieve memory efficiency is a critical challenge we address in this chapter.

### 4.2.2 Problem Formulation

In our optimization framework, we consider MATLAB statements of the following form:

\[
\text{result} = f(\text{exp})
\]

Here, \(\text{exp}\) can be as follows:

\[
\text{exp} = X_1 \text{ op } X_2 \text{ op } \cdots \text{ op } X_n; (n \geq 1)
\]  

(4.1)

where, \(\text{op}\) can be “+”, “-”, or another element-to-element operator. Each \(X_i\) is:

\[
T_1 \ast T_2 \ast \cdots \ast T_k (k \geq 1)
\]  

(4.2)

Every \(T_i\) is a vector or matrix, a function call \(f\) with a return value, or an expression.

The function \(f\) we refer to in the discussion above is among the following. It could be an element function or a dimension-aggregation function. The first group includes MATLAB functions like \(\text{norm}, \text{isinf}, \text{isnan}, \text{finite}, \text{tril}, \text{triu}, \text{diff}\) and \(\text{find}\). Those functions apply the same computation on each element, and therefore, no aggregation is needed at vector level. The second group includes \(\text{min}, \text{max}, \text{all}, \text{sum}\) and \(\text{any}\). When any of these functions are applied on a matrix, one value is extracted by applying an aggregating operation on each row or each column of the matrix.

### 4.2.3 Solution Approach

We present our solution in two steps. First, we focus on matrix operations only, and next, we consider application of different functions.

#### Optimizing Code for Matrix Operations

In this section, we explain translation strategies for expressions with the Form (4.1). We use the example of the M phase in the EM clustering algorithm. The MATLAB code for this is shown in Figure 4.4, sub-figure (a). The intuitively translated C code is shown
for (i = 0; i < maxRow; i++)
    for (j = 0; j < k; j++)
    {
        /* code to compute distances */
        d[i][j] = sqrt(distances);
    }
for(int i1 = 0; i1 < maxRow; i1++)
    {
        z[i1] = 0;
        g[i1] = MAX_VALUE;
        for(int i2 = 0; i2 < k; i2++)
        {
            if(z[i1] > d[i1][i2])
            {
                z[i1] = d[i1][i2];
                g[i1] = i2;
            }
        }
    }

Figure 4.3: Generated C code for k-means with Naive Algorithm

in Figure 4.4, sub-figure (b). Particularly, we will focus here on the expression $V + E(j, i) \ast dX M_M \ast dX M_M'$ (Line 4 of sub-figure (a)).

For an expression of the form (4.1), a loop can be generated. The expression inside the loop body is in the format of the expression in Form (4.3).

\[
result[i] = (X_1[i] \ op X_2[i] \ op \cdots \ op X_k[i]),
\]

(4.3)

For the M phase of EM clustering, the code generated after this step can be seen from Figure 4.4, sub-figure (c), where $X_1$ represents $E(j, i) \ast dX M_M \ast dX M_M'$. Now, let us consider the case where an $X_i$ is in Form (4.2). First, we need to decide the sequence of matrix multiplications. We use a modified version of the well-known solution to the classical matrix chain multiplication problem. The original dynamic programming solution is as follows [2]: For a certain chain $C$ with $k$ matrices, the problem is solved by iterating with the chain length ranging from 2 to $k$. At a particular iteration, with the chain
length being \( l \), the multiplication is performed by splitting \( C \) into two parts, \( C_1 \) and \( C_2 \). First, \( C_1 \) and \( C_2 \) are individually computed, and then, the operation \( C_1 \times C_2 \) is performed. Thus, there are \( l-1 \) ways to split the chain. A metric is evaluated for each potential splitting method, and the split with the minimum value is chosen. In the traditional algorithm, the metric used is the count of the multiplication operations on individual elements. With our focus on code generation for GPUs, we modified this algorithm by using a weighted sum of two metrics. Besides the number of multiplication operations, the second metric we use is the size of the temporary memory usage. The reason is that smaller temporary intermediate matrices are more likely to be placed on the faster \textit{shared memory}. Note that on GPUs, this \textit{shared memory} can be accessed about 100~150 times faster than the \textit{device memory}.

The following example shows how this approach selects the multiplication order with efficient memory usage. Suppose we want to multiply a chain of matrices: \( A \), \( B \), and \( C \). The dimensions are \( A(6 \times 100) \), \( B(100 \times 2) \), and \( C(2 \times 1) \). For this chain, there are 2 different orders to perform the multiplication: \((A \times B) \times C\) and \(A \times (B \times C)\). The number of multiplication operations for the first case is \(6 \times 100 \times 2 + 6 \times 2 \times 1\), or, 1212. For the second case, this metric evaluates to \(100 \times 2 \times 1 + 6 \times 100 \times 1\), or 800. Thus, if we use operation count as the metric, we will choose the second option.

However, let us consider the temporary memory usage. The intermediate result with the first case is a \( 6 \times 2 \) matrix, so the usage is 12 words. Similarly, the memory usage for intermediate results in the second case is 100 words. However, if this computation is done by every thread on the GPU, then all the temporary matrices that are updated by the threads need to be replicated for each thread. Thus, the memory usage increases by a factor of 256, if 256 threads are executed concurrently. Thus, it may make sense to choose the first option, where we could fit the intermediate results in the \textit{shared memory}.

After the multiplication order is decided, the code for all the multiplications, except the last one, is inserted before the loop of the expression of the form (4.3). Then, the last multiplication is fused into this loop. In the expression (4.3), if \( X_i \) is in the form of (4.2), \( X_i \) is substituted by the final result of the multiplication chain in \( X_i \). In the example with the M phase, the generated code is as shown in Figure 4.4, sub-figure (d). We can see that
the code for multiplication of $E(j, i) \times dXM_M \times dXM_M'$ is fused into the outer loop. $E(j, i)$ serves as a coefficient of $dXM_M$.

**Combining Functions to Save Temporary Storage**

Suppose there is a function $f$ that is applied on a matrix $M$, as part of an expression we are considering. If the matrix $M$ is an input or output variable, we will need to allocate space for it. Thus, we can simply evaluate the function $f$ on all elements of the matrix $M$. However, we can reduce the space requirements for the cases where $M$ is a temporary variable. We can avoid computing $M$ for such cases, and apply the function $f$ on values as they are generated.

In naive code generation, the matrix $M$ is evaluated with two nested loops, an inner-loop, and an outer-loop. For *element functions*, the function can be applied over the inner-loop. For *dimensional-aggregation* functions, the function can be evaluated during each iteration of the outer-loop (with loop reordering, if needed).

Let us now revisit the k-means example that was shown at the beginning of this section. We had shown the MATLAB code in Figure 4.2, sub-figure (b). The C code with naive translation is shown in Figure 4.3. When this code is executed on GPUs, the temporary matrix $d$ cannot fit in shared memory, assuming $maxRow$ is sufficiently large. Thus, the reference to $d$ will result in a significant overhead. Since $d$ is a temporary variable, we can eliminate it by combining the function $min$ with the loop before it.

### 4.2.4 Other Optimizations

Several other optimizations to reduce the memory requirements were also performed in our framework. One issue is dealing with a column and row expression like $m(i,:)$. Instead of allocating a new memory block for this row or column, we refer to the original matrix with an appropriate index. Matrix transposing can also cause large memory overheads. In GMAT-DM, to reduce this overhead, we also use modified index instead of actual memory allocation.

Another optimization is in the situation when the same element in an array is accessed in many iterations of a loop. On GPU, it will make a significant difference if access to this element is changed to a *register reference*. This is because accessing device memory is
much slower than register reference. Thus, a renaming strategy can be used to eliminate the extra memory overhead.

4.3 Experiments and Results

This section reports on the experiments we conducted to evaluate our transformation framework and optimizations. We so far used two data mining applications, which are k-means clustering and EM clustering. MATLAB version of k-means we use is as shown earlier in Figure 4.2, which is based on a tutorial by Toknomo. The EM MATLAB code was obtained from a distribution by Mathworks.

These MATLAB programs were automatically transformed to C (with added annotations) by the GMAT-DM framework. Multiple versions were generated to reflect the different optimizations. These versions, together with an original C input to GREENRIDE used in our previous work as described in Chapter 3 [110], were translated to CUDA by GREENRIDE. In reporting our results, different versions are referred to as follows. C means a CUDA program generated from a user-written C code by GREENRIDE. MATLAB means CUDA program generated from MATLAB code by GMAT-DM and GREENRIDE. unoptimized and optimized refer to MATLAB converted code with the straight-forward method and with optimizing strategies, respectively. Further, to understand the performance, we report different breakdowns of execution times. Here, with copy denotes the time spent computing and the time spent copying data to the device memory. without copy is the time spent only on the computation. Since the interpreted and sequential MATLAB code is too slow (more than 5 orders of magnitude slower than a GPU-based version), our experiments do not include comparisons against interpreted MATLAB on CPU.

The GPU we used for our experiments is a NVIDIA Fermi GPU, with 14 multiprocessors, 3 GB device memory, 48 KB shared memory (as configured for our experiments) and 16KB L1 cache on each multiprocessor. Most of our results are reported as speedups over a baseline sequential versions. The sequential baseline executions were obtained on a Quad-Core Intel Xeon E5520 CPU, with the frequency of 2.27 GHz.

7http://www.mathworks.com/matlabcentral/fileexchange/8636
4.3.1 Results from K-means Clustering

The experiments we report were conducted on a 168 MB representative data set. The results, reported as speedups of various versions over a sequential CPU-based code are shown in Figures 4.5 and 4.6, depicting timing without and with data copy times between the host and the device memory. In all figures shown in this section, the numbers on X axis are the grid configuration. For example, 256•14 implies that we are using 14 blocks of threads, with 256 threads in each block.

Initially, let us consider the three versions in Figure 4.5, which do not include the time spent on copying the data. It turns out that the automatically translated and optimized code takes less time computing on GPUs than the original C code. However, the automatically translated code requires significantly more copying between CPU and GPU, as we will show below, and thus, it is slightly slower overall. In Figure 4.6, the C-manual with copy version has a speedup of about 13 in the best case. The optimized with copy version is only about 35% slower. The difference arises because the hand written CUDA codes use somewhat different algorithm.

4.3.2 Results from EM

The second application we consider is EM clustering. We used a 42 MB data set. This application involves two distinct reduction phases, which are the E phase and the M phase, respectively. The speedups over CPU sequential versions for these two phases are shown in Figures 4.7 and 4.8.

In Figure 4.7, we have compared five different versions, two of which include the data movement costs. The optimized version is again very close to the C-manual version. Further, the improvement from our optimizations is close to 15%. Finally, even after including the data movement costs, the optimized with copy version automatically generated from MATLAB has a speedup of more than 16 over the sequential version.

The results from the M phase of this application are shown in Figure 4.8. Here, we include versions with two different levels of optimizations. optimized-1 only uses matrix operation combination, whereas optimized-2 uses renaming as well. The two sets of
optimizations improve the performance by a factor of 5.7. After the optimizations, the automatically generated version has a speedup of about 38.6 over a CPU version. Even after including the data movement costs, the automatically generated code can achieve a speedup of about 37.5 over a sequential version. The longer computing time of the automatically generated code is because of the column-major storage used for matrices in MATLAB. In the future, we will consider transforming the layout of matrices automatically to further improve performance.

4.3.3 Comparison with GPU version of MATLAB

Jacket [107] is a software providing interface for GPU programming with MATLAB. With Jacket, by denoting certain loops, operations, and functions with the symbol “g”, MATLAB code can be executed on the GPU. We implemented kmeans with Jacket. The execution time on the GPU with the 168 MB dataset is about 5.3 times longer than the sequential CPU version that we compared with. Thus, overall, Jacket generated GPU code is more than 50 times slower than the code we generate. This observation confirms that our optimizations are very effective for the applications we are targeting.

4.4 Summary

Bridging the gap between very high-level languages and modern multi-cores and accelerators is a tremendous challenge for compilers and run-time systems. We made our contribution to this research area by providing a code translation system for data mining computations written in MATLAB to be ported to GPUs. Since our work targets the particular class of applications with the common reduction structure in Figure 3.3, we were able to make use of GREENRIDE, which was the system described in Chapter 3. MATLAB code is translated into C for the purpose of being converted to CUDA. We have shown that these applications can be translated automatically for execution on GPUs, and can achieve good scalability. We have developed a number of optimizations. By reducing memory usage and making efficient use of shared memory on GPU, those optimizations improved the performance very significantly. We have also shown that by focusing on matrix operations and functions in
MATLAB, such optimizations can be performed without extensive dependence on program analysis.
for $i = 1 : k$
for $j = 1 : n$
\[ dX_M M = X(j,:) - M(:,i); \]
\[ V = V + E(j,i) \ast dX_M M \ast dX_M M'; \]
end
end

(a) MATLAB Code for M phase of EM
for ($i = 0; i < k; i++$)
for ($j = 0; j < n; j++$) {
  for(int $i_1 = 0; i_1 < d; i_1++$)
    \[ dX_M M[i_1] = X[i_1][j] - M[i_1][i]; \]
  for(int $i_3 = 0; i_3 < d; i_3++$)
    tmp2[$i_3$][$i_4$] = 0;
    for(int $i_2 = 0; i_2 < 1; i_2++$)
      \[ tmp2[i_3][i_4] = E[j][i] \ast dX_M M[i_4] \ast (1 + i_2) \ast dX_M M[i_3 + i_2 \ast d]; \]
  for(int $i_6 = 0; i_6 < d; i_6++$)
  for(int $i_7 = 0; i_7 < d; i_7++$)
    \[ V[i_6][i_7] = V[i_6][i_7] + tmp2[i_6][i_7]; \]
\}

(b) C code with Naive Translation
for ($i = 0; i < k; i++$)
for ($j = 0; j < n; j++$) {
  for(int $i_1 = 0; i_1 < d; i_1++$)
    \[ dX_M M[i_1] = X[i_1][n + j] - M[i_1][k + i]; \]
  for(int $i_3 = 0; i_3 < d; i_3++$)
    \[ V[i_2][i_3] = V[i_2][i_3] + X_1[i_2] \ast d + i_3; \]
\}

(c) C code with First-level Transformation
for ($i = 0; i < k; i++$)
for ($j = 0; j < n; j++$) {
  for(int $i_1 = 0; i_1 < d; i_1++$)
    \[ dX_M M[i_1] = X[i_1][j] - M[i_1][i]; \]
  for(int $i_3 = 0; i_3 < d; i_3++$)
    \[ V[i_2][i_3] = V[i_2][i_3] + tmp2; \]
\}

(d) C code with Matrix Operation Combination

Figure 4.4: M phase in EM Clustering
Figure 4.5: K-means: Speedup of Different Versions (Copy time Excluded)

Figure 4.6: K-means: Speedup of Different Versions - Copy Time Included
Figure 4.7: EM: Speedups of E Phase

Figure 4.8: EM: Speedups of M Phase
Though we have provided a system to automatically generate GPU code for common data mining and data analysis applications, our work had been limited to single PCs. Today, a popular parallel processing configuration is a cluster of machines, with an accelerator like GPU on each node. Even a small cluster of nodes with GPUs can have a very high peak performance. GPU clusters have received a significant attention lately in the parallel computing community [80, 56, 116].

However, obtaining performance improvement from such clusters involves a very difficult programmability challenge. Clusters have traditionally been programmed using MPI, whereas GPUs are programmed using CUDA or OpenCL. While both MPI and CUDA have been popular, they both require low-level and explicit parallel programming. Thus, developing a highly tuned application for a cluster with GPUs requires a lot of programming effort, besides requiring expertise in both. It will clearly be desirable to have compilation and/or run-time systems that can enable higher-level programming of such clusters.

As mentioned in Section 3.2.1, many data mining and data analysis applications follow a common programming structure, which is Figure 3.3. Thus, we developed a compiler solution that enables automatic code generation for clusters with GPUs for this class of algorithms, which we refer to as AUTO-GC [111]. In our approach, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the variables. We use program analysis and code generation to map the applications to a distributed memory cluster, and further accelerate the processing by using the GPU.
For the former, our code generation system generates API code for a middleware system, Freeride, which we had developed in our previous work [86, 87].

We have evaluated our system using two popular data mining applications, k-means clustering and Principal Component Analysis (PCA). The main observations from our experiments are as follows. The automatically generated version did not have any noticeable overheads compared to handwritten codes, and has good scalability over the number of computing nodes. The usage of GPU gives a speedup of between 3 and 21, over the parallel code executing just on CPUs.

5.1 System Design

This section describes the code generation system, AUTO-GC. Initially, we give an overview. This is followed by the system API, and details of program analysis and code generation for cluster and GPUs.

5.1.1 System components

The overall configuration we consider is as follows. The data files to process are distributed among the computing nodes. On each node, when processing the data blocks, the main computing task, which is implemented as the reduction function, can be executed by the GPU. To enable this, our tool, AUTO-GC, generates both the Freeride API code and the CUDA code.

The system design is shown in Figure 5.1. There are three components in the user input: variable information, reduction function(s), and additional optional functions. AUTO-GC comprises two components, a program analyzer and a code generator. The program analyzer includes the variable analyzer and the code analyzer. The code analyzer obtains variable access patterns and extracts the reduction objects, with a combination operation. The variable analyzer extracts variable information which is an input to the code-generator, based on the user input and the code analysis. Same as in Chapter 3, we used LLVM as the framework for program analysis [99]. After getting the variable information and reduction objects, the code-generator generates the code for the Freeride API, and CUDA code.
for the computation on GPUs. The CUDA functions are invoked by the FREERIDE code for executing the reduction function.

5.1.2 System API

Before discussing the program analysis and code generation, we describe the API of the system, i.e., the input that needs to be provided by an application developer.

Using the generalized reduction structure of our target class of applications, we provide a convenient API for the user, as used in Chapter 3. The format of input for a reduction function is shown in Figure 5.2. If there are multiple reduction functions, a user can specify them by including labels for each. For each function, the following information is needed.

Variables for Computing: As shown in Figure 5.2, the declaration of each variable follows the following format:

\[ \text{name, type, size}[\text{value}] \]

This is the same as the format we used in Section 3.2.2.
Variable information:

variable declare:
variable declare:
......
variable declare.

functions // reduction and some optional functions

variable declare:

name; type; length

Figure 5.2: Format of the User Input

1
kmeans
step int 0
endcondition int 0
MSE float 999999e+20
k int
n int
data float* n 3
update float* 5 k
cluster float* 3 k

Figure 5.3: Variable List in the User Input for K-means

Sequential Reduction Function: The user can write the sequential code for the main loop of the reduction operation in C. Any variable declared inside the reduction function should also appear in the variable list as shown in Figure 5.2, and memory allocation for these variables is not needed.

User defined Finalize Function: After the reduction objects are combined at the end of each iteration, there might be some extra work to do with the reduction objects. This work can be done by providing a finalize function.

Optional Initialization and Combination Functions from the User: Still same as in Section 3.2.2, these functions are provided by user who are familiar with CUDA programming.
5.1.3 Program Analysis

There are two main components in the program analyzer, the code analyzer and the variable analyzer. The code analyzer accomplishes two tasks: obtaining the access pattern and extracting the reduction objects with their combination operation. These two tasks are performed in the following way:

**Obtaining Variable Access Features:** We classify each variable as one of **input**, **output** and **temporary**. An **input** variable is input to the reduction function, which is not updated in the function, and does not need to be returned. An **output** variable is updated and to be returned in the reduction function. A **temporary** variable is declared inside the reduction function for temporary storage. Thus, an **input** variable is **read-only**, and **output** and **temporary** variables are **read-write**. Variables with different access patterns are treated differently in declaration, memory allocation strategies, and result combination, as described in the rest of this section.

Such information can usually be obtained from simple inspection of a function. However, since we are supporting C language, complications can arise because of the use of pointers and aliasing. In our implementation, we used the method introduced in Section 3.3.1 to extract the access features of each variable.

As an example, let us consider the user input for k-means. The variable list is shown in Figure 5.3 and the reduction function is the same as Figure 3.5. The output obtained by analyzing the IR generated by LLVM for the reduction function is shown in Figure 3.6.

**Extracting Reduction Objects and Combination Operations:** The **output** variables are identified as the reduction objects. At the end of each iteration, the reduction objects on each node are combined into a single one, by using the MPI calls automatically invoked by FREERIDE. Because we are focusing on reduction functions where **output** variables are updated with associative and commutative functions only (see Figure 3.3), the **output** variables updated by each computing node (and different threads in GPU) can be correctly combined in the end. However, we need to identify the particular operator that is being used. Earlier, we have generated the **point-to** sets for each parameter of the reduction function when extracting access features for variables. We now conduct a new scan on the
IR to find the reduction operator for each reduction object. In the combination function, the values for a reduction object from each thread is combined using this function.

After the above information has been extracted, the variable analyzer will proceed to summarize the variable information and extract the parallel loops.

**Analysis for Parallelization** We map the structure of the loop being analyzed to the canonical reduction loop we had shown earlier in Figure 3.3. We focus on the main outer loop and extract the loop variable. We also identify (symbolically) the number of iterations in the loop, and denote it as num_iter. If there are nested loops, for simplicity, we only parallelize the outer loop.

The variable analyzer, which is the other component of the program analyzer, focuses on the variables accessed in the loop. If a variable is only accessed with an affine subscript of the loop variable, it is denoted as a loop variable. Note that this variable could be an input, output, or temporary variable. The significance of denoting it is that when run on GPU, a loop variable can be distributed among the threads, while all the other variables need to be replicated, if they are written in the loop.

### 5.1.4 Code Generation for FREERIDE

The issues in generating code for FREERIDE API are as follows. The base class for any application is a template FREERIDE TECH. For a particular application, we derive its corresponding class from FREERIDE TECH, with the variables in each reduction function declared as class members. There are three main functions in the class. We discuss the code generation for each of them as below.

**Initialization:** After variable analysis, we already know which variables form the reduction object. In the Initialization() function, these variables are declared and initialized with the default values given by the user. Arrays are allocated with the given size. One thing that needs attention is the reduction objects that are to be computed with CUDA. Since each thread needs its own copy, the size of the variable is the declared as size multiplied with a block number and a thread number within the block. The block here refers to thread block in CUDA.

**Reduction:** The Reduction() function is the main processing function for the data blocks.
The computation in the sequential reduction function given by the user is included in this function. At the end of the function, the reduction objects are updated with the output of the local reduction. For each reduction function, the user can denote whether to use GPU or not in the input file. If GPU is chosen, a CUDA version for the reduction function is generated, as described in the next subsection.

**Finalize:** As described previously, after one iteration, every data block has been processed, and the reduction objects have been combined with MPI message passing at the back end. Thus, they are copied to the corresponding local variables, and the user provided functions are added after that, if any.

```cpp
void reduc_class::kmeans(void *block)
{
    float* data=(float*)block;
    kmeans_func(step,endcondition,k,n, MSE,data,update,cluster);
    for (int RO_i=0;RO_i<1;RO_i++)
    {
        for (int RO_j=0;RO_j<1*5*k;RO_j++)
            reductionobject->reduction(RO_i,RO_j,update[RO_j]);
    }
}
```

**Figure 5.4:** System Generated Reduction Function of K-means

To show how the code generation is done, let us take k-means as an example. The user input has two parts, which are the variable list in Figure 5.3 and reduction function in Figure 3.5. Figure 5.3 is the variable description, where `step`, `endcondition` and `MSE` are used in testing for termination of the execution, `k` is the number of clusters, `n` is the number of points in the data block, `data` is the input data array, and `update` stores the updates to each cluster, including count, distance, and accumulated point coordinates.

After code analysis, we find that `update` is an output variable, so it is determined as the reduction object for this reduction function. Then, reduction object is allocated according
to its size. In the system generated code, \textit{reductionobject} is updated with the value of \textit{update}, as shown in Figure 5.4.

The CUDA code is generated with the GREENRIDE system, which is described in Chapter 3, according to the analysis and user input. The generated CUDA functions are invoked by the \textit{reduction} function in FREERIDE, and the values of reduction objects are copied back to the appropriate variables in the host program.

5.2 Experiments

This section presents an evaluation study with our code generation system, using two popular data mining algorithms. Specifically, we had the following three goals in our experiments:

- Evaluating the overall performance and scalability of the system generated programs, including evaluating gains from using GPUs on each node of a cluster, over the performance on just the cluster.

- Comparison of our automatically generated code with a hand-written or manual version, to quantify the overheads of our approach.

- Comparison of the impact of using different computing devices (CPUs and GPUs) for different data set sizes.

Our experiments were conducted on a 8 node cluster with AMD Opteron 8350 machines, each of which is equipped with a GeForce 9800 GX2 graphic card. The amount of memory on each node is 16 GB, and the interconnect network in the cluster is Infiniband.

5.2.1 K-means Clustering

The code generation for k-means was explained with several code examples in Section 5.1.4. We conducted our tests with three versions: a manually written FREERIDE program that could only use the CPU on each node of the cluster, a manually written FREERIDE application with manually written CUDA code, and finally, a system generated code, where both the FREERIDE API and CUDA codes are automatically generated. These three versions are denoted as CPU, GPU-manual and GPU-automatic.
We experimented with two different data sets, which were 1.5 GB and 3 GB, respectively. The number of dimensions is 3 and the parameter \( k \), the number of clusters to be obtained, is set at 10. In the figures, we only show the performance with the best configuration for all the CUDA version, which is 16 thread block and 256 threads per block.

The first set of results are from a data set with 1.5 GB, and are shown in Figure 5.5. We can see that each of the three versions has a good scalability, as the number of nodes in the cluster is increased. The automatically generated CUDA code has almost the same performance with the manual CUDA code, which shows that we can generate very efficient code for programs that fall in the structure we specified in Figure 3.3. The GPU versions have a speedup of more than 5, over the CPU version, when we are using 4 nodes or less. The relative speedup, over the CPU version using the same number of nodes, reduces to 3 when 8 nodes are used. This is because the amount of data to be processed on each node becomes smaller as the same data set is processed on more nodes. Thus, the execution time gets more dominated by the overheads of initiating the processing on the GPUs, and the global combination time. In Figure 5.6, the results are shown for the 3 GB data set. The relative performance of different versions follows a similar trend.
5.2.2 Principal Component Analysis

The algorithm of PCA was introduced in Section 3.5.3. We did not provide a manual CUDA version for PCA, since the functions to be converted to CUDA are relatively simple, and the automatic generated CUDA will be very similar to hand written versions. Thus, the two versions we compare with PCA are as follows. One is the FREERIDE-based CPU version, without using GPUs, denoted as CPU, which is written manually. The other is the system generated CUDA version, denoted as GPU-automatic. In GPU-automatic, we generated FREERIDE APIs, and CUDA code for the computing of mean, standard deviation and the entire correlation matrix. For all the GPU executions, we used the best configuration, which is 128 threads per block, and 16 blocks in total for one GPU.

Figure 5.7 shows the performance of the two versions on a data set with 64 M rows and 3 principal components. The CPU-based FREERIDE code has good scalability as the number of nodes is increased. The CUDA version also scaled well, but the performance is worse than the version without GPU. This is because of the domination of I/O with these parameters. Without much computation, the use of GPU is not helpful.

In Figure 5.8, we use a data set with 2M rows and 64 principal components, and projection is done on the first 6 components. With these parameters, PCA is extremely compute-intensive. Thus, the benefits of using GPU are very significant. The performance of GPU-automatic on 1 node has a relative speedup of about 21 over the CPU versions on 1, 2, 4, and 8 nodes.

From the experiments, we can see that the benefits of using the GPU can vary widely depending upon the nature of the application and the parameters. In the future, we will like to develop cost models that can predict whether or not moving an application to a GPU will be beneficial.

5.3 Summary

We introduced a system to generate FREERIDE code with GPU support, which is a system for parallel processing of data mining and scientific data analysis applications. The common
processing structure is extracted for user provided sequential functions, then we produce optimized C and CUDA code within the FREERIDE framework.

We have evaluated our system using two popular data mining applications, k-means clustering and Principal Component Analysis (PCA). We got good scalability with the number of computing nodes, and satisfactory speedup with the usage of CUDA. The code automatically generated by our system did not have any noticeable overheads compared to hand written codes. We also leave the option for the user to choose the device to use, which will be done automatically in future.
In Chapter 3, we used an intuitive approach to arrange the data in shared memory on GPU. The work depicted in Chapter 4 and Chapter 5 followed the same scheme. To provide a more systematic and optimized shared memory layout decision, we developed a framework using ILP methods.

Memory hierarchy optimizations have been crucial for obtaining performance on standard processor architectures. Cache optimizations for improving performance has been one of most widely studied topics in high performance computing, with compile-time transformations [97, 161], runtime optimizations [71], and/or empirical search (auto-tuning) [12] being aggressively used for improving cache reuse. Another topic that has received some attention is careful use of main memory for out-of-core programs [5, 154, 122].

Memory hierarchy is an important consideration in extracting high performance from GPUs as well. Current NVIDIA cards have 16 or 48 KB shared memory per streaming multiprocessor, available to each thread bundle that is executed on this multiprocessor. (At the time this work is being done, we have only cards with 16KB shared memory). For automatically generated GPU code to obtain performance comparable to that of manually tuned codes, a critical challenge is of automatically allocating frequently used variables on the shared memory. This problem cannot be solved by direct application of the existing work on cache transformations, because the shared memory on GPUs is programmer controlled, similar to registers on standard processors. At the same time, the shared memory is larger than registers, and is capable of storing sections of arrays. Thus, direct application of the existing register allocation techniques is not possible either. There have been multiple
recent efforts on compiler-driven management of shared memory on GPUs [20, 110, 121]. Among these, however, one of the efforts is limited to allocation for the execution of a single affine loop nest [20], another is based on very simple heuristics [110], and the last one allows allocation of only a single variable (array) on shared memory at any given time [121].

We formulate the shared memory allocation problem as an integer programming problem [112]. We show how decisions are made for a full function, as opposed to a single loop nest, and consider sections of arrays for allocation on GPUs. Furthermore, we show how the results of integer programming can be used to determine the most profitable loop transformations. Integer linear programming is a special case of linear programming, which requires all the unknown variables be integers. Though it is proven to be NP-hard, existing algorithms can solve many problems in a reasonable time, and therefore, it is widely used in many scheduling and routing applications [1].

Consider any variable (a scalar, an array, or a section of an array) during any point in the execution of a program. Either this variable is currently allocated on shared memory, or it is not. Thus, the shared memory allocation problem can be modeled as a special (but popular) case of integer programming, which is the binary or 0-1 integer programming. While making this decision, our goal (or the objective function) is to minimize the total device memory accesses. This, in turn, depends on the difference between the number of memory accesses that will refer to the shared memory (instead of device memory) and the number of data transfers between device memory and shared memory. To maximize the value of the objective function, we are subject to several constraints. The major constraint is that the size of the variables allocated in shared memory cannot exceed the total size of the shared memory, at any point in the program.

As we stated previously, shared memory allocation problem has many similarities with the register allocation problem, which has been studied extensively by the compiler community. Besides graph coloring, integer linear programming is one of approaches used for register allocation [62]. Among the register allocation algorithms using integer programming, Barik et al.’s work is most closely related to our work. They proposed an algorithm for register allocation in embedded systems [18]. In their work, multiple variables can reside in one register, which means arranging the usage of register is done with a bit-wise
pattern. In our case, we view the shared memory as a single register (though much larger), and formulate a similar problem for shared memory allocation. There are two main differences between our problem and the bit-wise register allocation for embedded systems. First, in our case, not all the variables need to be copied to shared memory. Therefore, in the objective function, instead of calculating the overhead of register spilling, we maximize the utilization of shared memory. Second, in shared memory allocation problem, the variables are arrays, instead of single data elements considered for embedded system. Thus, in our work, it is possible to copy a section of an array in shared memory. This makes our formulation quite distinct.

We have evaluated our approach using several non-scientific applications, including k-means clustering [84], EM clustering [51], PCA [3], and co-clustering [40]. Our approach can outperform a recently published simple heuristic (See [110]) by up to 27%. For two of the applications, PCA and co-clustering, our model also suggested loop transformations that resulted in a factor of 5-6 improvement in performance.

6.1 System Design

This section presents details of our integer programming based framework for shared memory allocation.

6.1.1 Assumptions and Basic Ideas

Our goal is to be able to perform global (intra-procedural) analysis to determine shared memory allocation for a function executed on a GPU. Before discussing about the solution, we make several assumptions.

- We assume that the function has a structured control flow. Thus, while we obviously allow conditionals and loops, goto or exit/break inside loops are not allowed.

- We assume the number of iterations for all the loops have been determined. For many array based codes, such information can be determined by constant propagation, or by using profile information. Similarly, we assume that the frequency with which different branches of a conditional are taken is known. Because of the SIMD nature
of GPUs, this information is important only for the branches that are not inside a parallel loop. Using this information, we can determine the number of times that a basic block is executed.

- We also assume that the size (or estimated maximal size) of each dimension of each array is known. Again, this information can be obtained by constant propagation or using profile information.

In our formulation, allocation of a variable in shared memory is performed only at the beginning of a basic block. This simplifies our formulation, and for most cases, does not lead to sub-optimal results. In the applications we consider, most arrays are accessed with a stride-1 access, which is important for obtaining high memory bandwidth from the device memory. For arrays where this is not true, changing accesses to achieve more stride-1 accesses can also improve performance, but we do not consider this optimization here.

In a function, each variable may have one or more live ranges. A live range is an interval between two definition points of a variable, where a definition point can be the first basic block of the function, the last basic block of the function, the basic block where it is assigned a value, or the basic block where the value becomes dead.

To explain our formulation, we use an example that is shown in Figure 6.1. As stated previously, a variable for allocation on shared memory can be a scalar, a complete array, or a section of an array. Note that scalars can be placed on registers also to achieve faster accesses. While this is handled by our code generation system, we do not make this distinction while presenting our framework.

For any live range of a variable, our goal is finding the basic block, if any, at the start of which the variable is assigned to shared memory. For any live range of a variable, we consider several potential assign points. For example, for the code section in the left-top box in Figure 6.1, the assign points are shown in the right-top box. For simplicity, we require that each variable occupy a fixed portion of shared memory in one live range. This size, however, can vary depending upon the assign point. For example, in Figure 6.1, if we choose assign point 4 for the variable $B$, we will actually allocate only a single element of the array. However, if choose assign points 3, we will allocate one row of the array. If the assign point is 1 or 2, we will be allocating the entire array $B$ on shared memory. Once a
variable is assigned to shared memory, we assume that it stays allocated there till the end of its live range.

Consider a variable that is initially allocated to shared memory in a particular basic block. This allocation could result in one of the following four access features for this live range of the variable: temp, read, write, and read-write. A temp access feature implies that the variable is defined after allocation on shared memory, and becomes dead before the end of the function. Thus, no copying between device and shared memory would be required. A read access feature implies that the shared memory allocation is done after the start of the live range, and therefore, the variable will need to be copied from device memory to shared memory at the assign point. A write access feature implies that the live range is till the end of the function and the variable is updated during its live range. Therefore, at the end of the function, the variable needs to be copied from shared memory to device memory. A read-write access feature implies both a copy from the device memory to shared memory at the assign point, and a copy from shared memory to device memory at the end of the function.

6.1.2 Detailed Integer Programming Model

Any integer programming problem has unknown terms that we are solving for, subject to an objective function and one or more constraints. Consider any live range of a variable $i$ in the program. For any basic block $j$, we could be allocating the variable $i$ on shared memory at the beginning of the basic block. This is captured through the term `assign_point`. `assign_point^i_j = 1` implies that the variable $i$ is initially allocated to shared memory at the start of the basic block $j$. If this is not the case, we simply say `assign_point^i_j = 0`. Thus, the problem we are addressing is modeled as a 0-1 or binary integer programming problem.

We next elaborate on our objective function and constraints. The coefficients for the constraints and the objective functions are the terms listed and described in Table 6.1.

Objective Function

Consider any function that is executed on a GPU. Assuming we do not change the control structure of the program in any fashion, the number of memory accesses is fixed. Our
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVar</td>
<td>Number of variables</td>
</tr>
<tr>
<td>nBB</td>
<td>Number of basic blocks</td>
</tr>
<tr>
<td>nLive&lt;i&gt;</td>
<td>Number of live ranges of the variable i</td>
</tr>
<tr>
<td>live_blocks&lt;i&gt;&lt;k&gt;</td>
<td>Basic blocks that belong to live range k of the variable i</td>
</tr>
<tr>
<td>assign_point&lt;i&gt;&lt;j&gt;</td>
<td>Whether the variable i is initially allocated to shared memory in the basic block j</td>
</tr>
<tr>
<td>Is Assigned&lt;i&gt;&lt;j&gt;</td>
<td>Whether the variable i is allocated in shared memory during execution of basic block j</td>
</tr>
<tr>
<td>iters&lt;i&gt;&lt;j&gt;</td>
<td>Number of executions of basic block j in the function scope</td>
</tr>
<tr>
<td>Data_trans&lt;i&gt;&lt;j&gt;</td>
<td>Transfers between device and shared memory for variable i in basic block j</td>
</tr>
<tr>
<td>size_alloc&lt;i&gt;&lt;j&gt;</td>
<td>Shared memory usage for variable i if assigned to shared memory in basic block j</td>
</tr>
<tr>
<td>Refs&lt;i&gt;&lt;j&gt;</td>
<td>Number of shared memory accesses to a variable i in the basic block j</td>
</tr>
<tr>
<td>Agg.SMref&lt;i&gt;&lt;k&gt;</td>
<td>Aggregated number of shared memory references to the variable i in the live range k</td>
</tr>
<tr>
<td>Total_memcopy&lt;i&gt;&lt;k&gt;</td>
<td>Number of copies between device and shared memory for live range k of the variable i</td>
</tr>
<tr>
<td>live_list&lt;i&gt;&lt;j&gt;</td>
<td>Live variables in the basic block j</td>
</tr>
<tr>
<td>Access&lt;i&gt;&lt;j&gt;</td>
<td>Access feature of the variable i in the live range j</td>
</tr>
</tbody>
</table>

Table 6.1: Description of Terms Used in the Integer Programming Formulation
main goal is to maximize the fraction of the memory accesses that are to shared memory, instead of the device memory. However, in doing so, we have to account for the cost of data movement between device memory and shared memory. 

Thus, our overall objective function can be listed as the difference between the total number of shared memory references ($Agg_{SM ref}$) and the total number of memory copies between device and shared memory ($Total_{memcpy}$). We assign the same weight to both these factors, i.e., we did not consider different weights for the reduced device memory accesses and data loads into shared memory. This was based on experimental evidence, which suggested that the time for accessing shared memory or registers is negligible compared to the cost of each access to the device memory.

Both the terms, $Agg_{SM ref}$ and $Total_{memcpy}$, can be summed over all live ranges of all variables in the function, resulting in the following objective function:

$$Maximize \ z = \sum_{i \in \{1...nVar\}, \ k \in \{1...nLive_i\}} Agg_{SM ref}^i_k - \sum_{i \in \{1...nVar\}, \ k \in \{1...nLive_i\}} Total_{memcpy}^i_k$$

As stated in Table 6.1, $nVar$ is the total number of variables, and $nLive_i$ is number of live ranges for the variable $i$. We next explain how $Agg_{SM ref}$ and $Total_{memcpy}$ are calculated.

The expression for calculating $Total_{memcpy}^i_k$, which is the total device to shared memory and shared memory to device data transfers associated with the live range $k$ of the variable $i$, is

$$Total_{memcpy}^i_k = Data_{trans}^i_j \times iters_j$$

where,

$$j \in \{live\_blocks_k^i, \ and \ (assign\_point_j^i == 1)\}$$

and,

$$Data_{trans}^i_j = \begin{cases} 
2 \times size\_alloc_j^i, \ if \ Access_k^i = \text{read} - \text{write} \\
0, \ if \ Access_k^i = \text{temp} \\
size\_alloc_j^i, \ otherwise \tag{6.1} 
\end{cases}$$
In calculating $Total_{\text{memcpy}}$, we focus on the basic block $j$ where we will determine that the variable is initially allocated in shared memory, i.e. $assign\_point_j = 1$. If there is no such basic block, $Total_{\text{memcpy}}$ is 0. Otherwise, it is a product of amount of data transfer associated with its allocation in the basic block $j$ and the frequency of execution of the basic block $j$. The former is captured through $Data\_trans_j$, which further depends upon the access feature associated with allocation in $j$. If the variable is defined after allocation in shared memory, and becomes dead while in shared memory (temp access feature), the cost is 0. If the access feature is read-write, we multiply the size of the variable with 2. If the access feature is either read or write, the data transfer equals the size of the variable.

Finally, $size\_alloc$ can be calculated by using the indices of the array. For example, in Figure 6.1(b), $size\_alloc$ of $B$ is 1 in basic block 4, $r$ in basic block 3, and $m \times r$ in basic block 1 and 2. There are situations where the index is not an iterator. In this case, we need to estimate the size. Currently, we consider two situations. First, if the index is another array, and we know the value range of that array, the size is the value range. Otherwise, the user provided size will be used.

Next, we focus on calculating the total number of shared memory references resulting from allocation of a particular variable at a basic block. For a particular live range $k$ of the variable $i$, it is calculated as:

$$Agg_{SM\_ref}^i_k = \Sigma_{j \in \{\text{live\_blocks}\_i^k\}} \text{Is\_assigned}_j^i \times Refs_j^i \times iters_j$$

We consider all basic blocks that are within this particular live range of the variable. Depending upon whether this variable is assigned to shared memory, and where it is assigned to shared memory (captured through $assign\_point$), this variable will be in shared memory for none, some, or all of these basic blocks. For basic block $j$, this is indicated through $Is\_assigned_j^i$, i.e., a value of 1 indicates that the variable is in shared memory during its execution, and a value of 0 indicates that it is not. The expression above simply takes the product of number of references to (shared) memory in each execution of a basic block and the number of iterations of the basic block, over the basic blocks where $Is\_assigned_j^i$ is 1. Note that $Is\_assigned_j^i$ is further related to the $assign\_point_k^i$ values for all basic blocks $k$ that dominate the basic block $j$. 

71
Constraints
While solving for the values of assign_point to maximize the above metric, we have two important constraints. The first, and the obvious, constraint is that at any point in the program, the shared memory allocation cannot exceed the available shared memory.

Formally, for each basic block $j$, we have the following constraint:

$$\sum_{i\in live_list_j} Is\_assigned^i_j \times size\_alloc^i_j \leq limit$$

Here, live_list$_j$ is the list of live variables in the basic block $j$. As we explained earlier, Is\_assigned$_j^i$ captures whether the variable $i$ is allocated in the shared memory during the execution of the basic block $j$. Finally, limit is the size of the shared memory.

In addition, we have another constraint, which is that within each live range, a variable can be allocated to shared memory only once. Formally,

$$\sum_{j\in \{live\_blocks_i\}} assign\_point^i_j \leq 1$$

In our formulation, we have so far considered the start of every basic block a potential assign point for each variable. In practice, this can be unnecessarily expensive. Thus, in our implementation, we carefully select only realistic assign points for each variable. The algorithm we use for this purpose is shown in Figure 6.2. Simply put, if a variable $i$ is accessed in the basic block $j$, only $j$ and the outer loops of $j$ will be considered as assign points.

Obtaining Parameters
We obtain the terms used in our integer programming formulation by code analysis implemented with LLVM [99]. First, we extract the access feature of each variable while an intermediate representation is being generated by the LLVM compiler. Then, in a second pass, we construct a control flow graph, focusing on the loop nesting relationship. In the same pass, the live ranges of each variable are obtained, along with information such as the basic blocks where each variable is accessed, how many different elements are accessed, which basic block is in the live blocks of a certain live range, and the iterators of each loop. Then, with the obtained parameters, we can calculate the other terms used in the integer programming problem, by traversing the control flow graph. We used the integer programming solver from the GLPK package [115] to obtain the final results.
Revisiting the Example

To illustrate how the integer programming solver determines shared memory allocation, we revisit the code segment shown earlier in Figure 6.1 (a).

Two sets of parameter values are used to illustrate our algorithm. In Figure 6.1 (c), \( n=2048 \), \( m = 4 \) and \( r=4 \). Therefore, \( A \) is a matrix of size \( 2048 \times 4 \), \( B \) is a \( 4 \times 4 \) matrix, and \( C \) has 4 elements. 256 threads are used in one block (and only 1 block is used in this sample), and we assume that the size of shared memory is 8192 bytes. All the data elements are 4-byte float numbers.

With the above parameter values, the integer programming solver will come up with the following values: \( \text{assign-point}_0^1 \) and \( \text{assign-point}_0^2 \) are 1, and all the other elements of \( \text{assign-point} \) are 0. This result indicates that \( B \) and \( C \) will be assigned to shared memory in the basic block 0, which is the beginning of the function. Therefore, \( B \) will be copied to shared memory at the beginning of shared memory, and writes on \( C \) will be on shared memory. And the values of \( C \) from different threads are combined and written into device memory at the end of the function.

In another case, as shown in Figure 6.1 (d), if \( m=3 \) and \( r=3 \), the output of the integer programming solver will be as follows. \( \text{assign-point}_0^1, \text{assign-point}_0^1, \) and \( \text{assign-point}_0^2 \) are all 1. This result show that \( A \) can also reside in shared memory, but the assign point is at the basic block 1, which implies that \( A[i][0] \ldots A[i][r-1] \) will be copied to the shared memory at the beginning of every \( i \) loop.

6.1.3 Code Generation

To generate code for data movement between device memory and shared memory, we need another vector, which indicates the range of elements for a variable that need to be kept in shared memory. This vector can be obtained while extracting the parameter \( \text{size.alloc} \) for each variable at each basic block. After the \( \text{assign-point} \) for the variables are determined, the corresponding code is generated. The overall method is shown in Figure 6.3. \( \text{shared} \) is a parameter derived from the results of the integer programming solver, indicating whether a variable uses the shared memory in a certain live range. Lines 4 and 5 address the possibility that a certain variable is eliminated from shared memory in a basic block, and the space is
likely assigned to other variable. One possible method for avoiding fragmentation will be that all the variables located after the evicted variable should be moved forward. It turns out that the overhead of such data movement within the device memory is extremely small. Thus, our implementation performs such data relocation to avoid fragmentation.

### 6.1.4 Improving Shared Memory Use with Loop Transformations

Loop transformations have been widely used for improving reuse on memory hierarchies [161]. Loop transformations for enabling better use of shared memory on GPUs is a topic that will likely be studied extensively in the future. While a detailed study on this topic is well beyond the scope of this thesis, we briefly show our integer programming framework can be used to find useful transformations, and to further validate the benefit of applying those transformations.

The transformation we focus on is *loop permutation*, which switches the ordering of the loop within a single loop nest. The goal behind performing loop permutation is to enable more data reuse [161] and techniques for determining the correctness of this transformation are well known. We show this transformation with an example, arising from a data mining application, co-clustering [40]. The original code is shown in Figure 6.4 and the transformed code is in Figure 6.5. The transformation shown enables \( data[r][c] \) and \( colCL[c] \) to be copied to shared memory at the Line 7, resulting in a better use of shared memory. The transformed code, however, requires an *array expansion* for \( tempDis \). If the expanded array cannot reside in shared memory, the transformation could potentially not be useful.

Based on the terms used in our integer programming formulation, we use the following method for determining the profitability of this transformation. Suppose basic blocks \( j_1 \) and \( j_2 \) are two potential *assign points* for one live range of the variable \( i \). Let the size that \( i \) occupies in shared memory be \( limit.coe[i][j_1] \) and \( limit.coe[i][j_2] \), respectively, for the two assign points. The total overhead of memory copy associated with these two assign points is similarly \( overhead.coe[i][j_1] \) and \( overhead.coe[i][j_2] \). If \( limit.coe \) are the same, but \( overhead.coe \) are different, then the loop permutation to achieve lower \( overhead.coe \) is likely profitable. For example, in Figure 6.4, the same \( data[r][c] \) is loaded \( rc \) times. Intuitively, the *for* loop of Line 5 and the *for* loop of Line 8 can be switched, so \( data[r][c] \)
only need to be loaded once, which means it can be allocated in shared memory (or replaced by a scalar). Given the size of the arrays, the code analyzer can get the value of limit.coe and overhead.coe for data in each basic block. Obviously, limit.coe has the same value in the for loops on Line 5 and Line 8, while overhead.coe[i][j1] and overhead.coe[i][j2] are different.

We now show how the loop transformations suggested by our framework help improve the performance. For two of the applications, k-means, and EM, our limited framework (which suggests only loop permutations) did not suggest any transformations. Thus, we show results from PCA and co-clustering, which are shown in Figures 6.6 and 6.7. We can see that loop transformations can play a critical role in enabling better usage of shared memory. For PCA, we got a speedup of about 6 over the original Int-solved version. For co-clustering, loop transformation yielded a speedup of about 5 over the original version. These experiments show that the transformations suggested by our framework can have even more significant benefits than simply allocating shared memory.

6.2 Experiments

In this section, we present the results from the experiments conducted to evaluate our approach, i.e, we show the performance with shared memory layout determined by our approach. We compare it with an approach where shared memory is not used, and with another recently published heuristic approach [110]. The heuristic we compare against is the greedy sorting approach. Here, variables are sorted according to their sizes, and shared memory is allocation in increasing order of the size. This is a simple approach requiring minimal analysis from the compiler, and does not consider the frequency of accesses to a particular variable.

We conducted our experiments using four non-scientific applications. They are, k-means clustering [84], EM clustering [51], PCA [3], and co-clustering [40]. There were several reasons for choosing this set of applications. First, three of these applications were also used for evaluating the greedy sorting scheme we are comparing against. Second, these applications have irregular or data-dependent access patterns. Therefore, methods restricted to affine loops (e.g. [20]) are not applicable to these applications. Besides the experiments
listed above, we also compared the performance of our k-means implementation with a GPU implementation available from Northwestern University [95].

Our experiments were conducted using a Dell Dimension 9200 PC, equipped with Intel\textsuperscript{(tm)} Core\textsuperscript{TM} 2 E6420 Duo Processor with 2.13 GHz clock rate, with a 768MB NVIDIA GeForce 8800 GTX, with 16 multiprocessors and 16KB shared memory on each multiprocessor.

The integer programming solver we used is from the GLPK package [115]. For the functions we parallelized using GPU, the key terms in our formulation and the execution time for the integer programming solver is listed in Table 6.2. In the case of PCA, only covariance computation is parallelized, as other functions are not compute-intensive. Overall, we can see that for modest-sized functions, the running time for the integer programming solver is very small. Therefore, this formulation is suitable for use in an interactive automatic code generation (or optimization) framework.

First, we show how we improve the performance of the four applications by allocating shared memory through our integer programming based approach. We have compared up to three versions for each application. no shared memory is a version that does not use
shared memory, i.e., all memory references are to the device memory. basic is the code based on the simple and intuitive shared memory arrangement, the greedy sorting[110]. Finally, Int-solved is the version with the shared memory allocation suggested by the integer programming based approach.

**K-means Clustering:** The basic k-means [84] algorithm is as follows. Inputs are points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are involved: 1) start with k given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the k centroids from the points assigned to the corresponding center, and 4) repeat this process until the assignment of points to cluster does not change.

Figure 6.8 shows the running time with three shared memory management schemes. We used a data set of 384 MB for the results reported here, though the relative performance of the three schemes does not depend upon the dataset size. We executed the three versions with different number of thread blocks and number of threads in each such block. With the best configuration, basic version has a speedup of 10 over the version without using share memory, and the Int-solved version is about 16% faster than the basic version. The reason that the Int-solved version performs better is because we consider array sections for allocation.

We also compared the performance of our best version with another implementation for the same application that is publicly available. This implementation is from Northwestern University [95], and is based on a sophisticated management of texture memory, specific to this application. The results are shown in Figure 6.9. This experiment was done using a smaller data size (98MB), since the texture memory version did not work properly for larger data sizes. As we can see, our version, with an automatic scheme for shared memory allocation, clearly outperforms the version which is based on texture memory. Another significant difference in this implementation is that more computations are performed on the CPU. As a result, even with a carefully optimized use of shared memory, they are only slightly faster than our implementation without the use of shared memory.

**EM Clustering:**
For EM clustering, we compared the \texttt{Int-solved} version with the \texttt{basic} version, and show the performance in Figure 6.10. The dataset size was 12 MB. We did not include the \texttt{no shared memory} version, since it was very slow, and even was unstable. It can be seen that the \texttt{Int-solved} approach is 27\% percent faster than the \texttt{basic} version. Similar to what we observed with k-means, enabling allocation of array sections turn out to be crucial.

**Principal Component Analysis:**

With PCA, we experimented with an input matrix of 256K rows, with 80 columns. The results are shown in Figure 6.11. One interesting observation for this application is that the \texttt{basic} scheme could not find any variables to allocate in shared memory. Thus, the \texttt{basic} version and the the \texttt{no shared memory} versions are identical. Therefore, we only show two versions in Figure 6.11, and the \texttt{no shared memory} version is not explicitly shown.

Using the integer programming solver, we were able to get a 14\% improvement over the \texttt{basic} version. Because of the nature of data accesses in this application, shared memory use is limited. As a result, the overall gains from the use of shared memory are small, though our approach can lead to some improvements, unlike the heuristic method. For this application, a much more significant advantage came from a loop transformation suggested by our framework. This is shown in the next subsection.

**Co-clustering:**

Co-clustering is another widely used data mining application. It is similar to k-means clustering, but conducts reduction on multiple dimensions. Our implementation considered two dimensions, i.e. it had row clustering and column clustering phases. We developed a shared memory implementation based on a sequential implementation available from University of Texas [40]. Figure 6.12 shows the performance of different shared memory schemes.

The experiments were done with a matrix of $4096 \times 4096$ float point numbers. The integer programming solver got the same shared memory usage as the intuitive approach did, so we just show the \texttt{Int-solved} version, omitting the \texttt{basic}. For this application, \texttt{Int-solved} is 85\% faster than the version without shared memory usage, under the best configuration. In Figure 6.12, we only include tests with up to 128 threads per block, which
turn out to have the best performance. Similar to PCA, the main advantage of our approach is in the loop transformation it could suggest, which is shown in the next subsection.

6.3 Summary

Effectively exploiting memory hierarchy has been a critical step in performance tuning applications on standard processors. Memory hierarchy is an important factor on GPUs also. We built an integer programming framework for allocating shared memory on GPUs. Three key aspects of this framework are: 1) it can be applied to a full function with structured control flow, i.e. it is not restricted to a single loop nest, 2) it can consider allocation of array sections, and 3) it can suggest beneficial loop transformations.

We have evaluated the effectiveness of our approach using four non-scientific applications with data-dependent or irregular data accesses, where approaches based on affine data accesses are not applicable. We show a performance improvement of up to 27% over an intuitive approach, with our ability to allocate sections of arrays being the main factor behind the improvements. For two of the four applications, the loop permutation suggested by our framework resulted in a 5-6 fold improvement in performance.

In the future, we will like to extend our work by considering additional loop transformations. We will also like to explore runtime preprocessing for shared memory allocation for applications with accesses through indirect arrays. Another issue will be considering the use of texture and constant memory for read-only variables. As GPU architectures continue to evolve, new opportunities and challenges continue to arise. For example, in the upcoming Fermi GPU by NVIDIA, on-chip memory can be split into shared memory and L1 cache with different configurations [127]. Our method can be extended to decide the configuration suitable for a particular application, besides the allocation of shared memory.
Figure 6.1: A Running Example to Show Shared Memory Allocation Using Our Framework
1. for each live variable $i$
2. for each basic block $j$
3. if ($i$ is accessed in $j$ or any block nested in $j$)
4. insert $j$ into to_assign[$i$]
5. for each loop nest
6. $MAX\_LEVEL$ = highest nesting level from this loop nest in to_assign[$i$]
7. for each basic block $j$ in to_assign[$i$]
8. if $j \geq MAX\_LEVEL$
9. remove $j$ from to_assign[$i$]

Figure 6.2: Algorithm of Finding Potential Assign Points

---

1. for each basic block $j$
2. for each live variable $i$
3. if ($j$ starts a new live range $k$ of $i$ \\&\& \( shared[i][k] == 0 \) \\&\& \( shared[i][k-1] == 1 \))
4. move the variables after $j$ to overlap the space of $j$
5. Adjust shared memory offset
6. if (assign_point[$i$][$j$])
7. Generate code for memory operation on $i$ according to the adjusted shared memory offset

Figure 6.3: Method for Code Generation
1. for (int r = 0; r < numRow; r++) {
2.     minCL = 0;
3.     minDistance = MAX;
4.     for (int rc = 0; rc < numRowCluster; rc++) {
5.         if (rowCS[rc] > 0) {
6.             tempDis = 0;
7.             for (int c = 0; c < numCol; c++) {
8.                 for (int rc = 0; rc < numRowCluster; rc++) {
9.                     tempDis += data[r][c] * Acomp[rc][colCL[c]];
10.                 }
11.             }
12.             tempDis = -2*tempDis + rowQual[rc];
13.             if (tempDis < minDistance) {
14.                 minDistance = tempDis;
15.                 minCL = rc;
16.             }
17.         }
18.     }
19.     rowCL[r] = minCL;
20. }

Figure 6.4: A Code Segment of Co-clustering

1. for (int r = 0; r < numRow; r++) {
2.     minCL = 0;
3.     minDistance = MAX;
4.     for (int rc = 0; rc < numRowCluster; rc++) {
5.         tempDis[rc] = 0;
6.         for (int c = 0; c < numCol; c++) {
8.             if (rowCS[rc] > 0) {
9.                 tempDis[rc] += data[r][c] * Acomp[rc][colCL[c]];
10.             }
11.         }
12.         if (tempDis[rc] < minDistance) {
13.             minDistance = tempDis[rc];
14.             minCL = rc;
15.         }
16.     }
17.     rowCL[r] = minCL;
18. }

Figure 6.5: Transformed Code for Figure 6.4
Figure 6.6: Performance of PCA with Loop Transformation

Figure 6.7: Performance of Co-clustering with Loop Transformation

Figure 6.8: K-means: Impact of Shared Memory Allocation Scheme

Figure 6.9: K-means: Performance Comparison with an Implementation Using Texture Memory

83
Figure 6.10: Performance of EM with Different Shared Memory Allocation Schemes

Figure 6.11: Performance of PCA with Different Shared Memory Usage

Figure 6.12: Performance of Co-clustering with Different Shared Memory Usage
Besides data mining applications, another class of applications on GPU that we investigated are tensor contraction functions, which are widely used in many computation intensive tasks. For this kind of application, significant performance gain has been observed on GPUs [114].

In late 2010, we did an evaluation for the latest GPU series, the Fermi GPU, as well as an older device, the T10 GPU, for tensor contraction functions, in the context of NWChem [94]. By testing different algorithmic aspects, we found that the two cards require quite different code patterns.

In comparing the architectural specifications of NVIDIA 10-series Tesla cards to 20-series (Fermi) cards, we can see several challenges in application development and performance tuning. While there is a large increase in the peak double precision performance, the memory bandwidth only increased by 42% (from 102 GB/sec to 144 GB/sec), and the memory frequency increased by less than a factor of 2 (from 800 MHz to 1.5 GHz). The memory bus width has been reduced from 512-bits to 384-bits. Unlike the previous generation, the DMA engine in Fermi supports concurrent bi-directional transfer, allowing better data movement overlap. In addition, the memory hierarchy of Fermi is significantly different due to the introduction of an on-chip Level 2 cache.

We used tensor contraction expressions to evaluate the algorithms on the two devices, which arise frequently in computational chemistry. A tensor contraction is a generalized matrix multiplication. It is the fundamental operation in coupled cluster (CC) methods [132],
a collection of ab initio electronic structure calculation theories. In achieving high performance on Fermi cards (and cluster of Fermi cards) for tensor contraction functions, we addressed the following challenges:

**Reducing data transfers over PCIe:** On Fermi, better numerical performance resulted in the data movement between the host memory and the device memory becoming the bottleneck. Therefore, we modified the algorithm to keep intermediate result matrices in the GPU memory.

**Exploiting larger shared memory and register file:** Based on the larger, configurable shared memory and a larger register file in Fermi, we applied register tiling [124].

**Optimizing coalesced access:** With modifications like register tiling and the availability of L1 and L2 cache, accesses to the output matrix, and not the input matrices, constitute the dominant fraction of device memory access time. Therefore, we modified the index calculation to enable coalesced accesses to the output matrix.

**Optimizing conditionals for better performance:** The optimizations we employ to improve thread-block utilization requires the write back to device memory being enclosed in a conditional evaluation. Register tiling results in multiple elements of the output array being evaluated by each thread. The conditionals evaluated for these elements can be coalesced, further improving performance. We have restructured this computation, to perform reversed checking, and thus reduce the number of branches.

We have carried out a detailed evaluation study of various optimizations on Tesla 10-series and Fermi GPUs, and also clusters with these GPUs. We have used 9 different tensor contraction expressions and 4 distinct problem sizes. The first three optimizations we have listed above all result in very substantial performance improvements, ranging between 20% to a factor of 4, depending upon the tensor contraction. The last optimization yielded additional improvements between 5% and 18%. After these optimizations, the speedup on Fermi cards with respect to the Tesla 10-series cards ranged between 3 and 7, depending upon the expression and the problem size, with an average of 4.78. This corresponded to 72% of the improvement factor (6.6) in the peak double precision performance for Fermi.
Table 7.1: Architectural comparison of Tesla T10 and T20 series GPUs

<table>
<thead>
<tr>
<th></th>
<th>Tesla T10</th>
<th>Tesla T20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. multiprocessors (SM)</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>Num. cores per SM</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>SM clock frequency</td>
<td>1.3GHz</td>
<td>1.15GHz</td>
</tr>
<tr>
<td>Single Precision peak GFLOPS</td>
<td>933</td>
<td>1030</td>
</tr>
<tr>
<td>Double Precision peak GFLOPS</td>
<td>78</td>
<td>515</td>
</tr>
<tr>
<td>Memory Frequency</td>
<td>800 MHz</td>
<td>1.5GHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>102 GB/sec</td>
<td>144 GB/sec</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>512 bit</td>
<td>384 bit</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>16KB</td>
<td>16KB/48KB</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>–</td>
<td>48KB/16KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>–</td>
<td>768KB</td>
</tr>
</tbody>
</table>

7.1 Background

In this section, we briefly introduce the two most recent series of GPUs from NVIDIA, which are the T10 (Tesla) and T20 (Fermi) cards.

The detailed specifications of T10 and T20 series cards are compared in Table 7.1. One Tesla T10 GPU has 30 multiprocessors, each of which has 8 cores. The cores run at 1.44GHz peak clock, and each card has 4GB memory, with 792 MHz GPU memory clock. Every multiprocessor has 16KB shared memory, and a register file of 16KB.

Tesla 2050, known as “Fermi”, is the latest series of graphic card products by NVIDIA. The architecture of Fermi features 14 multiprocessors (possibly 16 in the near future), each of which has 32 cores. The clock rate is 1.215GHz, and memory clock is 3348MT/s. The biggest difference, however, is in support for double precision operations. The peak double precision performance has improved by a factor of 6.6. For scientific applications that are
dominated by double precision operations, potentially resulting in much higher performance. However, the memory bandwidth and clock speed for device memory accesses, and the data transfer rates between the host and the device, have improved by a much smaller factor. Thus, we can expect that accesses to device memory and data transfers from host memory can be even more of a bottleneck with Fermi cards.

Another key difference in Fermi is in the memory hierarchy. Fermi has a much larger shared memory/L1 cache, which can be configured as 48KB shared memory and 16KB L1 cache, or 48KB L1 cache and 16KB shared memory. The register file is also much larger, i.e. 32KB. Unlike previous cards, an L2 cache is also available.

In view of the architectural characteristics of the new Fermi cards, we can see several challenges in scaling performance at a ratio close to the improvement in the peak double precision performance. For the specific application we focused on, i.e. tensor contractions, the most pertinent issues were as follows:

1. How do we modify the algorithm in view of a factor of 6.6 improvement in peak double precision performance, but with unchanged PCI-e bandwidth and the possibility of concurrent bidirectional PCI-e transfers?

2. How do we exploit the larger shared memory and the presence of caches on every multi-processor?

3. How do we exploit larger number of available registers on each multi-processor?

7.2 System Design

7.2.1 Tensor Contraction and Initial GPU Implementation

Our focus is on the tensor contractions that occur in the context of the Coupled Cluster (CC) theory [83, 132, 19], which is a widely used method for solving the electronic Schrödinger equation. The methods such as the CCSD(T) approach [139] enable accurate predictions for the molecular structure, inter-molecular interactions, transition states, and activation barriers. These computations, however, also have a high computational complexity \(O(N^7)\), and thus require the use of parallel machines and/or accelerators.
The primary data structures in the coupled cluster theory are the tensors, which are multi-dimensional matrices. Tensor contractions translate into generalized matrix-matrix multiplication operations. The tensors, which exhibit sparsity and symmetry, are organized as a collection of dense blocks, each of which can be computed upon in parallel. A parallel tensor contraction is thus translated into a collection of block-block contractions. Henceforth, we shall use the notation corresponding to tensor contractions to refer to both the complete contraction involving the input tensors and the contraction between individual blocks of the corresponding tensors. The reference shall be clear from the context.

Consider the following tensor contraction.

\[ R[h_1,h_2,h_3,p_1,p_2,p_3] = T[p_4,p_2,h_3,h_2] \ast V[p_4,h_1,p_1,p_2] \]

(7.1)

The tensor contraction involves computing tensor \( R \) from tensors \( T \) and \( V \). The indices \( h_1, h_2, \ldots \) correspond to the unoccupied orbitals, while \( p_1, p_2, \ldots \) correspond to the occupied orbitals. The index \( p_4 \) is common to both input tensors, corresponding to the common dimension in matrix-matrix multiplication, and is referred to as the contracted index. Unlike conventional matrix multiplication, several dimensions can be contracted in a binary tensor contraction. Since a \( h(\text{ or } p) \) index in a tensor could potentially contract with a \( h(\text{ or } p) \) index in any other tensor, all \( h(\text{or } p) \) indices are tiled using the same block sizes. The size along a dimension of a block is the minimum of this tile size and the size limit imposed on it by the sparsity constraints. The high dimensionality of tensors together with the sparsity results in the sizes of each dimension of a block being small. The generality in the number of contracted indices results in the block-block tensor contractions approximating highly rectangular matrix-matrix multiplication operations, exhibiting various “shapes”, from outer products to inner products.

We now describe our earlier work on developing parallel implementations on earlier GPU cards [114]. The arguments to the kernel include the dimensions and the pointers to the buffers in GPU memory. Since the host buffer is passed as a linear array, strides to access the different dimensions need to be computed. It is expensive to compute these strides in the kernel executed in the GPU. Hence, the strides are computed in the host and passed as arguments to the kernel.

Our approach to computation within a thread-block is analogous to the approach used
in mapping matrix-matrix multiplication to GPUs. The output indices are mapped to the thread grid on the GPU. Therefore, each thread will compute one output element. The innermost indices of each input matrix are mapped to the thread ID on two dimensions. Therefore, every thread in a block will calculate the output elements with the same indices except for the two from inner most indices of the input matrices.

In CUDA, all thread blocks and threads share the same values of the kernel function arguments. The differentiation between the work to be performed by the different threads and thread-blocks needs to be derived from their location in the thread block and thread grid. This is done by modulo and division operations. In order to minimize the number of such operations, the encoding of the tensor dimensions is split between the number of grid dimensions available. In addition to the contracted index, we choose one dimension each from the two tensors to form a matrix-multiplication operation at the thread-block level. The threads in each thread block cooperate in moving data between the GPU device memory and shared memory. The code snippet for the kernel, is shown in Figure 7.1.

**Restructuring the Computation to Improve Performance**

We now describe two restructuring optimizations that were used to improve performance. They are, *index combining* and *dimension flattening*.

In some tensor contractions, the index order allows further optimizations. When a pair of indices occur in the same order in every occurrence, they can be replaced by one index whose size is their product. This optimization does not incur any runtime overhead of data transformations. It is performed before further steps to increase the size of the dimensions, so as to improve thread-block utilization where possible, and reduce index computation overhead in the kernel. In equation 7.1, for example, the index sequence \{p1,p2\} can be combined into \{x\}, where \(|x| = |p1| \times |p2|\).

One of the primary disadvantages of the baseline approach is that only one dimension is tiled for each input array. This works well when the tiled dimensions are large or match the thread block configuration. With small dimension sizes, the loop trip counts could be small than the thread block size employed, impairing thread-block utilization. We utilized an optimization we refer to as *dimension flattening*. In the baseline scheme all the threads in a
void kernelCode(devR, devT, devV, ad, ..., ld, Rld_\text{k}, \ldots, Rld_{\text{a}}, Tld_{\text{b}}, \ldots, Tld_{\text{l}}, Vld_{\text{j}}, \ldots, Vld_{\text{l}})
{
  \_\_\text{shared}\_\_ \text{double} \text{shmT}[16][16], \text{shmV}[16][16];
  \text{int rest1}=\text{blockIdx}.x, \text{rest2}=\text{blockIdx}.y;
  \text{double tlocal}=0;
  \text{const int tidx=threadIdx}.x, \text{tidy}=\text{threadIdx}.y, \text{Tl};
  \{\text{* determine indices to compute *}\}
  \text{c}=\text{rest1}/\text{cd}; \text{rest1}=\text{rest1}/\text{cd};
  \text{b}=\text{rest1}/\text{bd}; \text{rest1}=\text{rest1}/\text{bd};
  \text{kT}=\text{rest1}*16; \text{j}=\text{rest2}/\text{jd}; \text{rest2}=\text{rest2}/\text{jd};
  \text{i}=\text{rest2}/\text{id}; \text{rest2}=\text{rest2}/\text{id}; \text{aT}=\text{rest2}*16;
  \text{k} = \text{kT}+\text{tidy};
  \text{a} = \text{aT}+\text{tidx};
  \text{for(\text{int lT}=0; lT<\text{ld}; lT+=16) \{\}
    \text{shmT} \leftarrow \text{devT}[\text{offsetT}(lT, k, c, b)];
    \text{shmV} \leftarrow \text{devV}[\text{offsetV}(lT, a, i, j)];
    \_\_\text{syncthreads}();
    \{\text{* actual computations *}\}
    \text{if (k<kd && a<ad)}
      \text{tlocal} += \text{shmT}[\text{tidy},:] * \text{shmV}[:,tidx];
    \_\_\text{syncthreads}();
  \}\n  \text{if (k<kd && a<ad)}
    \text{devR}[\text{offsetR}(a, b, c, i, j, k)] += \text{tlocal};
}\n
\text{Figure 7.1: GPU kernel function for Equation 7.1}\n
thread block contribute to output array elements which differ in only two indices, one from each input array. In the dimension-flattened version, the indices in the output array are grouped into those from each of the input arrays. Each of these groups is further flattened into a single-linear dimension, and tiled according to the thread block dimensions. The flattening could result in different threads in one thread block operating on output array elements that differ in more than two indices. This is illustrated in Figure 7.2. Some of the thread blocks operate on distinct \textit{planes} of the multi-dimensional array. Only the last thread block along each flattened dimension is under-utilized.
Other Optimizations

Two other optimizations were performed on our implementation on Tesla 10-series cards, in view of the architectural features.

**Pipelining:** In the GPUs we use, it is possible to launch the kernels in such a way that kernel execution and data movement over PCI Express are overlapped. We utilize this feature by implementing a pipelined implementation using *streaming*, which turned out to be very effective in improving the performance.

To keep data movement contiguous and efficient, we select the outer-most index of the output as the *streaming index*. The number of kernel invocations is the same as the value of streaming index. Because of the nature of the contractions that we deal with, the input matrices are much smaller than the output matrix. So, we copy the input synchronously before any kernel launches. For the output, to avoid copying it into device memory before the kernel execution, we use CPU to do the accumulation. Therefore the entire execution is done in a pipeline involving three stages: *kernel execution, data copy* between device and host memory, and *CPU accumulation*.

**Optimized Memory Allocation:** When allocating GPU memory, we used an approach for reducing the overhead of allocating new memory blocks multiple times. Once a chunk of memory is allocated, it is kept until the end of the execution, even if it is freed while the program is running. When a new request for GPU memory is issued, the previously freed memory is used, as long as its size is large enough.

### 7.2.2 Algorithmic Modifications and Performance Tuning for Fermi

As we described earlier, while the Fermi or T20-series architecture has nearly a factor of 7 higher peak double precision performance, there are several challenges in scaling performance of an application. We now describe the algorithmic modifications and optimizations we performed on our tensor contraction implementation. Particularly, we report on four different modifications or optimizations. First, we reduce the frequency of data transfers between the host memory and the device memory by keeping the intermediate data across tasks in GPU device memory. Second, we perform register tiling and exploit the larger shared memory of Fermi and its larger register file. Third, we modified the way that indices
of input and output indices are computed so that we have better access patterns to the device memory. Fourth, we used reversed condition checking to accelerate accesses to the output matrix in the device memory by minimizing the number of branches encountered in the common case.

**Removing Data Transfer over PCI Express**

On the Tesla T10 cards, our experiments with single tensor contraction functions had shown that data copy over PCIe (PCI Express) takes a similar amount of time as the execution time for the kernel function. Therefore, the pipelining mechanism described in Section 7.2.1 is very effective. However, as we had stated earlier, the ratio of peak double precision performance and data transfer bandwidths are very different in Fermi cards. Thus, instead of just overlapping data movement and computations, we focused on reducing the total data transfer volume.

The CCSD(T) part of NWChem works in the following way. The entire problem space is tiled into tasks, each process grabs a task and processes it until all the tasks are finished. One task has three major functions, as shown in Figure 7.3. Two matrices are created as intermediate results $T_3.s$ and $T_3.d$. Those two matrices have the same index range on every dimension. The final output of this task is two double precision numbers, which are the energy values. Based on this pattern, we can keep the intermediate data on GPU, and only copy the final result back to main memory. By doing this, we can improve the performance significantly, especially when the execution is dominated by data movement between host memory and device memory.

Therefore, after our transformation, the code works as follows. In each iteration (one task), one block (one tile on each dimension) of the two matrices $T_3.s$ and $T_3.d$ are allocated at the beginning of the iteration. Then, the particular block of $T_3.s$ is computed by `ccsd_t_single`. After that, the block of $T_3.d$ with the same indices is accumulated by `ccsd_t_doubles_1` and `ccsd_t_doubles_2`. Finally, the results are used in another kernel which computes the final result, which comprises two double precision energy values. Each thread block keeps one copy of the two values in the device memory, which are copied back to host memory. By using this strategy, we only have to copy the scalars from each thread.
block between the device and the host memory. Once all the scalars belonging to each thread block are computed and copied back in host memory, the CPU performs the final accumulation.

**Using Register Tiling**

The main idea of using register tiling on Fermi card is making good use of the larger shared memory and the larger register file. Studies and analysis with other applications have shown that available shared memory and register file size was inadequate for performing this optimization on earlier cards [124].

We modify our algorithm in the following way. In the host function that invokes the kernel, the thread grid is configured in a similar way as our baseline algorithm, except that the size of each dimension of the grid is reduced by a factor of 4. For example, if the grid size is (256, 256) in the original algorithm, the new algorithm will result in a grid configuration of (64, 64). Blocks are configured in the same way, which is (16, 16) in our implementation.

In the kernel, each thread computes 16 output elements, as shown in Figure 7.4. Here, \( \text{shm}_t \text{v2} \) and \( \text{shm}_v \text{v2} \) are the buffers in shared memory to store the input data. In each iteration (one tile of one row of \( T_2 \) and one tile of one column of \( V_2 \)), 4 rows of the input \( T_2 \) and 4 columns of the input \( V_2 \) need to be loaded to the shared memory for the computing task of each thread, which means 64 rows of input and 64 columns of output in total. This results in the calculation of 4 index sets on the input \( T_2 \) and 4 index sets on the input \( V_2 \). In each thread, 4 elements in each corresponding row of \( T_2 \) and 4 elements in each corresponding column of \( V_2 \) are loaded from the shared memory to registers. To ensure coalesced access, each thread accesses the 4 elements with a stride of 16. Then, 16 multiplications are performed on the 4 input values from \( T_2 \) and 4 input values from \( V_2 \). The resulting 16 products are accumulated to 16 double precision registers. After all elements in one column of \( T_2 \) and one row of \( V_2 \) have been processed, the results are accumulated to the 16 elements in the output matrix in the device memory.

**Modified Index Calculation Order**

In the new algorithm, the overhead of loading the input matrix is reduced significantly, because of the use of larger shared memory and more aggressive usage of registers, as we
described in the previous subsection. Subsequently, accumulations to the output matrix are now the dominant device memory accesses. To reduce their overhead, we change the way that the indices are calculated. Our goal is that adjacent threads should calculate adjacent set of output elements, and thus, accesses to the output element are coalesced. This, in turn, reduces the overhead of device memory accesses.

Figure 7.5 shows the original index calculation in a function that performs tensor contraction, whereas Figure 7.6 shows the modified order. Here, $k$ is combined with $c$, and $j$ serves as the stream index, so, they are not calculated inside the kernel. In Figure 7.5, the inner-most index of the input matrices are decoded first, which means consecutive threads will access adjacent elements in the input, regardless of the index order of the output. This strategy works well with the original algorithm where there is no register tiling, because access to the input matrices dominates the kernel execution. On the other hand, with the new algorithm on Fermi cards, because each thread in one block writes 16 output elements instead of 1 element, the accesses to the output matrix become the main device memory access overhead. Hence, in Figure 7.6, the inner most index of output is decoded from the thread ID (in the entire grid) first, resulting in consecutive output elements being accessed by adjacent threads.

**Reversed Condition Checking**

In the new algorithm with register tiling, since every thread computes 16 elements instead of 1 element, boundary checking has to be addressed when each element is being written to device memory. This checking leads to a large number of branches. In our first implementation, there are 20 conditional branches in the code which stores results to the device memory. To reduce the overhead of branches, we used a reversed order of boundary checking, which are shown in Figures 7.7 and 7.8. $t_3$ is the output matrix in device memory, and $\text{thread}_y$ is the thread index on $y$ dimension. $\text{offset}_1$ to $\text{offset}_4$ are the offsets of the 4 output elements that a thread will update which have the same offset on $x$ dimension. This section of code performs the checking of boundary on the $y$-dimension. The checking on the $x$-dimension happens outside of this checking, and therefore, is repeated 4 times. In Figure 7.7, all the condition checking has to be performed by every thread, resulting in
Table 7.2: Expressions of the 9 sd2 functions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( sd_{2.1} ):</td>
<td>( t_3[h_3, h_2, h_1, p_6, p_5, p_4] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.2} ):</td>
<td>( t_3[h_2, h_1, h_3, p_6, p_5, p_4] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.3} ):</td>
<td>( t_3[h_2, h_3, h_1, p_6, p_5, p_4] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.4} ):</td>
<td>( t_3[h_2, h_1, h_3, p_6, p_4, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.5} ):</td>
<td>( t_3[h_2, h_1, h_3, p_6, p_4, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.6} ):</td>
<td>( t_3[h_2, h_3, h_1, p_6, p_4, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.7} ):</td>
<td>( t_3[h_2, h_1, h_3, p_6, p_4, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.8} ):</td>
<td>( t_3[h_2, h_1, h_3, p_4, p_6, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
<tr>
<td>( sd_{2.9} ):</td>
<td>( t_3[h_2, h_3, h_1, p_4, p_6, p_5] = t_2[p_7, p_4, h_1, h_2] \times v_2[p_7, h_3, p_6, p_5] )</td>
</tr>
</tbody>
</table>

4 conditional branches. With the algorithm shown in Figure 7.8, 4 conditional checks are performed only in the worse case. In the best case, there is only one check, since if the first condition is met, all the following conditional statements are not executed.

### 7.3 Experimental Evaluation

We now report on a detailed evaluation study of the performance of tensor contractions on Tesla T10 and Fermi cards. We designed experiments with the following goals:

- Evaluating overall speedups obtained on Fermi GPUs over Tesla T10 GPUs.
- Understanding the impact of different restructuring optimizations we introduced previously in Section 7.2.2.
- Quantifying the benefits obtained from exploiting two of the architectural features in modern GPUs, which are the shared memory and the streaming support.
- Evaluating the overall scalability obtained on a cluster of Fermi cards.
Our study is based on a computational chemistry suite, NWChem [94]. We specifically evaluated the GPU performance of the non-iterative perturbative triples correction in the CCSD(T) calculation, the widely used coupled cluster method. It is one order-of-magnitude more expensive ($O(N^7)$) than the rest of the calculation, and dominates the total computation time on even moderate sized problems.

The correction involves contributions from three sets of tensor contraction expressions. Each set corresponds to a different versions of a single contraction, derived from permuted versions of the input tensors. Representing them as a single contraction induces array indirections, interfering with optimizations. As a result, each of the three contractions is typically translated into 9 contractions. Throughout this section, we focused on comparison of these 9 resulting ccsd_t_doubles_2 functions, which are the most computing intensive of the three sets. The computations performed in these functions are summarized in Table 7.2.

We consider four distinct problem sizes, which are chosen as a representative of problem sizes arising in real tensor contraction applications. A problem size for this application is a seven-tuple, with seven numbers denoting the size of the seven dimensions used in the three tensors. They also correspond to $h_1$, $h_2$, $h_3$, $p_4$, $p_5$, $p_6$, and $p_7$ in the expressions listed in Table 7.2.

(A) (16, 16, 16, 16, 16, 16), each dimension of the tensors has the same size as each dimension of the thread block configuration.

(B) (64, 16, 16, 16, 16, 16), to represent the case when the uncommon index is relatively large.

(C) (24, 24, 24, 24, 24, 24), represents the case when all dimensions are large.

(D) (16, 16, 15, 19, 20, 20, 20) is a representative problem size selected from the CCSD(T) execution trace.

We evaluated the following six implementation versions:

- **CPU**: Sequential implementation on the CPU as they appear in NWChem

- **t10base**: Baseline implementation on the Tesla T10 GPU, described in section 7.2.1, preceding Section 7.2.1.
Table 7.3: Execution time (milliseconds) of the sd2 functions on problem size A

<table>
<thead>
<tr>
<th>sd2_#</th>
<th>CPU</th>
<th>t10base</th>
<th>t10opt</th>
<th>fermi</th>
<th>t10base</th>
<th>t10opt</th>
<th>opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>652</td>
<td>229</td>
<td>44</td>
<td>82</td>
<td>18</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>894</td>
<td>267</td>
<td>39</td>
<td>93</td>
<td>24</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>874</td>
<td>229</td>
<td>45</td>
<td>91</td>
<td>26</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>657</td>
<td>237</td>
<td>54</td>
<td>84</td>
<td>21</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>894</td>
<td>272</td>
<td>80</td>
<td>90</td>
<td>44</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>873</td>
<td>237</td>
<td>56</td>
<td>85</td>
<td>47</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>658</td>
<td>236</td>
<td>52</td>
<td>79</td>
<td>21</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>894</td>
<td>275</td>
<td>84</td>
<td>87</td>
<td>48</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>873</td>
<td>236</td>
<td>46</td>
<td>81</td>
<td>42</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- **t10opt**: Implementation optimized for the Tesla T10 GPU – incorporates all optimizations in Section 7.2.1.
- **fermi-t10base**: The baseline GPU implementation on the Fermi GPU
- **fermi-t10opt**: Version optimized for Tesla T10 (with changes to adapt for larger shared memory, etc.) on the Fermi GPU
- **fermi-opt**: The optimized Fermi GPU implementation

*Throughout this section, all reported times are in milliseconds, with an exception of Table 7.13, where the times are in minutes.*

### 7.3.1 Overall Speedups on Fermi

We first evaluate the overall execution times on the CPU and the two GPUs considered. We report separate speedups for the nine compute-intensive functions in CCSD(T), which
Table 7.4: Execution time (milliseconds) of the sd2 functions on problem size B

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>t10base</th>
<th>t10opt</th>
<th>fermi</th>
<th>t10base</th>
<th>t10opt</th>
<th>opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2624</td>
<td>925</td>
<td>174</td>
<td>325</td>
<td>67</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3622</td>
<td>1181</td>
<td>151</td>
<td>365</td>
<td>156</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3527</td>
<td>1085</td>
<td>181</td>
<td>364</td>
<td>168</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2274</td>
<td>1110</td>
<td>235</td>
<td>333</td>
<td>81</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3125</td>
<td>1235</td>
<td>312</td>
<td>364</td>
<td>172</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3040</td>
<td>1110</td>
<td>243</td>
<td>363</td>
<td>183</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2278</td>
<td>1106</td>
<td>225</td>
<td>334</td>
<td>79</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3125</td>
<td>1261</td>
<td>332</td>
<td>358</td>
<td>186</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3041</td>
<td>1107</td>
<td>228</td>
<td>361</td>
<td>189</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

were listed earlier. We also use the four different problem sizes we stated earlier, which are denoted as (A), (B), (C), and (D), respectively. In the next 2 subsections, we quantify the performance gains from each individual optimization.

The execution times for problem sizes A, B, C, and D are shown in Tables 7.3, 7.4, 7.5, and 7.6, respectively. All versions on the GPU perform better than the CPU with the more optimized versions and the Fermi GPU achieving the best results.

Different problem sizes incur very different execution times. Even for a given problem sizes, the seemingly similar contractions incur different costs. While the nine functions involve the same floating point operation count, the indexed being contracted and the implicit data permutation impacts the data access pattern resulting in different costs for each function.

Particularly, functions sd2_1, sd2_4, and sd2_7 have the same index order in both input and output, therefore, the accesses to both input and output are coalesced. Hence, memory accesses are a smaller component of the kernel execution, and as a result, the speedups for
<table>
<thead>
<tr>
<th>sd2_*</th>
<th>CPU</th>
<th>t10base</th>
<th>t10opt</th>
<th>fermi</th>
<th>t10base</th>
<th>t10opt</th>
<th>opt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10669</td>
<td>2847</td>
<td>511</td>
<td>997</td>
<td>263</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14518</td>
<td>2878</td>
<td>551</td>
<td>1156</td>
<td>349</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12158</td>
<td>2916</td>
<td>531</td>
<td>1027</td>
<td>363</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10651</td>
<td>3085</td>
<td>639</td>
<td>964</td>
<td>289</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14518</td>
<td>3101</td>
<td>811</td>
<td>1160</td>
<td>794</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12156</td>
<td>3090</td>
<td>801</td>
<td>1049</td>
<td>574</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>10668</td>
<td>3076</td>
<td>638</td>
<td>964</td>
<td>294</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>14520</td>
<td>3156</td>
<td>880</td>
<td>1154</td>
<td>799</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12156</td>
<td>3077</td>
<td>798</td>
<td>1050</td>
<td>578</td>
<td>168</td>
</tr>
</tbody>
</table>

Table 7.5: Execution time (milliseconds) of the sd2 functions on problem size C

these functions are higher for the Fermi card. `sd2_5` has a worse performance degradation on Tesla, because of the large stride in the accesses to the output element. As a result, performance improvement on Fermi was very large.

There was a relatively moderate variation across problem sizes, when the common index is larger (problem sizes (C) and (D)), the speedup is less, likely due to device memory accesses.

The baseline versions on both GPUs improve upon the CPU, with Fermi expectedly achieving greater speedups. The version optimized for Tesla T10 significantly improves upon the baseline version on the Tesla T10 GPU(2.5 ~ 7.8). The results show that the optimizations targeted at the Tesla T10 GPU also benefit the Fermi. While these benefits are observed over the baseline when this version is run on the Fermi GPU, the speedup is lower (1.4 ~ 4.8). We believe this is due to the architectural improvements in Fermi that have reduced the burden on the programmer in extracting a reasonable performance out of the GPU. While our results show that careful optimization is rewarded with strong
Table 7.6: Execution time (milliseconds) of the sd2 functions on problem size D

<table>
<thead>
<tr>
<th>sd2__</th>
<th>CPU</th>
<th>t10base</th>
<th>t10opt</th>
<th>fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>t10base</td>
<td>t10opt</td>
<td>opt</td>
</tr>
<tr>
<td>1</td>
<td>1384</td>
<td>411</td>
<td>75</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>1634</td>
<td>458</td>
<td>69</td>
<td>162</td>
</tr>
<tr>
<td>3</td>
<td>1441</td>
<td>407</td>
<td>78</td>
<td>161</td>
</tr>
<tr>
<td>4</td>
<td>1376</td>
<td>421</td>
<td>106</td>
<td>154</td>
</tr>
<tr>
<td>5</td>
<td>1633</td>
<td>466</td>
<td>167</td>
<td>155</td>
</tr>
<tr>
<td>6</td>
<td>1441</td>
<td>421</td>
<td>106</td>
<td>156</td>
</tr>
<tr>
<td>7</td>
<td>1375</td>
<td>420</td>
<td>105</td>
<td>145</td>
</tr>
<tr>
<td>8</td>
<td>1631</td>
<td>476</td>
<td>187</td>
<td>157</td>
</tr>
<tr>
<td>9</td>
<td>1430</td>
<td>421</td>
<td>97</td>
<td>156</td>
</tr>
</tbody>
</table>

speedups, even a baseline mapping of the computation on to the GPU is more beneficial on Fermi. The Tesla T10-optimized version does not maximally exploit the Fermi GPU, as shown by the additional speedups (2.1 sim5.0) obtained by fermi-opt over fermi-t10opt.

The Fermi GPU considered has a peak floating point rate 6.6 times that of the Tesla T10 GPU. For the various scenarios considered in terms of problem sizes and tensor contraction expressions, we observe an average speedup factor of 5.2, corresponding to a 78% of the improvement in the peak flop rate.

In order to better understand the performance implications, we show the speedups of the various GPU versions with respect to the CPU version in Figure 7.9. fermi-opt achieves a speedup factor between 54 and 104 over the CPU.
7.3.2 Impact of Restructuring Optimizations

In Section 7.2.2, we had described four different optimizations we had performed while porting tensor contractions from Tesla to Fermi. In this subsection, we report on the improvement obtained from each optimization.

**Removing Data Transfers:** As addressed in Section 7.2.2, the execution of tensor contraction functions gets significant performance gain when redundant data movement between the host memory and the device memory is removed. In Table 7.7, we have compared the execution times with 3 distinct versions. The first version uses CPU accumulation, based on pipelining of data movement and computation, as described in Section 7.2.1. The second version uses asynchronous copy over PCIe. In this algorithm, we used a similar pipelining scheme, but with different stages. Instead of using CPU to do the accumulation, the output matrix is first copied to the device asynchronously in one stream. Then, the kernel function performs accumulation on the data on the device memory. In the third stage, the data in that stream is copied from the device memory to the host memory asynchronously. This version is only feasible on the Fermi card because it supports bi-directional transfer on PCIe. This optimization involved non-local changes in the NWChem source code to co-ordinate data placement across kernel invocations.

Columns 2-4 in the Table 7.7 report performance of these three versions on the four problem size. As a representative, we have reported performance from one of the functions. We can see that by avoiding CPU accumulation and reducing the frequency of data transfers over PCIe, the performance is improved $3\sim6$ times. Performance on the other 8 functions is very similar, and the details are not shown here.

**Effect of Register Tiling:** We compared the original algorithm, which was developed on Tesla T10 cards, with the algorithm based on the register tiling optimizations. In Table 7.8, the second column shows the performance of the original algorithm on a Fermi card, and the third column shows the performance of the algorithm with register tiling. As the benefit of this optimization varied across different functions, we are showing results from three representative functions in Table 7.8. Overall, across four different problems sizes and 9 functions, register tiling results in improvements between the factor of 1.7 and 3.1. One
observation is that functions in which the accesses are less coalesced (such as sd2.5) has poorer performance on Tesla T10 card, leading to a better speedup of Fermi.

**Changing Index Order on Fermi:** We also evaluated the effect of changing the index order on Fermi GPUs. In the version optimized for Tesla T10 cards, index calculation was done in a way that favors access to the input. This is because without caching, more time was spent on loading of the input. On Fermi GPUs, because of caching and reduced overhead of loading input data, we made an effort to reduce the access time to the output data.

Our experiments show that this strategy helps improve the performance on Fermi GPU significantly. Table 7.9 shows the performance with the two index calculation orders, using 3 of the tensor contraction functions. For the first of these functions, the performance of 2 versions is almost identical. For the second function, there is a 20-40% improvement, depending upon the problem size. For the third function we report performance from in Table 7.9, the improvement from the new version ranges between 3.2 and 3.9.

These three functions turn out to be representative of the 9 functions. Specifically, function sd2.4 and sd2.7 show a similar trend as sd2.1, i.e. neither of them benefit from the new version. The reason is that the inner-most index of the input and output matrices are the same, which leads to the same access pattern with both the orders. The function

<table>
<thead>
<tr>
<th>Size</th>
<th>With CPU Accumulation</th>
<th>With Asynchronous Copy</th>
<th>Without Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>51.8</td>
<td>41.3</td>
<td>11.8</td>
</tr>
<tr>
<td>(B)</td>
<td>203.2</td>
<td>162.4</td>
<td>44.5</td>
</tr>
<tr>
<td>(C)</td>
<td>607.1</td>
<td>458.9</td>
<td>158.1</td>
</tr>
<tr>
<td>(D)</td>
<td>94.0</td>
<td>71.2</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Table 7.7: Execution time of sd2.5 with different data transfer schemes on Fermi (milliseconds)
Table 7.8: Execution times with and without register tiling on Fermi (milliseconds)

<table>
<thead>
<tr>
<th>Size</th>
<th>Without Register Tiling</th>
<th>With Register Tiling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd2.1</td>
<td>(A) 17.0</td>
<td>7.4</td>
</tr>
<tr>
<td></td>
<td>(B) 65.0</td>
<td>26.7</td>
</tr>
<tr>
<td></td>
<td>(C) 257.1</td>
<td>106.7</td>
</tr>
<tr>
<td></td>
<td>(D) 37.3</td>
<td>16.8</td>
</tr>
<tr>
<td>sd2.2</td>
<td>(A) 17.7</td>
<td>10.7</td>
</tr>
<tr>
<td></td>
<td>(B) 69.5</td>
<td>40.0</td>
</tr>
<tr>
<td></td>
<td>(C) 265.8</td>
<td>139.9</td>
</tr>
<tr>
<td></td>
<td>(D) 37.5</td>
<td>21.2</td>
</tr>
<tr>
<td>sd2.5</td>
<td>(A) 30.0</td>
<td>11.8</td>
</tr>
<tr>
<td></td>
<td>(B) 115.0</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>(C) 486.0</td>
<td>158.1</td>
</tr>
<tr>
<td></td>
<td>(D) 53.6</td>
<td>24.7</td>
</tr>
</tbody>
</table>
sd2_2 shows a very similar trend to the function sd2_3, i.e. the second function we report performance on, and where the new version is 20-40% faster. For problem size (B), the big difference is because this problem size implies a higher output/input ratio, and using the algorithm which favors input causes a more severe penalty.

The remaining four functions, including sd2_5 that we report performance from, have much larger speedup by using the index order that favors the output. This is because in sd2_2 and sd2_3, p4 serves as the stream index, and is not decoded as the first index of t2. So, when the index order favors the input, the first index in t2 being decoded is h1, which leads to a smaller stride for accesses to t3. In contrast, for the function sd2_5 and three other similar functions, p4 is decoded as the first index in input. As a result, elements in the output that are accessed by adjacent threads are scattered far from each other. In another word, the stride between two elements accessed by two adjacent threads is very large. Overall, we can see that for 6 out of 9 functions, we achieve a significant performance improvement, ranging from 20% to almost a factor of 4.

We also compared the performance of these two versions on Tesla T10 cards. On the average, the speedup achieved from using the different index calculation order was very small, so we do not show the details of this experiment here.

Impact of Reversed Conditional Branch: By checking the conditions in a reversed order, we were able to reduce the number of conditionals that are executed on the average. As we show in Table 7.10, this results in a performance improvement of 5~18%. We have shown results from only one function, as all the other functions show a similar improvement. It should be noted that this optimization is relevant only for Fermi as the number of branches increased due to register tiling (only possible because of a larger register file).

7.3.3 Benefits from Other Optimizations

We also conducted experiments evaluating the benefits from two architectural features of Fermi cards – the support for bi-directional streaming and a larger shared memory.

Effect of Streaming: Earlier, in Table 7.7, we had shown that the without copy version, where data transfer volume was significantly reduced, had much better performance as compared to the versions where streaming or pipelining was used. We now evaluate if
<table>
<thead>
<tr>
<th>Size</th>
<th>Favors Input</th>
<th>Favors Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sd2_1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>7.4</td>
<td>7.4</td>
</tr>
<tr>
<td>(B)</td>
<td>27.4</td>
<td>26.7</td>
</tr>
<tr>
<td>(C)</td>
<td>106.4</td>
<td>106.7</td>
</tr>
<tr>
<td>(D)</td>
<td>16.8</td>
<td>16.8</td>
</tr>
<tr>
<td><strong>sd2_3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>13.6</td>
<td>11.0</td>
</tr>
<tr>
<td>(B)</td>
<td>105.5</td>
<td>41.5</td>
</tr>
<tr>
<td>(C)</td>
<td>199.7</td>
<td>149.9</td>
</tr>
<tr>
<td>(D)</td>
<td>27.1</td>
<td>22.6</td>
</tr>
<tr>
<td><strong>sd2_5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>44.5</td>
<td>11.8</td>
</tr>
<tr>
<td>(B)</td>
<td>178.2</td>
<td>44.5</td>
</tr>
<tr>
<td>(C)</td>
<td>596.4</td>
<td>158.1</td>
</tr>
<tr>
<td>(D)</td>
<td>78.7</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Table 7.9: Execution times with different index calculation orders on Fermi (milliseconds)

<table>
<thead>
<tr>
<th>Size</th>
<th>In Order Checking</th>
<th>Reversed Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>12.3</td>
<td>11.8</td>
</tr>
<tr>
<td>(B)</td>
<td>46.0</td>
<td>44.5</td>
</tr>
<tr>
<td>(C)</td>
<td>164.4</td>
<td>158.1</td>
</tr>
<tr>
<td>(D)</td>
<td>25.8</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Table 7.10: Execution time of sd2_5 with and without reversed condition checking on Fermi (milliseconds)
Table 7.11: Running time of sd2 functions with and without streaming on the Fermi GPU (milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>Without Streaming</th>
<th>With Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sd2_1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>8.8</td>
<td>7.4</td>
</tr>
<tr>
<td>(B)</td>
<td>33.3</td>
<td>26.7</td>
</tr>
<tr>
<td>(C)</td>
<td>128.7</td>
<td>106.7</td>
</tr>
<tr>
<td>(D)</td>
<td>19.8</td>
<td>16.8</td>
</tr>
<tr>
<td><strong>sd2_3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>12.1</td>
<td>11.0</td>
</tr>
<tr>
<td>(B)</td>
<td>46.7</td>
<td>41.5</td>
</tr>
<tr>
<td>(C)</td>
<td>167.8</td>
<td>149.9</td>
</tr>
<tr>
<td>(D)</td>
<td>25.5</td>
<td>22.6</td>
</tr>
<tr>
<td><strong>sd2_5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>12.1</td>
<td>11.8</td>
</tr>
<tr>
<td>(B)</td>
<td>46.5</td>
<td>44.5</td>
</tr>
<tr>
<td>(C)</td>
<td>168.5</td>
<td>158.1</td>
</tr>
<tr>
<td>(D)</td>
<td>25.4</td>
<td>24.7</td>
</tr>
</tbody>
</table>

streaming can further help improve performance when data movements have already been reduced. Our experiments show that even without using the pipelined structure, streaming still helps improve the performance, as shown in Table 7.11. Specifically, we can see an improvement of between 2 and 20 percent. The main advantage of the streaming version appears to be that one set of division and modulo operations are avoided.

**Benefits of Using Shared Memory** One of the features of Fermi GPUs is the larger shared memory. At the same time, it also has L1 and L2 cache, unlike earlier GPUs. Thus, it is interesting to evaluate the relative gains from the use of shared memory on Fermi GPUs.

We evaluated the versions with and without the use of shared memory on Tesla T10 and Fermi GPUs. The results with the function sd2_5 are shown in Table 7.12. It can
### Table 7.12: Running time of sd2.5 with and without shared memory on both Tesla T10 and Fermi GPUs (milliseconds)

<table>
<thead>
<tr>
<th>Size</th>
<th>Without Shared Memory</th>
<th>With Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tesla T10</strong> (A)</td>
<td>203.7</td>
<td>80.4</td>
</tr>
<tr>
<td>(B)</td>
<td>804.1</td>
<td>312.5</td>
</tr>
<tr>
<td>(C)</td>
<td>2668.7</td>
<td>811.4</td>
</tr>
<tr>
<td>(D)</td>
<td>427.0</td>
<td>167.5</td>
</tr>
</tbody>
</table>

| **Fermi** (A) | 19.9 | 11.8 |
| (B) | 78.0 | 44.5 |
| (C) | 301.0 | 158.1 |
| (D) | 42.4 | 24.7 |

be seen that on Tesla T10 GPU, using shared memory provides a speedup of about 2~4. On the other hand, the speedup of using shared memory on Fermi card is slightly under 2. Thus, despite a larger shared memory, degradation from not using shared memory is smaller with the Fermi card. This is because of L1 and L2 cache. However, as we can see from these results, despite the presence of L1 and L2 cache, careful programmer controlled use of shared memory is still critical for obtaining performance on Fermi cards.

### 7.3.4 Application Scalability on a GPU Cluster

Using Aggregate Remote Memory Copy Interface (ARMCI) [125], the full NWChem application has already been parallelized for clusters, including clusters of GPUs. With ARMCI, tasks are obtained by each process with atomic operations. We now evaluate the performance on Fermi and Tesla clusters, focusing on the following two aspects: 1) speedups on
Fermi cluster and Tesla clusters over CPU only executions on the same clusters, and 2) scalability on a Fermi cluster. We ran the entire NWChem application on a green fluorescent protein (model D of [57]).

The experiments were done on 16 nodes on two clusters, one with Tesla T10 GPUs, and the other with the Fermi cards. Each node on the cluster with Tesla T10 GPUs has two Quad-Core Intel Xeon X5560 CPUs, with a frequency of 2.80 GHz, and 8 MB L2 cache. Two nodes share one Tesla S1070 box, implying that every node has two Tesla T10 GPUs. Each node on the Fermi cluster is equipped with two Quad-Core Intel Xeon E5520 CPUs, with the frequency of 2.27 GHz. One node has a single GPU.

Table 7.13 shows the performance of the CCSD(T) part on the two machines, where the execution times are reported in minutes. Only one process on each node is used, for both CPU and GPU versions. It can be seen that the Fermi cluster has a speedup of about 3.4 over the Tesla cluster, with the same number of nodes. The CPU version in the Tesla T10 cluster is faster because of faster CPUs. On the Fermi cluster, GPU based versions result in a factor of 43 improvement over the single core CPU version. In comparison, this speedup is only 9 on the Tesla cluster.

In order to evaluate the scaling behavior on at smaller node counts, we did scaling experiments with smaller inputs, which are the uracil and NiO₂ molecules. The execution times are shown in Figure 7.10. We can see an almost linear speedup for both problem sizes.

<table>
<thead>
<tr>
<th></th>
<th>1 core</th>
<th>With GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fermi Cluster</td>
<td>552</td>
<td>13</td>
</tr>
<tr>
<td>Tesla T10 Cluster</td>
<td>409</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 7.13: Execution time of CCSD(T) part of NWChem with Fermi GPU on Green Fluorescent Protein (16 nodes) (minutes)
7.4 Summary and Discussion

With the availability of Fermi cards with enhanced support for double precision operations, GPGPUs have become a critical element of high performance computing. While they offer excellent peak performance to price ratio, often much effort is required in tuning a particular application.

A detailed study is conducted involving tensor contraction expressions on Tesla T10 and Fermi GPUs. We show how a number of optimizations were applied in view of the architectural features of Fermi cards. Our detailed evaluation has shown that optimizations that either are not applicable or do not impact performance on Tesla T10 series cards can lead to up to a factor of 4 improvement in the performance on Fermi cards. After all optimizations, we achieve a speedup of between 3 and 7 over Tesla T10 cards. On a full application involving tensor contractions, we achieved a speedup of 3.4 on a Fermi cluster, over a Tesla cluster with the same size, and a factor of 43 speedup over the single core CPU version. We have also demonstrated scalability of our implementation with the size of the cluster.

Our experiences in developing optimized implementations of tensor contraction expressions on the Fermi GPU architecture identified application-specific (eg., improved conditionals) and more general optimizations (eg., register tiling). This work also serves to understand the challenges to be faced and addressed when a kernel is ported from the Tesla T10 cards to the Fermi GPUs. We observe that data movement is more of a bottleneck on the Fermi GPUs. While the availability of caches can alleviate some of the burden on the programmer, effective shared memory utilization is still an important factor, necessitating explicit shared memory management. For one of the nine functions evaluated, explicit shared memory management improved performance by up to a factor of 2 on Fermi, while providing a speedup between 2 and 4 on the Tesla T10 cards. Register tiling showed that some optimizations can enable further code transformations and that optimizing conditional expressions, even those that do not lead to significant thread divergence, is beneficial (up to 18% in our case). Finally, algorithmic changes isolated to a single Tesla T10 kernel might not be sufficient. In particular, our implementation necessitated multiples changes in the
NWChem source code to retain the intermediates in the GPU device memory across kernel invocations.

We believe the optimizations and performance studies reported in this chapter should be of broad interest to applications and tools developers interested in using Fermi cards.
Figure 7.2: Illustration of dimension flattening
1. get_input_data();
2. ccsd_t_single(T3_s);
3. ccsd_t_doubles_1(T3_d);
4. ccsd_t_doubles_2(T3_d);
5. compute_energy(T3_s, T3_d);

Figure 7.3: Major computation in one task (one iteration) in CCSD(T)

/* calculate index for 4 elements on each dimension (x and y) */
for( each tile of the consumed index)
    shm_t2 ← t2 in device memory
    shm_v2 ← v2 in device memory
    for (i=0; i<tile_of_consumed_index;i++) {
        a1 = shm_t2[threadID][i];
        ...
        a4 = shm_t2[threadID+3][i];
        b1 = shm_v2[threadID][i];
        ...
        b4 = shm_v2[threadID+3][i];
        result1=a1*b1;
        result2=a1*b2;
        ...
        result16=a4*b4;
    }
    t3 in device memory ← result1...result16

Figure 7.4: Tensor contraction with register tiling
rest_x = 16 * 4 * blockIdx.x + threadIdx.x;
rest_y = 16 * 4 * blockIdx.y + threadIdx.y;
i_0 = rest_y % i_d;
rest_y = rest_y / i_d;
a_0 = rest_y % a_d;
rest_y = rest_y / a_d;
b_0 = rest_y;
c_0 = rest_x;

Figure 7.5: Index calculation with input order

rest_x = 16 * 4 * blockIdx.x + threadIdx.x;
rest_y = 16 * 4 * blockIdx.y + threadIdx.y;
b_0 = rest_y % b_d;
rest_y = rest_y / b_d;
a_0 = rest_y % a_d;
rest_y = rest_y / a_d;
i_0 = rest_y;
c_0 = rest_x;

Figure 7.6: Index calculation with output order

if(thread_y < total_y)
    t3[offset_1] += tlocal1
if(thread_y + 16 < total_y)
    t3[offset_2] += tlocal2;
if(thread_y + 16 * 2 < total_y)
    t3[offset_3] += tlocal3;
if(thread_y + 16 * 3 < total_y)
    t3[offset_4] += tlocal4;

Figure 7.7: Boundary checking – original order

114
if(thread_y+16*3<total_y) {
    t3[offset_1]+=tlocal1;
    t3[offset_2]+=tlocal2;
    t3[offset_3]+=tlocal3;
    t3[offset_4]+=tlocal4;
}
else if(thread_y+16*2<total_y) {
    t3[offset_1]+=tlocal1;
    t3[offset_2]+=tlocal2;
    t3[offset_3]+=tlocal3;
}
else if(thread_y+16<total_y) {
    t3[offset_1]+=tlocal1;
    t3[offset_2]+=tlocal2;
}
else if(thread_y<total_y) {
    t3[offset_1]+=tlocal1;
}

Figure 7.8: Boundary checking – reversed order
Figure 7.9: Speedup of the various versions with respect to the CPU. x-axis - the different sd2 functions. y-axis - speedup

Figure 7.10: Execution time of CCSD(T) part of NWChem with GPU on Uracil and NiO₂ molecules
CHAPTER 8

AUTOTUNING FOR TENSOR CONTRACTIONS ON

GPUS

In the previous chapter, we showed that there is significant variation between the favorable algorithms on the two types of GPUs. Thus, it is desirable to have some mechanism to automatically optimize the code for various functions on various architectures. Therefore, we developed an auto-tuning system to solve this problem.

Auto-tuning has emerged as a key approach for obtaining performance on important computational kernels. One of the reasons for the success of auto-tuning, over other approaches like code generation driven by modeling [31], has been the increasing complexity of architectures. The initial work on auto-tuning was driven by processors in late 90s, which had multiple levels of cache. As a result, a kernel like matrix multiplication had many parameters (like unrolling factor, tile size for each level of hierarchy), which could have a significant impact on the performance. Thus, an empirical search, with intelligent heuristics to reduce the search space, was proposed as a practical means for tuning a specific kernel [12].

Given the challenge of extracting performance from a modern GPU, there has been a lot of interest in auto-tuning for GPUs as well, with extensive studies focusing on kernels like dense matrix multiply [103], sparse matrix computations [41], and FFTs [54]. Despite the large number of studies on GPUs, performance tuning of many complex real-world applications involve challenges that are not adequately addressed by the current frameworks. Our auto-tuning framework is motivated by Tensor contractions in NWChem [94]. The interesting challenge that arises with an application like NWChem is that a single application involves several different contractions, which differ significantly in the size of the operands.
and the nature of the computations. Moreover, in the same environment, NWChem may be executed with different problem sizes, leading to expressions with different dimensionalities and problem sizes.

The optimized versions of these expressions be quite different, because of the different size and shapes of operands and differences in the calculations performed. Thus, some of the existing auto-tuning approaches, if applied to NWChem, will involve using expensive empirical search for each of the contractions, and thus will be extremely expensive. Other methods, which might involve accurate cost models, can only be applicable if the underlying architecture can be modelled very accurately. Unfortunately, an architecture like Fermi has many features that are not well understood, and one cannot expect performance of a tensor contraction on a Fermi card to be captured through an accurate and complete cost model.

Driven by the need for tuning an application like NWChem for an architecture like the Fermi GPUs, I will present a novel approach for auto-tuning. The key aspect of our approach is the use of parametrized micro-benchmarks. Our approach involves the following features:

- Based on a broad knowledge of the target architecture and the class of kernels being optimized, we choose the micro-benchmarks and their parameter space.

- The micro-benchmarks are executed with their chosen parameter space, to learn how the architectural features that can impact the performance of the target kernels.

- Based on the performance characteristics learnt from micro-benchmarks, the best versions of kernels are chosen and executed.

Our approach has several advantages over the existing methods. First, by using a single micro-benchmark or a small number of micro-benchmarks, we avoid empirical search on each distinct compute-intensive task in the application. Second, since the micro-benchmarks and their parameter space are created keeping the target kernels in mind, we focus only on the architectural features that actually impact the performance of the target kernels.

Using our parameterized micro-benchmarking approach, we focus on optimizing for two features of Fermi architecture. The first is the memory hierarchy.
8.1 System Design

This section describes our overall approach for auto-tuning, and how we apply it for tuning tensor contractions on modern GPUs.

8.1.1 Tensor Contractions and Performance Tuning Opportunities

As in the previous chapter, we focus on tensor contractions that occur in the context of the Coupled Cluster (CC) theory [83, 132, 19]. Consider the following tensor contraction.

$$R[h_1,h_2,h_3,p_1,p_2,p_3] \leftarrow T[p_4,p_2,h_3,h_2] \ast V[p_4,h_1,p_1,p_2]$$  \hspace{1cm} (8.1)

The tensor contraction involves computing tensor $R$ from tensors $T$ and $V$. The index $p_4$ above is common to both input tensors, corresponding to the common dimension in matrix-matrix multiplication, and is referred to as the contracted or common index. Unlike conventional matrix multiplication, several dimensions can be contracted in a binary tensor contraction. In comparison, uncommon index set denotes the indices that appear in the output matrices, and one (but not both) of the input tensors.

As we stated previously, tensor contractions arising within a single application could involve different sizes, number of dimensions, and/or other factors which could impact their optimized implementation for a particular architecture. In our paramterized micro-benchmark based auto-tuning approach, we consider how different expressions could differ from each other, as well as how their different implementations can differ from each other. Based on these differences, the micro-benchmark(s) and their corresponding parameter spaces are chosen. Thus, we now summarize how different tensor contractions and their possible implementations can vary.

The tensors involved in an expression could have any number of indices or dimensions. One of the first factors that could impact the performance of a given expression is the ratio between sizes along the common indices and the sizes along the uncommon indices. Particularly, the product of the sizes along the uncommon index determines the amount of data movement from CPU to GPU, and the size of common index determines the ratio between computation and the data transfer.
For example, consider the following two expressions:

\[ C[a, b, c, d] = A[a, b, e, f] \times B[e, f, d, c] \]  \hspace{1cm} (8.2)

\[ C[a, b, c, d] = A[a, f] \times B[b, f, d, c] \]  \hspace{1cm} (8.3)

Assuming each dimension has an identical size, the first expression above has a larger computation to I/O ratio, whereas the total output costs for both the expressions are identical. This factor turns out to be very important in determining the benefit and type of consolidation, as we will demonstrate later.

In terms of the actual implementation of a given kernel, a number of interesting issues arise. The pattern of implementation we use involves a thread block on each SM which can be viewed as a two-dimensional grid, \((x, y)\).

The access strides can be classified as three types: stride between threads with adjacent \(x\) values, denoted as \(\text{stride}_x\), stride between threads adjacent \(y\) values, denoted as \(\text{stride}_y\), and the stride between two iterations of one thread, denoted as \(\text{stride}_\text{iter}\). The threads along a certain dimension \((x\ or \ y)\) access consecutive locations along a certain dimension of the input matrix. To illustrate this, consider the following expression:

\[ C[a, b, c, d] = A[b, e, a, f] \times B[f, e, d, c]. \]  \hspace{1cm} (8.4)

With a \((x, y)\) block of threads, two threads with adjacent values along the \(x\) dimension will read adjacent elements along the common index dimension of the input \(A\). They will also read adjacent elements along the uncommon index dimensions of the input \(B\). When writing to the output, they will access adjacent elements along the \(x\) dimension of \(C\), which is calculated by the uncommon index of the input \(B\). Similarly, two threads with adjacent values along the \(y\) dimension will access adjacent elements along the uncommon index of \(A\), and adjacent elements along common index of \(B\), and adjacent elements along the \(y\) dimension of \(C\), which come from the uncommon index of \(A\).

The common indices here are \(\{c, f\}\), whereas the uncommon indices are \(\{b, a\}\) in \(A\), and \(\{d, c\}\) in \(B\). Suppose the index calculation order is determined by the output matrix. So the access of uncommon index of \(A\) will be in the order of \(\{a, b\}\). This implies that the index \(b\) is the innermost index, and the \(\text{stride}_y\) for \(A\) will be \(T^3\), where \(T\) is the size of
each dimension. For example, if the size of each dimension is 16, then the stride between $A[0, 0, 0, 0]$ and $A[1, 0, 0, 0]$ is $16^3$. On the other hand, if index calculation order is done with the index order of the input tensor $A$, then the $\text{stride}_y$ for $A$ is $T$, because now $a$ is the innermost index in the uncommon index set.

Because of the features of the GPU memory hierarchy, which we will explain below, the choices of index calculation can have a significant impact on the performance. Particularly, we need to choose the index calculation order among uncommon indices, and the index calculation order among common indices. In both of the cases, we have two options to choose from. To choose uncommon index calculation order, let us revisit the example above. If we choose the output-favored index order, the index calculation order for $C$ will be: uncommon index from $A$: $a$ followed by $b$, and uncommon index from $B$: $c$ followed by $d$. In contrast, if we choose input-favored index order, the index calculation orders will be reversed. For the common index, if we choose index order that favors $A$, the calculation order of common index will $e$ followed by $f$. If we choose index order that favors $B$, the order of common index is now reversed.

All these options lead to different strides between threads along $x$ and $y$ dimensions, and iterations. Thus, our goal is to use our parametrized benchmarks to learn the performance on a given machine, and to choose these orders.

### 8.1.2 GPU Architecture Features

We now explain two of the important features of the modern GPU architectures (Tesla 10-series and Tesla 20-series (or Fermi) GPUs), which are the memory hierarchy and support for consolidation.

**Device Memory Accesses:** There are several important characteristics associated with the cost of device memory accesses on Fermi and T10 architectures. First, coalesced accesses, where simultaneously executing threads access consecutive memory locations, help improve performance over the cases where a large stride value is involved. Second, the Fermi series of cards also have a L1 and a L2 cache. The L1 cache is either 16 or 48 KB, depending upon the size of the shared memory that is chosen, whereas L2 cache is 768 KB.

**Consolidation:** Though GPUs initially focused on exploiting data parallelism, recently
task parallelism has also been supported through *concurrent kernels*. Different kernel functions can be launched simultaneously, with each kernel performing a distinct task. There are at least two cases where concurrent kernels can help improve performance of a single application that involves several independent computations. First, if the amount of computation in a single kernel is not very high, one could run several such kernels together, and obtain overall higher performance for the entire application. This is because modern GPUs have a high-degree of parallelism, which cannot be adequately used by single kernel with a modest amount of computation. Second, consolidation can also be an easy mechanism to achieve an overlap between data movement and computation. By launching several kernels at the same time, we can have one or more of them performing the computation, while others may be transferring the output to the host at the same time. This can further help reduce the total execution time for these sets of kernels.

### 8.1.3 Parameterized Micro-benchmarking Approach

Figure 8.1 gives a description of our approach for auto-tuning. The space of different kernels in the target application, the possible implementations of each, and the problem sizes for which the target application may be executed is analyzed to obtain the micro-benchmarks and their parameter space. In some cases, key architectural features that can
impact performance are also reflected in the micro-benchmark design and the choice of the parameters.

Next, with the execution of the micro-benchmarks, we extract cost models and/or thresholds that could predict performance of any particular implementation of the given kernel. At run time, when the problem sizes and the target architecture is known, these models are used to select the best version.

We have applied and evaluated our approach in the context of auto-tuning for tensor contractions on GPUs. As stated in the previous chapter, we found that the index calculation order plays an important role in the performance of a tensor contraction application. This difference is caused by different access strides to device memory. Therefore, we used a very simple parameterized loop to extract a partial cost model of memory accesses, with different access strides. We consider our cost model to be partial, since it is not designed to predict performance for all cases, but instead, for the cases that are important for choosing the best implementation of a given kernel with a given problem size.

The second feature of the GPUs we have optimized for is the consolidation support, which can help us fully utilize the set of multiprocessors in a GPU and effectively overlap computation and data transfer. Here, we measured the benefit of kernel consolidation with a small kernel based on matrix multiplication. By specifying different sizes for each of the three operands, we can simulate and predict the performance of various tensor contractions with and without kernel consolidation.

8.2 Using Parametrized Micro-benchmarking for Analysis of Device Memory Accesses

Optimizing for memory hierarchy has been one of the most popular optimization for modern uniprocessors and multiprocessors. On GPUs, the most popular optimization has been for using shared memory for frequently accessed data [20, 121, 100]. However, there are other features of the memory hierarchy that need to be carefully optimized for. This includes the support for coalesced accesses, which has targeted in some frameworks [162], and L1 and L2 cache, which are new to the Fermi architecture.
In this section, we describe how we use our parametrized micro-benchmarking approach for developing cost models of memory access performance. Our goal is not to develop a complete cost model, since such models tend to be very complex and difficult to apply for a real machine. Instead, our goal is to capture the features we see in tensor contraction expressions and their different implementations, and on typical problem sizes, and design micro-benchmarks and their parameter space. These, in turn, are used to predict the relative performance of different implementations of the same expression, and choose the best version. The goal is to be able to apply the same model to different tensor contractions executed on different problem sizes.

8.2.1 Micro-benchmarks and Parameter Space

We use a simple loop that accesses different memory locations, with three parameters: \( \text{stride}_x \), \( \text{stride}_y \), and \( \text{stride}_\text{iter} \). \( \text{stride}_x \) denotes the distance between the addresses accessed at any given time by consecutive threads in one warp. Similarly, \( \text{stride}_y \) indicates the distance between the addresses accessed in the same instruction, by threads in consecutive warps. \( \text{stride}_\text{iter} \) is the temporal stride within a certain thread, which means, the distance between the consecutive memory locations accessed by the same thread.

By varying the above three parameters, we can capture the behaviour of a variety of structured applications. However, when considering only tensor contractions, we can simplify the parameter space. Particularly, for tensor contractions, the data accesses follow such a pattern: for the same matrix being accessed, the values of \( \text{stride}_x \), \( \text{stride}_y \), and \( \text{stride}_\text{iter} \) are either 1 or a power of the dimension size, and furthermore, the 3 parameters are always different. For example, suppose the dimension size is \( T \), then if \( \text{stride}_x \) is 1, \( \text{stride}_y \) could be \( T \), \( T^2 \), \( T^3 \), \ldots. Therefore, our tests are conducted with permutations of these three parameters, assuming that the four candidate values are 1, \( T \), \( T^2 \) and \( T^3 \). The following list shows the 6 test sets with \( \text{stride}_\text{iter} \) being 1.

1. \( \text{stride}_x = T, \text{stride}_y = T^2 \)
2. \( \text{stride}_x = T, \text{stride}_y = T^3 \)
3. \( \text{stride}_x = T^2, \text{stride}_y = T \)
4. \( \text{stride}_x = T^2, \text{stride}_y = T^3 \)

5. \( \text{stride}_x = T^3, \text{stride}_y = T \)

6. \( \text{stride}_x = T^3, \text{stride}_y = T^2 \)

With larger numbers of dimensions, the number of configurations can be larger, but follows a similar trend. However, to simplify the parameter space, we measured strides only less than or equal to \( T^3 \). This is a reasonable simplification, because the cost of accesses with a stride larger than \( T^3 \) is usually similar to the cost with a stride equal to \( T^3 \).

We suppose the dimension size is between 8 and 40, according to the common tensor contraction expressions and the memory constraints of current systems. In the expressions we derive, \( x \), \( y \), and \( \text{iter} \) are used to represent \( \text{stride}_x \), \( \text{stride}_y \), and \( \text{stride}_\text{iter} \).

### 8.2.2 Analysis of Device Memory Accesses on Fermi

On the Fermi or 20-series architecture, to predict the performance of tensor contractions more precisely, we need to consider the output matrix of tensor contractions. On the Fermi card, register tiling is used for tensor contractions, and therefore, \( \text{stride}_\text{iter} \) for the output matrix is almost always 16. So we also tested all combination of values of \( \text{stride}_x \) and \( \text{stride}_y \) with \( \text{stride}_\text{iter} \) being 16, no matter whether \( \text{stride}_x \) or \( \text{stride}_y \) has the same value as \( \text{stride}_\text{iter} \).

Therefore, the following functions are developed to describe the cost of a certain memory access pattern, taking \( \text{stride}_x \) and \( \text{stride}_y \) as input parameters. In the following formula (and the formula in the next subsection), \( S \) represents a constant value. The three lines in Figure 8.2 show the four main cases we will consider here. Note that there are many additional cases, but for simplicity, we only focus on these four cases. With some approximations, we have:

\[
\text{cost} = \begin{cases} 
2S, & \text{if } x = 1, \ 4S, & \text{if } y = 1, \\
(1.1T - 5)S, & \text{if } x = T, y = T^3, \text{iter} = 16, T < 16 \\
13S, & \text{if } x = T \ y = T^3 \ \text{iter} = 16, \ T \geq 16; 
\end{cases}
\]
When $x$ is 1, the accesses to the device memory are coalesced. Therefore, irrespective of the other parameters, the cost is low. When $y$ is 1, it means consecutive half warps are also accessing the same data, which will enable good cache reuse, therefore the costs are again low. In the other cases, because the accesses are not coalesced, and the cache miss rates are potentially higher, the cost is increased. Therefore, the last case in the above equation has a fixed cost of 13S when the dimension size is $\geq 16$. The 3rd case has the same expression pattern as the fourth case, while the dimension size is smaller than 16. In this situation, a dimension size smaller than 16 will enable better cache performance, therefore, the cost is increasing linearly with the dimension size. After the dimension size reaches 16, the miss rate becomes stable, therefore, we have a constant cost.

8.2.3 Analysis of Strided Accesses to Device Memory on T10 GPU

The experiments on T10 were done with the same configurations, except that to match the access pattern of the output, $\text{stride}\_\text{iter}$ is always 0 instead of 16. This is because register tiling is not used for tensor contraction implementations on these cards. Similar to the cases on Fermi, we can group the configurations into several categories. A set of them (but
Figure 8.3: Memory Access Costs (Main Cases) on T10

not all) are shown in Figure 8.3. The cost expressions derived from them are listed below.

\[
\text{cost} = \begin{cases} 
4S, & \text{if } x = 1 \\
(-1.7T + 36)S, & \text{if } x = T, y = 1, \text{iter} = 1, T < 16 \\
9S, & \text{if } x = T, y = 1, \text{iter} = 1, T \geq 16 
\end{cases}
\]

Similar to the model derived for Fermi, when \( x \) is 1, coalesced accesses enable the lowest possible overhead. In the other configurations, the memory accesses are more expensive. The slope in the upper line of Figure 8.3 is due to the uncoalesced accesses caused by smaller dimension sizes. One important observation is that the cost difference among various cases is smaller for T10 GPUs, and therefore, the performance differences caused by index calculation order may not be very significant with most of the common expressions. The main reason is that there is no L1 and L2 cache on this generation of GPU.

However, there are still situations in which performance could differ a lot when strides are different. Particularly, we see spikes in the costs. One example can be seen in Figure 8.3, when the dimension size is 16, \( x = 1, y = T^3, \text{iter} = T^2 \). These spikes are hard to explain from the documented architectural characteristics made available by the vendor. We can conjecture that for certain combination of configuration and stride size, there are certain conflicts in the memory accesses by adjacent warps, leading to worse performance.

Overall, from the analysis presented in this section, we can see that: 1) parametrized
micro-benchmarks can help summarize cost factors which cannot always be explained from
the known features of the architectures, and 2) there are significant differences in the be-
avour of device accesses between two GPU architectures we have considered, leading to
the possibility that the best version for a given expression may differ across the two cards.
Later, we will evaluate if the models learnt from these micro-benchmarks can help choose
the optimal implementation of real tensor contractions.

8.3 Using Micro-benchmarks to Understand Benefits from
Kernel Consolidation

In this section, we apply our parametrized micro-benchmarking approach to learn the ben-
efits from kernel consolidation supported in modern GPUs. Kernel consolidation has been
introduced in modern GPUs to help exploit task parallelism, in addition to data parallelism
that has always been supported. There are several reasons why exploiting task parallelism
can be important. Particularly, a modern GPU like Fermi has a high degree of parallelism,
and not all tasks may have sufficient data parallelism to keep all resources fully utilized.

Kernel consolidation turns out to be useful for tensor contraction based applications as
well. This is because in an application like NWChem, a fixed tile size (along each dimension)
is used for all tensor contractions, according to the memory limit and the dominant matrix.
For example, suppose on a system with 1 GB device memory, the dominant matrix is 6-
dimensional, i.e., it has the size $T^6$, where $T$ is the tile size along each dimension. With
double precision numbers, we can determine that $T$ should be no more than 22, to meet
the memory constraints. Consequently, for expressions involving matrices with only a small
number of dimensions (for example, two-dimensional or four-dimensional matrices), which
tend to be quite common, a GPU’s resources are not fully utilized. Fortunately, there exists
a significant amount of task parallelism across such computations. So, in these cases, we
can launch these kernels together, or consolidate them.

Besides using task parallelism, there is another advantage of kernel consolidation, which
is achieving an overlap between data movement and computations. Thus, kernel consoli-
dation can be beneficial even if the amount of computation in each kernel is sufficient to
utilize all resources of the GPU. But, it turns that different methods of consolidation need to be used for exploiting task parallelism and achieving overlap between data movement and computations. We refer to these as tightly-coupled and lightly-coupled consolidation.

Before describing these two methods, we show the default procedure for executing a collection of tasks in Figure 8.4, sub-figure (a). Next, we show the tightly-coupled and loosely-coupled methods of consolidation. In both cases, one task will be executed as one stream on the GPU, in which there are three phases: data copy from host memory to device memory, kernel execution, and data copy from device memory to host memory. Figure 8.4, sub-figures (b) and (c) show the differences between the two consolidation schemes. In loosely-coupled consolidation, asynchronous data copy functions are invoked in a batch, then the $N$ kernels are launched, followed by another batch of asynchronous data copy. In other words, the three phases of execution of a single task or kernel are loosely-coupled. In comparison, with tightly-coupled consolidation, we iterate all the $N$ tasks, and invoke the three phases for each task one-by-one.

Compared to the work presented in the previous section, we only consider kernel consolidation on a 20-series or Fermi cards. This is because of two reasons. First, 10-series cards cannot actually execute multiple kernels at the same time, i.e., multiple kernels launched at the same time finish being executed Second, unlike Fermi, which has bi-directional data transfer on PCIe that enables overlapping of data transfer in two directions, 10-series cards only have one-directional data transfer on PCIe. Thus, consolidation cannot help achieve better overlapping on 10-series cards.

In auto-tuning for consolidation, we focus on choosing the right method for a given expression, since different tensor contraction expressions get different benefits from the two methods. Deciding whether or not to perform consolidation is not the critical issue, as long as correctness is maintained, and memory constraints are met. This is because our experiments have determined that consolidation does not degrade the performance in any case.
foreach task i
    Host process get a task;
    copy output data to device memory;
    invoke kernel;
    copy output data to host memory;

(a) Without Kernel Consolidation
allocate memory for data of N tasks on device;
foreach task i
    copy output data to device memory asynchronously;
foreach task i
    invoke kernel asynchronously;
foreach task i
    copy output data to host memory asynchronously;
Synchronize;

(b) Loosely-coupled Consolidation
allocate memory for data of N tasks on device;
foreach task i
    copy output data to device memory asynchronously;
    invoke kernel asynchronously;
    copy output data to host memory asynchronously;
Synchronize;

(c) Tightly-coupled Consolidation

Figure 8.4: Computation of N Tasks With and Without Consolidation

8.3.1 Micro-benchmarks and Parameter Space Selection
We now describe how the micro-benchmark and the possible parameters were chosen to understand how consolidation of tensor contractions can help improve performance on the Fermi architecture. The goal in micro-benchmarking is to be able to avoid experimenting with each separate tensor contraction, i.e. those with different number of dimensions and sizes along each dimension. For this purpose, we used a GPU implementation of matrix multiplication, closely based on the one available from NVIDIA CUDA SDK. The sizes of
Figure 8.5: Comparison of Consolidation of Micro-benchmark with Copy on Fermi

input and output matrices were varied, as part of the parameter space used to understand the architecture’s characteristics.

Specifically, we created test cases as follows. Suppose the matrix multiplication we test is \( C + = A \times B \), where \( A \) is a matrix of size \( HA \times WA \), and \( B \) is a matrix of size \( WA \times WB \). Then, the size of \( C \) is \( HA \times WB \). The absolute and relative costs computation and data movement can be varied by the choosing different values for the dimension sizes. The product of two uncommon indices, i.e. \( HA \times WB \), determines the amount of data being moved for the array \( C \). This turns out to be the dominant I/O cost, since \( C \) needs to be copied both in and out of the GPU memory. Also, most tensor contractions have the pattern that \( C \) is at least as large as the maximum of the sizes of \( A \) and \( B \). The common index \( WA \) determines the ratio between the computation and the data movement.

Overall, we chose three sets of cases, which are as follows.

Pattern 1: The size of the matrix \( B \) is small, where matrices \( A \) and \( C \) have the same size.

Pattern 2: All three matrices have the same size.

Pattern 3: \( A \) and \( B \) have the same size, whereas \( C \) is much larger than \( A \) and \( B \).

As overall problem sizes are scaled, the three cases above capture different scenarios. In the second case, the ratio of computation to data movement also increases with problem sizes. On the other hand, the first and third cases are dominated by data copy times.
8.3.2 Experiments with Micro-benchmarks and Observations

Figure 8.5 shows the performance of the two methods of consolidation, compared for the three patterns, with varying problem sizes. We can see that two consolidation methods can outperform each other for different cases. For Patterns 1 and 3, tightly-coupled consolidation has a better performance, whereas loosely-coupled consolidation is better for the Pattern 2. This is because Patterns 1 and 3 are dominated by data copy times. Thus, the main advantage of consolidation for these cases is in achieving an overlap between the computation and data movement, which is accomplished by tightly coupled consolidation.

On the other hand, expressions following the Pattern 2 are more compute-intensive. In such cases, loosely-coupled consolidation will enable better utilization of the computing resources on the GPU. While the relative performance difference from the two methods are quite small in most cases, the overall benefit from consolidation is quite high (up to 50%).

Overall, our results from micro-benchmarking show that tensor contractions with a larger common index should use loosely-coupled consolidation, whereas expressions with a smaller common index should use tightly-coupled consolidation. Later, we will apply this observation to tensor contractions with different number of dimensions, and determine if the observations from micro-benchmarks are still applicable.

8.4 Experiments

This section reports on a series of experiments we conducted to evaluate the effectiveness of our parametrized micro-benchmarking approach. Particularly, we conducted experiments with the following three goals:

- To demonstrate that the cost models about device memory accesses we learnt from micro-benchmarks can help us choose the best implementation for real tensor contractions. These experiments were performed on both Fermi and Tesla architectures.

- To demonstrate that observations about benefit from different consolidation patterns learn from micro-benchmarks match the observations on actual expressions, on the Fermi architectures.
• Overall, our approach can help obtain a significant performance improvement on an application comprising several tensor contractions.

8.4.1 Applying the Memory Access Cost Models to Tensor Contractions

The cost model for device memory accesses we learnt in Section 8.2 are used to choose among different index orders for a given tensor contraction. The method for choosing the index calculation order is shown in Figure 8.6. First, given an expression, the potential index orders are determined. In some cases, there is only one choice. For example, the index orders of uncommon index will be the same if both input indices and output indices follow the same order. In this situation, we do not have to make any choice. Second, for each index order choice, by looking at the expression, dimension size, and index calculation order, we can determine the three stride values. Then, for each index order, according to the strides and dimension sizes, we can choose which category does this scheme fall into, and the cost for this tensor contraction in this index order will be calculated. After determining the cost of all choices, the one with the smallest cost value is selected.
**Evaluation on Fermi** According to the information extracted from the tests on micro-benchmarks, we use the algorithm in Figure 8.6 to determine index calculation orders. As we mentioned before, there are two cases in which we can choose among different index calculation orders. One is selecting the index calculation order for the output or unconsumed index, which could either be in the order of the output matrix, or in the order of the input matrices. The other is selecting the index order for the common or consumed indices, where our choice could be to use the order from the first input matrix, or the order from the second input matrix.

Focusing on the first case, we show the effectiveness of our approach with two representative tensor contraction expressions, which are listed below:

1. \[ i_2[p_4, p_5, p_6, h_2, h_3, h_1] -= t_2[h_1, p_5, p_4, h_7] \times v_2[h_7, p_6, h_2, h_3] \]
2. \[ i_2[h_2, h_1, h_10, h_3] += t_2[h_7, h_1, p_9, p_3] \times v_2[p_9, h_7, h_10, h_2] \]

In our experiments, we used block configuration of 16 \times 16, and the size of each dimension, \( T \), is varied between 10 and 22, based on the commonly used dimension sizes according to the memory limit of the current generation of GPUs. To use the cost models extracted in Section 8.2, we need to find the strides and number of accesses for each matrix. We used the following method. For the Expression 1, according to our algorithm, for each one block, and in the most common cases, when \( T \) is smaller than or equal to 16, each thread accesses, including both read and write operations, \( t_2 \) 4 times, \( v_2 \) 4 times, and \( i_2 \) 32 times. If \( T \) is larger than 16, \( t_2 \) is accessed 8 times and \( v_2 \) is accessed 8 times, because the iteration number is doubled. Accesses to \( i_2 \) are still 32 in number.

In the Expression 1, there are 6 unconsumed indices. First, let us consider the case when the index calculation is done in the order of the input. For \( t_2 \), we get the following stride values: \( stride_x \) for \( t_2 \) is 1, because when loading \( t_2 \), adjacent threads load adjacent elements along the common index. \( stride_y \) is \( T \), because the threads with adjacent \( y \) values access adjacent elements along the uncommon index. \( stride_iter \) is not important here (set as 1 in the calculations), because there is only one consumed index, implying that each thread will not access the same set of unconsumed indices more than once. For \( v_2 \), the strides are calculated in a similar way, except that the threads with adjacent \( x \) index
<table>
<thead>
<tr>
<th>Tile Size</th>
<th>Predicted (*S) (In order)</th>
<th>Predicted (*S) (Out order)</th>
<th>Actual (ms) (In order)</th>
<th>Actual (ms) (Out order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>272</td>
<td>144</td>
<td>5.854</td>
<td>1.62</td>
</tr>
<tr>
<td>13</td>
<td>304</td>
<td>144</td>
<td>10.37</td>
<td>2.775</td>
</tr>
<tr>
<td>14</td>
<td>336</td>
<td>144</td>
<td>17.23</td>
<td>4.311</td>
</tr>
<tr>
<td>15</td>
<td>368</td>
<td>144</td>
<td>28.425</td>
<td>6.864</td>
</tr>
<tr>
<td>16</td>
<td>432</td>
<td>144</td>
<td>43.832</td>
<td>10.447</td>
</tr>
<tr>
<td>17</td>
<td>448</td>
<td>160</td>
<td>64.177</td>
<td>17.633</td>
</tr>
<tr>
<td>18</td>
<td>448</td>
<td>160</td>
<td>91.177</td>
<td>25.357</td>
</tr>
<tr>
<td>19</td>
<td>448</td>
<td>160</td>
<td>124.684</td>
<td>36.046</td>
</tr>
<tr>
<td>20</td>
<td>448</td>
<td>160</td>
<td>169.466</td>
<td>48.428</td>
</tr>
</tbody>
</table>

Table 8.1: Predicted Memory Access Costs and Actual Execution Times With Different Uncommon Index Orders - Expression 1 on Fermi

<table>
<thead>
<tr>
<th>Tile Size</th>
<th>Predicted (*S) (In order)</th>
<th>Predicted (*S) (Out order)</th>
<th>Actual (ms) (In order)</th>
<th>Actual (ms) (Out order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>120</td>
<td>237.6</td>
<td>0.241</td>
<td>0.295</td>
</tr>
<tr>
<td>13</td>
<td>122</td>
<td>274.1</td>
<td>0.312</td>
<td>0.302</td>
</tr>
<tr>
<td>14</td>
<td>124</td>
<td>314.4</td>
<td>0.425</td>
<td>0.504</td>
</tr>
<tr>
<td>15</td>
<td>126</td>
<td>358.5</td>
<td>0.487</td>
<td>0.584</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
<td>406.4</td>
<td>0.510</td>
<td>0.671</td>
</tr>
<tr>
<td>17</td>
<td>130</td>
<td>820.2</td>
<td>0.681</td>
<td>0.881</td>
</tr>
<tr>
<td>18</td>
<td>132</td>
<td>931.2</td>
<td>1.078</td>
<td>1.471</td>
</tr>
<tr>
<td>19</td>
<td>134</td>
<td>1049.8</td>
<td>1.151</td>
<td>1.519</td>
</tr>
<tr>
<td>20</td>
<td>136</td>
<td>1176</td>
<td>1.438</td>
<td>1.966</td>
</tr>
</tbody>
</table>

Table 8.2: Predicted Memory Access Costs and Actual Execution Times With Different Uncommon Index Order - Expression 2 on Fermi
access adjacent elements along the uncommon index, and threads with the adjacent $y$ index access adjacent elements on the common index. For $i2$, since the indices are calculated in the order of the input matrices, $\text{stride}_x$ is $T$, $\text{stride}_y$ is $T^5$, and $\text{stride}_i$ is 16, because of the register tiling in the algorithm we use.

If the index calculation is done favoring the output, then, for $t2$, $\text{stride}_x$ is still 1 and $\text{stride}_y$ is $T^3$. For $i2$, $\text{stride}_x$ is $T$, $\text{stride}_y$ is 1, since $h1$ is the inner loop, and $\text{stride}_i$ is 16. Strides with $v2$ is the same as in the case the input order is used.

The predicted memory access times and the actual execution times are listed in Table 8.1. Here, In order implies that the index order is calculated in the order of the input data, and Out Order indicates that the order of the output matrix is used. The predicted device memory cost is a multiplication of $S$, which is the constant value used in the cost model. The actual time is in milliseconds. Note that our cost model is used for predicting the device memory access costs, and not the actual execution times. However, different implementations arising from different index order selection can only vary in the device memory costs. Thus, while the device memory access costs may not be proportional to the execution times, they can be used to predict which implementation will have a lower execution time. For this expression, it turns out that out order is always better, irrespective of the dimension size, and this is predicted by the cost model learnt from micro-benchmarks as well.

In the second contraction, the number of accesses are calculated similarly, according to the dimension sizes. If the index calculation favors input, then $\text{stride}_x$ for $v2$ is the stride for the index $h2$, which is 1, $\text{stride}_y$ is $T^2$, and $\text{stride}_i$ is $T^3$. For $i2$, $\text{stride}_x$ is $T^3$, $\text{stride}_y$ is 1, and $\text{stride}_i$ is 16. If the index calculation favors output, then $\text{stride}_x$ for $v2$ is $T$, $\text{stride}_y$ is $T^2$, and $\text{stride}_i$ is $T^3$. For $i2$, $\text{stride}_x$ is $p3.d$, which is $T$, $\text{stride}_y$ is 1, and $\text{stride}_i$ is 16. Strides for $v2$ are the same for input order and output order.

The predicted device memory costs and the actual execution times are shown in Table 8.2. In all cases, the In Order implementation is predicted to have a lower execution time. This matches the actual observation, in all but one case, where the difference between two versions is only 3%. Thus, overall, the results show that our prediction methodology is very accurate, and can lead to optimal or near-optimal implementation version.
Choosing between the input and the output order is not the only choice made for these expressions. Another important consideration is determining the processing order of the common or the consumed index. We consider the following representative contraction:

\[ \text{i2}[h2, h1, h10, p3] += \text{t2}[h7, h1, p9, p3] \times \text{v2}[h2, h10, p9, h7] \]

Assuming that we use the input order scheme, and calculate the common index with the order of indices in \text{v2}, then \text{stride}_x \text{for} \text{t2} \text{is} \ T^3, \text{stride}_y \text{is} \ 1, \text{and stride_iter} \text{is} \ T. \text{For} \text{v2}, \text{stride}_x \text{is} \ T^2, \text{stride}_y \text{is} \ 1, \text{and stride_iter} \text{is} \ T. \text{Now, instead, if we calculate the common index in the order of} \text{t2}, \text{we will calculate} \ p9 \text{first, then} \ h7 \text{is the outer loop. In this case, stride}_x \text{for} \text{t2} \text{is} \ T, \text{stride}_y \text{is} \ 1, \text{and stride_iter} \text{is} \ T^3. \text{For} \text{v2}, \text{stride}_x \text{is} \ T^2, \text{stride}_y \text{is} \ 1, \text{and stride_iter} \text{is} \ T, \text{and stride_iter} \text{is} \ 1.

Using these strides and the cost models extracted earlier, we can predict the device memory access costs. The comparison between the predicted device memory costs and the actual execution times is shown in Table 8.3. We can see that the cost model directs correct selection of index order for every dimension size.

**Evaluation on T10:** The same index order selection approach is tested on T10 GPUs,
<table>
<thead>
<tr>
<th>Tile Size</th>
<th>Predicted (*S) (In order)</th>
<th>Predicted (*S) (Out order)</th>
<th>Actual (ms) (In order)</th>
<th>Actual (ms) (Out order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>32.8</td>
<td>27.8</td>
<td>0.222</td>
<td>0.214</td>
</tr>
<tr>
<td>13</td>
<td>32.8</td>
<td>27.8</td>
<td>0.280</td>
<td>0.270</td>
</tr>
<tr>
<td>14</td>
<td>31.6</td>
<td>26.6</td>
<td>0.364</td>
<td>0.354</td>
</tr>
<tr>
<td>15</td>
<td>31.6</td>
<td>26.6</td>
<td>0.511</td>
<td>0.482</td>
</tr>
<tr>
<td>16</td>
<td>36.8</td>
<td>25.8</td>
<td>0.854</td>
<td>0.644</td>
</tr>
<tr>
<td>17</td>
<td>40.2</td>
<td>40.2</td>
<td>0.943</td>
<td>0.920</td>
</tr>
<tr>
<td>18</td>
<td>36.8</td>
<td>36.8</td>
<td>1.193</td>
<td>1.124</td>
</tr>
<tr>
<td>19</td>
<td>33.4</td>
<td>33.4</td>
<td>1.455</td>
<td>1.354</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>30</td>
<td>1.661</td>
<td>1.553</td>
</tr>
</tbody>
</table>

Table 8.4: Predicted Device Memory Access Costs and Actual Execution Times with Different Uncommon Index Order - T10

<table>
<thead>
<tr>
<th>Tile Size</th>
<th>Predicted (*S) (v2 order)</th>
<th>Predicted (*S) (t2 order)</th>
<th>Actual (ms) (v2 order)</th>
<th>Actual (ms) (t2 order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>216</td>
<td>237.6</td>
<td>0.313</td>
<td>0.328</td>
</tr>
<tr>
<td>13</td>
<td>234</td>
<td>249.6</td>
<td>0.521</td>
<td>0.535</td>
</tr>
<tr>
<td>14</td>
<td>252</td>
<td>260.4</td>
<td>0.732</td>
<td>0.749</td>
</tr>
<tr>
<td>15</td>
<td>270</td>
<td>270</td>
<td>1.046</td>
<td>1.046</td>
</tr>
<tr>
<td>16</td>
<td>304</td>
<td>278.4</td>
<td>4.295</td>
<td>1.334</td>
</tr>
<tr>
<td>17</td>
<td>306</td>
<td>306</td>
<td>2.420</td>
<td>2.437</td>
</tr>
<tr>
<td>18</td>
<td>324</td>
<td>324</td>
<td>3.04</td>
<td>3.051</td>
</tr>
<tr>
<td>19</td>
<td>342</td>
<td>342</td>
<td>4.087</td>
<td>4.071</td>
</tr>
<tr>
<td>20</td>
<td>360</td>
<td>360</td>
<td>5.357</td>
<td>5.197</td>
</tr>
</tbody>
</table>

Table 8.5: Predicted and Actual Cost of Expression with Different Common Index Order on T10

138
where the extracted cost models are quite different. First, to test uncommon index order selection, we used the following tensor contraction expression:

$$t_3[p7, p3, h2, h1] += t_2[p9, p3] \times v_2[h2, h1, p9, h7]$$

The calculated costs and the actual performance is shown in Table 8.4.1. The notations used here have the same meaning as in the results reported from the Fermi cards (Table 8.1 through 8.3). From the table, we can see that for every dimension size, either the cost model directs the right selection, or the model does not show a difference in performance of the two versions, and the actual performance between the versions also differs very marginally.

Next, to evaluate the effectiveness in predicting the common index order, we consider the following contraction:

$$i_2[h2, h1, h10, p3] += t_2[h7, h1, p9, p3] \times v_2[h2, h10, p9, h7]$$

Similar to the evaluation study we performed on Fermi, we have the two index calculation orders here, which will be in the order of $t_2$, and in the order of $v_2$, respectively. If we calculate the common indices in the order of $t_2$, then $p9$ will be the inner loop, and $\text{stride}_x$ for $t_2$ is $T$, $\text{stride}_y$ is 1, and $\text{stride}_\text{iter}$ is $T^3$. For $v_2$, $\text{x_stirde}$ is $T^2$, $\text{stride}_y$ is $T$, and $\text{stride}_\text{iter}$ is 1. In the other order, which is the order of $v_2$, $h7$ will be the inner loop for common index. So, for $t_2$, $\text{stride}_x$ becomes $T^3$ and $\text{stride}_\text{iter}$ is changed to $T$. For $v_2$, $\text{stride}_y$ is now 1, and $\text{stride}_\text{iter}$ is $T$. The other strides are unchanged.

Table 8.5 shows the predicted device memory costs, and actual execution times. In most cases, the predictions with the two versions are the same, and correspondingly, there are no noticeable difference in the observed performance. An exception arises when the dimension size equals 16. We had seen from our micro-benchmarking that the performance degrades sharply in this case, and this is captured through our cost model. The prediction from the cost model indeed matches the performance of the actual expressions. Thus, we can see that our micro-benchmarking based approach can help guide optimizations that would not be possible from completely analytical approaches.
8.4.2 Evaluation of Kernel Consolidation on Tensor Contractions

We now focus on determining if the observations made from micro-benchmarking in Section 8.3 apply to real tensor contractions as well. We used the following three tensor contractions:

- \( i2[h2, h1, h7, p3] += t2[p9, p3] \times v2[h2, h1, p9, h7] \)
- \( i2[h2, h1, h10, p3] += t2[h7, h1, p9, p3] \times v2[h2, h10, p9, h7] \)
- \( t3[p4, p5, p6, h1, h2, h3] -= t2[h2, h1, p4, p7] \times v2[p5, p6, h3, p7] \)

While these tensor contractions involve significantly more complex computations than the matrix-multiplication based micro-benchmarks, there is an important correspondence between the above three expressions and the three patterns of micro-benchmarks used in Section 8.3. Specifically, the first expression above has one large input matrix and one large output matrix, the second expression has three operands with the same size, and the third expression has one small input matrix and an output that is very large in size. Note that if we assume that size of each dimension is the same, these properties are easy to see from the dimensionality of each operand. In this way, these three expressions correspond to the three patterns of the micro-benchmarks used earlier.

Figure 8.7 show the experimental results on each pattern on the Fermi card. The dimension size is varied in these experiments such that the resulting problem sizes correspond to
Figure 8.8: Comparison between Baseline and Auto-Tuned Versions

the problem sizes taken in Figure 8.5. Now, comparing Figures 8.5(a) and 8.7(a), we can see that the micro-benchmark and the real contraction show very similar trends. For all of the problem sizes, tightly-coupled consolidation is better. Larger problem sizes gets more benefit from the tightly-coupled consolidation, because of greater benefit from overlapping data movement and computation. Figure 8.7(b) plots the performance of the second Expression. Similar to the previous case, the micro-benchmark (shown in Figure 8.5(b)) predicts the performance of the real contraction correctly, with loosely-coupled consolidation having better performance. Figure 8.7(c) shows the performance on the last expression, which has a very large output. We can see that the trend is also similar to what is shown by the micro-benchmark.

### 8.4.3 Evaluation on a Collection of Tensor Contractions

To evaluate the overall effectiveness our auto-tuning framework, we did experiments with a collection of tensor contractions. We used 8 different tensor expressions for our experiments, and based on these, created two applications that closely resembles NWChem. The first application is created with 8 invocations for each of the 8 expressions. In the second case, there are 240 invocations of each of two expressions that involve smaller matrices, and 8 invocations for each of the other 6 expressions.

Three sets of experiments were performed using this application. On the Fermi GPU,
we created two versions, one which the output data had to be copied after execution of each task, and another where it did not need to be copied (since it was assumed that the same data is needed for the same task).

The third experiment was performed using the T10 GPU, where we tested the version without data copy. Before applying auto-tuning, we created a baseline version, which itself incorporated several optimizations, but not those that are very specific to the specific GPU architecture. Particularly, in the baseline version, no consolidation is done, uncommon indices are calculated in the order of the output matrix for Fermi, and in the order of input data for T10, and common indices are calculated in the order chosen arbitrarily by the code generator. We chose uncommon index order in this way because there is more redundant data movement for input data in the algorithm of T10, and output is a more dominant factor while execution on Fermi.

From the baseline version, an auto-tuned version was created for each case, using the methods presented in this chapter. Figure 8.8(a) and Figure 8.8(b) show the performance of the baseline version and the tuned version on Fermi, with and without data copy. For both the applications, we selected three dimension sizes: 14, 16, 18, which are typical choices in NWChem. The dimension size denoted with (S) refers to the first application (with smaller number of tasks), while the one denoted with (L) refer to the second application (with larger number of tasks).

On Fermi, for the version with data copy, the auto-tuned version has a speedup of more than 2 over the baseline version. Without data copy, the benefit are lower, but still substantial. When the number of small tasks is increased, the benefit is more obvious.

Figure 8.8(c) shows comparison between tuned version and baseline version without data copy on the T10 GPU, with the same dimension sizes and the same collections of tensor contractions. Since we did not do kernel consolidation for T10, the version with data copy is not tested. We can see that the tuned version is about 27% faster than the baseline version when tile size is 16, and the performance is the same for other tile sizes. This is consistent with the observation in our cost model, and earlier results with individual tensor expressions.

Overall, we can see that our approach can help explore optimizations that are not very
obvious to the programmer, and could be dependent on the specific GPU architecture and/or the problem sizes. Moreover, for a complex application involving several different compute-intensive expressions, we are able to apply the cost models learnt from a simple micro-benchmark effectively.

8.5 Summary

Auto-tuning has been an effective tool to get good performance on different systems, with compile-time and run-time supports. We developed an auto-tuning framework for tensor contraction functions on GPUs, motivated by the fact that algorithmic modification could result in quite different performance for different expressions on various architectures. The main contribution of our work is as follow.

We proposed an approach to evaluate system features with parameterizable micro-benchmarks. By using very simple and standard sample code, and run on a limited data set, we are able to get the influence on performance by different algorithm choices. These extracted features could be used to direct the choices for actual tensor contraction functions, which has been proved to be effective.

We focused on two parameters for tuning. One is the memory access pattern. We extracted some expressions with the micro-benchmarks. The real tensor contractions with a certain problem size could be mapped to the expressions, and get an estimated cost with very simple calculation. In this way, we were able to make the best choice of index calculation order for most cases.

The second parameter we tuned is the pattern of doing kernel consolidation. It is proved that consolidation provides better utilization of the resources. With the observation from micro-benchmarks, we developed a basic principal for selecting kernel consolidation scheme, which also turned out to be effective.

Overall, our system has shown the potential of improving performance with auto-tuning, and the micro-benchmark approach has been proved to be effective.
CHAPTER 9
PRACTICAL LOOP TRANSFORMATIONS FOR
TENSOR CONTRACTION EXPRESSIONS ON
MULTI-LEVEL MEMORY HIERARCHIES

As mentioned in the first chapter, our code generation frameworks focus on two classes of applications. The second class is tensor contractions, which are linear functions widely used in chemical computation, and are extendible to other scientific and commercial computations.

Developing applications on these emerging systems involves complex optimizations. Among other issues, orchestrating data movement across multiple levels of hierarchy, and restructuring the computation to reduce the cost of such data movement, are both challenging problems. As a specific example, consider processing of a disk-resident dataset on a machine with one or more GPUs. If the size of the main memory is smaller than the size of the dataset, there will be significant data movement costs between the disk and main memory. Similarly, it is likely that the GPU device memory is smaller than the main memory. Even though GPU memory capacities have increased rapidly, high-end GPUs today have 3-6 GB memory, and cheaper or older GPUs have much less memory. In comparison, servers today easily have 12 or more GB of main memory. Thus, data movements between main memory and device memory also need to be carefully planned. Moreover, when considering a cluster of GPUs and/or a single node connected with multiple GPUs, the data movement and optimization problems are even more challenging.

Optimizations such as tiling and loop fusion can enable better data reuse and reduce the cost of these data movements. However, what we need is an integrated framework to
enable these transformations across more than 2 levels of memory hierarchy. There has been a considerable amount of work on compiler optimizations for out-of-core programs [27, 92, 91, 25, 154], but the solutions do not directly extend when data movements for another level need to be considered. Similarly, there has been some work on optimizing data movements from main memory to device memory [153, 151, 110].

As multi-level processor caches became very common in mid-nineties, several compiler efforts considered optimizations for them [120, 143, 141]. However, the modern multi-level architectures have very different characteristics.

In this chapter, I present an optimization framework for systems requiring data movement across multiple levels of memory hierarchy [113]. Our work is specifically in the context of tensor contractions [147, 58], which can be viewed as generalized matrix products. Sequences of tensor contractions arise in several domains, particularly, in ab initio computational models in quantum chemistry. Sequence of tensor contractions are often performed over several large multi-dimensional arrays, making it very likely that input data does not fit in main memory. At the same time, these applications are compute-intensive, and can be benefit from GPUs.

We present an extensive framework for optimizing tensor contractions on a system requiring explicit data movement across 3 layers of memory hierarchy. We address the following challenges: 1) determining the loop structure, which comprises of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. While we specifically target machines with GPUs and problems with out-of-core data structures, our framework can be applied on any system with more than 2 explicitly managed memory hierarchy levels.

9.1 Background

In this work, we focus on the optimization of the execution of tensor contraction expressions. These expressions arise in the context of the Coupled Cluster (CC) theory [19], which is a widely used method for solving the electronic Schrödinger equation. A tensor contraction
expression is comprised of a collection of multi-dimensional summations of product of several 
input arrays. Consider the following contraction:

$$B(a, b, c, d) = \sum_{p,q,r,s} C1(s, d) \times C2(r, c) \times C3(q, b) \times C4(p, a) \times A(p, q, r, s)$$

This contraction is referred to as a *four-index transform*. Here, $A(p, q, r, s)$ is a four-
dimensional input array and $B(a, b, c, d)$ is the transformed (output) array. The four arrays,
$C1, C2, C3,$ and $C4$ are called *transformation* arrays. Minimizing the operation count 
results in the following sequence of binary tensor contractions.

$$T3(a, p, r, s) = \sum_p C4(p, a) \times A(p, q, r, s)$$

$$T2(a, b, r, s) = \sum_q C3(q, b) \times T3(a, q, r, s)$$

$$T1(a, b, c, s) = \sum_r C2(r, c) \times T2(a, b, r, s)$$

$$B(a, b, c, d) = \sum_s C1(s, d) \times T1(a, b, c, s)$$

where $T1, T2,$ and $T3$ are the *intermediate* tensors. In each of the four contractions, a set of indices are summed over, or *contracted*, and are referred to as the *summation (or contracted) indices*. These indices occur in both the inputs of a binary contraction and are also referred to as the common indices. This is a generalization of the matrix-matrix 
multiplication operation with individual indices replaced by sets of indices.

The size of each dimension (or loop index) varies from 100 to a few thousand. As a result, 
the input, output, and the intermediate tensors often do not fit into the main memory of 
a single node. This, together with the high floating-point intensity of these expressions, 
lends themselves to execution on a parallel system, and considerable effort has been put on 
parallelization of these methods [77, 14].

Effective execution of tensor contraction expressions on parallel systems with multi-level 
memory hierarchies necessitates several transformations. In particular, the allocation and
freeing of memories in the the multi-level hierarchy together with the associated data transfer needs to be carefully orchestrated. While the execution is dominated by the floating point operations, data movement can quickly overwhelm the total execution time if not effectively managed. Tiling directly benefits these calculations, analogous to its applicability to matrix-matrix multiplication implementations. While much work on matrix multiplication focuses on square matrices, tensor contractions often involve highly rectangular matrices, exacerbated by the higher dimensionality involved. In particular, it is not uncommon for one of the arrays involved in a contraction to be small enough to fit at some higher-level of the memory hierarchy, such as in the example above.

Loop fusion is an integral part of effective data locality management in such multi-level memory hierarchies. Much of the early work on loop fusion either focused on identifying the feasibility of fusing two loops, performing fusion of identified loops [117, 118], possibly with other transformations that enable loop fusion [163], or a general loop transformation framework that encompasses loop permutation, skewing, fusion, and other transformations under an abstract cost model [105, 7, 93].

In our work, we focus on identifying effective fusions for tensor contractions, executed on systems with more than two levels in the memory hierarchy, with as concrete a cost model as possible. Specifically, we want to capture data movement costs under memory constraints. For systems with only two levels of memory hierarchy, this problem has been addressed before by Sahoo et al. [147]. Besides considering additional levels in the memory hierarchy, we also focus on the prescriptive approaches that help us choose candidate loop fusions without requiring expensive optimization procedures. We also note that unlike the work on loop fusion applied to a more general class of applications, tensor contractions, which consist of fully permutible loops, can be fused without requiring enabling loop transformations.

9.2 System Design

In this section, I will introduce the problem statement, followed by the description of our solution framework.
9.2.1 Problem Statement and Notation

Given a sequence of binary tensor contractions, such as the one in Section 9.1, efficient code needs to be generated that is tailored to the specific machine of interest. In our work, we focus on machines comprising more than two levels in the memory hierarchies.

The key decisions to be made are 1) determining the loop structure, comprising of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. It has been shown previously that the space of possible choices is exponentially large [147]. Our work focuses on effective approaches to finding profitable fusion structures, under certain assumptions, for multi-level memory hierarchies.

In our presentation, \( N_i, N_j, \ldots \) and \( T_i, T_j, \ldots \) denote the sizes of the full dimensions and individual tiles along dimensions \( i, j, \ldots \), respectively. Lower case letter denote loop and array reference indices. The memory levels are denoted by \( \alpha, \beta, \ldots \), with \( \alpha \) being the slowest memory hierarchy level. The size of memory available at each level is denoted as \( M_\alpha, M_\beta, \ldots \), and the time to move one element of the data between adjacent levels is \( \Lambda_{\alpha\beta}, \Lambda_{\beta\gamma}, \ldots \). We assume that data movement is always performed between adjacent levels of the memory hierarchy. The cost to perform one double precision floating point operation is denoted by \( C_{\text{flop}} \).

In the next section, we consider optimizations for a single tensor contraction on a multi-level memory hierarchy. In Section 9.2.4, we then present approaches to identify loop fusion for a sequence of tensor contractions for multi-level memory hierarchies.

9.2.2 Single Contraction Optimization

In this section, we determine the loop structure to map a single tensor contraction to a multi-level memory hierarchy.

Single Level Memory

We first summarize existing results for the case when data movement is considered at only a single level (e.g. from disks to main memory). A detailed description is available from an earlier publication [147]. In a tensor contraction, the indices can be grouped into those
that are contracted and the remaining indices in the two input tensors. Given that the loops are fully permutable and we reason about the total data movement volume, ignoring issues such as stride of access, the indices can be grouped into three composite indices. In the ensuing discussion, we refer to the contracted indices as $N_k$, and the remaining indices in the two inputs as $N_i$ and $N_j$.

**One Array Fits in Memory:**

Consider the three tensors $X[x, y]$, $Y[y, z]$, and $Z[x, z]$ in a tensor contraction, where $X$, $Y$, and $Z$ correspond to the three tensors participating in a binary tensor contraction, one of them being the output tensor. Without loss of generality, assume that $X$ fits in memory.

Let the size along the dimensions $x$, $y$, and $z$ be $N_x$, $N_y$, and $N_z$, respectively. It can be shown that to minimize the data movement costs, the smallest array is retained in memory, with the other two arrays being streamed. This results in a memory cost of [147]:

$$N_x \times N_y + \min(N_x, N_y) + 1 \leq M_\beta \quad (4.1)$$

We approximate Equation 4.1 by replacing the left hand side with $N_x \times N_y$. For values of $M_\beta$ encountered in practice, the value of the min term in the worst case ($N_x = N_y = \sqrt{M_\beta}$) is much smaller than the memory size, and can be ignored.

The loop structures for the cases where one of the arrays fits in the memory are shown in Figure 9.1, where $X$, $Y$, $Z$ represent any of the three tensors. Note that none of the tensors are surrounded by loops that do not index it. This ensures that there is no redundant data movement for any of the arrays, achieving the minimum possible data movement cost, i.e. we read each input array and write the output array exactly once [147]. The position of a tensor in the loop structure identifies both data movement and memory management steps. Viewing the loop structure as a loop nest tree, a tensor label, denoted by an allocate statement for a tensor, identifies its position in the loop nest tree. The memory allocation and read statements (if the array is read) are inserted into the structure before the code corresponding to the nested statements are generated. After the processing of all nested statements, the deallocation and write statement (if it is an output array) are generated. While the optimal choice of loop structure that minimizes the memory usage is given by Figure 9.1(a) or Figure 9.1(b), the difference in practice is a small reduction
in memory overhead, not the dominant memory overhead or data movement cost. We, therefore, consider the simplified structure shown in Figure 9.1(c) for further analysis. Figure 9.1(d) illustrates the insertion of memory allocation and data movement statements for the abstract structure shown in Figure 9.1(c) for the contraction $C = A \times B$.

**No Arrays Fit in Memory:**

For a two-level memory hierarchy, say the $\alpha \leftrightarrow \beta$ levels, it has been shown that the optimal loop structure involves the data transfer for the output array being higher in the loop structure than that for the other two arrays. Recalling that tiles sizes along $X$ and $Y$ dimensions are $T_i$ and $T_j$, respectively, the data movement and memory costs are now given by:

$$Data_{\alpha\beta} = \Lambda_{\alpha\beta} \times \left( N_i \times N_j + N_i \times N_j \times N_k \times \left( \frac{1}{T_i} + \frac{1}{T_j} \right) \right)$$  \hfill (4.2)

$$Memory_{\beta} = T_i \times T_j + T_i + T_j < M_\beta$$  \hfill (4.3)

Minimizing the data movement under the memory constraint results in $T_i = T_j = T \approx \sqrt{M_\beta}$. Intuitively, optimized mapping of a tensor contraction to a two-level memory hierarchy.

---

**Figure 9.1:** Loop structures for a single contraction on a two-level memory hierarchy.

(a) Loop structure if $N_y \leq N_x$ (b) Loop structure if $N_x \leq N_y$ (c) Simplified loop structure for discussion (d) Example code for the contraction $C = A \times B$
hierarchy requires maximizing the tile size for the output array with a small amount of data for the input arrays brought in at a time.

9.2.3 Multi-Level Memory Hierarchies

We now focus on extending the existing work to the systems where data movement is required across more than two levels in the memory hierarchy. Specifically, consider a memory hierarchy consisting of three levels, namely $\alpha$, $\beta$, and $\gamma$. The total data movement cost is the sum of costs of the two $\alpha \leftrightarrow \beta$ and $\beta \leftrightarrow \gamma$ data transfers. We shall assume in this discussion that levels cannot be bypassed, with data movement always occurring between adjacent levels. We also assume that the memories get faster and smaller as we move along the memory hierarchies, with $\alpha$ being the slowest level with infinite capacity.

First, if one of the tensors fits into $\gamma$, minimal data movement can be achieved by placing the array in $\gamma$, or reading directly into $\gamma$, and streaming the other two arrays through all levels of memory hierarchy using the loop structure shown in Figure 9.1. Each tensor label in such a schedule corresponds to data transfer across all memory hierarchy levels.

Consider the data movement order with all $\alpha \beta$ transfers grouped to nest the $\beta \gamma$ transfers, with $C_{\alpha\beta}$ being the outermost data placement and the data transfer for the A and B arrays at the same loop nesting. As explained earlier, we ignore the nesting between $A$ and $B$ given the limited memory usage improvement with no change in data movement cost. The costs for this data placement are given by:

$$\text{Data}_{\alpha\beta} = \Lambda_{\alpha\beta} \left( N_i \times N_j + N_i \times N_j \times N_k \left( \frac{1}{T_i} + \frac{1}{T_j} \right) \right)$$  \hspace{1cm} (4.4)

$$\text{Data}_{\beta\gamma} = \Lambda_{\beta\gamma} \times N_i \times N_j \times N_k \left( \frac{1}{T_i'} + \frac{1}{T_j'} + q_k \frac{1}{T_k} \right)$$  \hspace{1cm} (4.5)

$$\text{Memory}_\beta = T_i \times T_j + T_i \times T_k + T_j \times T_k$$  \hspace{1cm} (4.6)

$$\text{Memory}_\gamma = T_i' \times T_j' + T_i' \times T_k' + T_j' \times T_k'$$  \hspace{1cm} (4.7)

Here, $T_i'$, $T_j'$, and $T_k'$ are the tile sizes at the $\gamma$ level, and $q_k$ is 1 if $T_k = N_k$, 2 otherwise. This is because when $T_k = N_k$, the values of $C$ could be written to the output without having to be loaded and accumulated. The tile sizes for the data movement placement can
be determined by solving this non-linear constrained optimization problem. However, the solution requires additional information on the problem size, which is not always available to a compiler. Thus, here our focus is on deriving prescriptive solutions that do not require precise information on the problem size. Where the problem size is needed, we reduced the determination to a quick runtime decision. In the process, we demonstrate the tile sizes for optimal data movement in a single-level memory hierarchy need not match those of a multi-level memory hierarchy.

Note that the data transfer cost at the $\beta\gamma$ level depends on the tile size $T_k$ determined for the $\alpha\beta$ data transfer. The optimal data transfer for the single-level memory hierarchy, when no array fits in $\beta$, involves $T_i = T_j = \sqrt{M_\beta}$ and $T_k = 1$. Even under the assumption that $\Lambda_{\alpha\beta}$ is much higher than $\Lambda_{\beta\gamma}$, for large values of $N_k$, the data transfer cost is dominated by $\text{Data}_{\alpha\beta} + \Lambda_{\beta\gamma} N_i \times N_j \times N_k$. We thus consider another alternative, with $T_i = T_j = T_k = \sqrt{M_\beta/3}$. This corresponds to equal amounts of all three arrays being stored in memory at any time, a less than optimal choice for a two-level memory hierarchy. The total data movement cost (with $T'_i = T'_j = \sqrt{M_\gamma}$) is given by:

$$N_i \times N_j \times \left[ \Lambda_{\alpha\beta} \times \left( 1 + N_k \times \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) + \Lambda_{\beta\gamma} N_k \times \left( \frac{2}{\sqrt{M_\gamma}} + \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) \right]$$

(4.8)

Comparing this cost with Equations 4.4 and 4.5, we observe that equal tile sizes are better when:

$$\frac{2}{\sqrt{M_\beta}}\Lambda_{\alpha\beta} + 2\Lambda_{\beta\gamma} \geq \frac{2\sqrt{3}}{\sqrt{M_\beta}} (\Lambda_{\alpha\beta} + \Lambda_{\beta\gamma}) \implies M_\beta \geq \left( \frac{\sqrt{3} - 1}{\Lambda_{\beta\gamma}} + \sqrt{3} \right) \quad (4.9)$$

This condition is evaluated with $q_k = 2$ in the above determination. It can be quickly evaluated at install time to choose the best tile size. Intuitively, the data transfer between the $\alpha\beta$ memory levels is increased to significantly reduce the cost of transfer between the $\beta\gamma$ levels, keeping it relatively small. While globally optimizing the objective function yields the best solution, the prescriptive solution is independent of the problem size while achieving similar ends.

When one of the arrays fits in $\beta$, the optimal single-level solution involves fitting the array in $\beta$ and streaming the other arrays. When it fits within $(M_\beta/3)$, both schemes result
in similar cost functions and the tile sizes chosen for the single-level optimal solution are chosen. However, for the smallest array size between \((M_β/3)\) and \(M_β\), we have a choice between the two tiling alternatives. With the streaming scheme, without loss of generality, we assume the smallest matrix is \(A\), then by keeping the smallest matrix in \(β\), the tile sizes are \(T_i = N_i, T_k = N_k\), and \(T_j \leq \frac{M_β - N_i N_k}{N_i + N_k}\). The data movement at the \(αβ\) level will be

\[
Λ_{αβ}(N_i N_k + N_j N_k + N_i N_j) \quad (4.10)
\]

This tiling decision at the \(αβ\) level results in tensors of sizes \(T_j, T_j N_k, \) and \(T_j N_i\) residing in \(β\), which need to be scheduled for data movement across \(βγ\). Since \(N_i N_k\) is larger than \((M_β/3)\), we assume it does not fit in \(γ\). However, because of possibly smaller \(T_j, T_j N_k, \) or \(T_j N_i\) might fit \(γ\). In this case, streaming is done in a way similar to \(β\) level. Suppose the smallest tile at \(β\) level is from \(B\), which is of the size \(T_j T_k\), then the total data movement at \(βγ\) is

\[
Λ_{βγ}\left(N_j N_k + N_i N_j + \frac{N_i N_k N_j}{T_j}\right) = Λ_{βγ} N_j (N_k + N_i) \left(1 + \frac{N_i N_k}{M_β - N_i N_k}\right) \quad (4.11)
\]

When none of the data tiles fit in \(γ\), a tiling solution analogous to the single-level tiling presented in Section 9.2.2 is employed. This results in the total data movement cost at \(βγ\) being:

\[
Λ_{βγ}\left(\frac{N_j N_k}{\sqrt{M_γ}} + \frac{N_i N_j}{\sqrt{M_γ}} + \frac{N_i N_k N_j}{T_j}\right) = Λ_{βγ} N_j (N_k + N_i) \left( \frac{2}{\sqrt{M_γ}} + \frac{N_i N_k}{M_β - N_i N_k} \right) \quad (4.12)
\]

The total data movement cost is computed from the \(αβ\) cost (given by Equation 4.12) together with the \(βγ\) cost. The latter is determined using either of the Equations 4.11 and 4.12, based on the problem size. By comparing this overhead with the data movement using equal tile sizes, whose cost is given by the Equation 4.8, the tiling decision could be made at runtime. The decision could also be made at installation time for different matrix shapes, which impact the \((N_i + N_k)\) term.
9.2.4 Fusion for Tensor Contraction Sequences

In this section, we describe our approach to determining fused loop structures for multi-level memory hierarchies. We begin with an analytical approach to identify profitable loop fusions for tensor contractions. Intuitively, we observe that loop fusion can result in substantial performance gains only if the overall execution time is bound by data movement, and not computation. Consider a series of loops, which correspond to the following series of tensor contraction expressions.

\[
I_1(d, c_2, \ldots, c_n) = I_0(d, c_1, \ldots, c_n) \times B_0(d, c_1, \ldots, c_n)
\]
\[
I_2(d, c_3, \ldots, c_n) = I_1(d, c_2, \ldots, c_n) \times B_1(d, c_2, \ldots, c_n)
\]
\[\cdots\]
\[
I_n(d) = I_{n-1}(d, c_n) \times B_{n-1}(d, c_n)
\]

The tensors \(B_0, \ldots, B_{n-1}\) and \(I_0\) are the input tensors and \(I_n\) is the output tensor. \(I_1, \ldots, I_{n-1}\) are the intermediate tensors. Data movement costs for such intermediate tensors can potentially be eliminated through fusion. \(c_j\) corresponds to the indices contracted in the production of \(I_j\). The various \(d\) indices are never contracted out and contribute to the indices in \(I_n\). \(I_i(d)\) represents the set of indices \(I_i\) contributes to the final output. \(I_i(c_j)\) denotes the indices in \(I_i\) that are a subset of the contracted indices in contraction \(j\). \(I_i\) denotes both the tensor produced by contraction \(i\) and set of all indices that constitute that tensor. The reference shall be clear from the context. \(|I_j(d)|\), and similar usage, denotes the total size of the indices in tensor \(I_j\) that contribute to the final output. The relationship between the indices is defined as:

\[
1 \leq i \leq n : I_i(d) = I_{i-1}(d) \cup B_{i-1}(d)
\]
\[
1 \leq i \leq n, 1 \leq j \leq i : I_i(c_j) = \emptyset
\]
\[
1 \leq i \leq n, i + 1 \leq j \leq n : I_i(c_j) = I_{i-1}(c_j) \cup B_{i-1}(c_j)
\]
\[
0 \leq i \leq n - 1 : I_i(c_{i+1}) \equiv B_i(c_{i+1})
\]
The fusion of the loops corresponding to the above tensor expressions is beneficial only if the data movement cost for the intermediate tensors ($I_1$ through $I_{n-1}$) dominates, or is at least comparable to, the data movement cost for the input and the output tensors and the computation cost.

Consider the fusion of the first two contractions. The computation cost of the first contraction is $C_{flop} \left( |I_0(d)| \times \prod_{j=1}^{n} |I_0(c_j)| \times |B_0(d)| \times \prod_{j=2}^{n} |B_0(c_j)| \right)$. The minimum data movement cost for $I_1$ that would be eliminated through loop fusion for the $\alpha\beta$ levels is given by: $2 \times \Lambda_{\alpha\beta} \times \left( |I_1(d)| \times \prod_{j=2}^{n} |I_1(c_j)| \right)$, the cost of reading and writing the tensor once for each of the contractions. Under the requirement that this is greater than the fraction, $0 < \frac{\Lambda_{\alpha\beta}}{C_{flop}} \leq 1$, of the computation cost of the two relevant contractions, we have:

$$2 \times \Lambda_{\alpha\beta} \geq \frac{\Lambda_{\alpha\beta}}{C_{flop}} \times |I_0(c_1)|$$

leading to

$$|I_0(c_1)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{C_{flop}}$$ (5.1)

For the second contraction, the computation cost is given by

$$C_{flop} \times \left( |I_1(d)| \times \prod_{j=2}^{n} |I_1(c_j)| \times |B_1(d)| \times \prod_{j=3}^{n} |B_1(c_j)| \right).$$

If we want the execution to be dominated by the movement of $I_1$, the condition now becomes

$$2\Lambda_{\alpha\beta} \geq \frac{\Lambda_{\alpha\beta}}{C_{flop}} \prod_{j=3}^{n} |B_1(c_j)||B_1(d)| \quad \Rightarrow \quad \prod_{j=3}^{n} |B_1(c_j)||B_1(d)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{C_{flop}}$$ (5.2)

Therefore, for a contraction to be fused with both its previous and the next contraction, the input $B_i$ has to satisfy both of the following requirements:

$$|I_i(c_{i+1})| \leq \frac{2 \times \Lambda_{\alpha\beta}}{C_{flop}}$$

$$\prod_{j=i+2}^{n} |B_i(c_j)| \times |B_i(d)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{C_{flop}}$$ (5.3)
Combining the above two expressions, together with the fact that $I_i(c_{i+1}) = B_i(c_{i+1})$, the size of $B_i$ should be less than $\left(\frac{2 \times \Lambda_{\alpha\beta}}{\frac{\text{froc}}{\text{C}_{\text{flop}}}}\right)^2$. In current systems, this number is typically much smaller than the memory size. We therefore assume that the $B$ tensors in all contractions in the sequence considered, except the first and last contraction, fit in memory.

The above evaluation does not preclude the $B$ arrays in the first or last contraction in a fusible list, $B_0$ and $B_{n-1}$ in the above example, from being too large to fit in memory. In these cases, fusion eliminates the data movement cost to read and write the intermediate tensor. On the other hand, fusion requires this contraction to share the available memory with other contractions, potentially increasing the cost incurred due to duplicated data movement.

Consider $B_0$ being large, with $qM$ ($0 \leq q \leq 1, M_{\beta}$ is the memory limit of level $\beta$) being the amount of memory required for fused execution of contractions producing tensors $I_2$ through $I_n$ in the running example. Comparing the two alternatives discussed above, fusing the first contraction is beneficial when the condition below is true:

$$\frac{2|I_1||I_0(c_1)|}{\sqrt{M_{\beta}}} + 2|I_1| \geq \frac{2|I_1||I_0(c_1)|}{\sqrt{(1-q)M_{\beta}}} \implies \frac{|I_0(c_1)|}{\sqrt{M_{\beta}}} + 1 \geq \frac{|I_0(c_1)|}{\sqrt{M_{\beta}} \sqrt{1-q}} \implies \frac{\sqrt{1-q}}{1-\sqrt{1-q}} \geq \frac{|I_0(c_1)|}{\sqrt{M_{\beta}}} \tag{5.4}$$

As shown in equation 5.1, $|I_0(c_1)|$ is small, usually much smaller than the memory available. The expression on the left hand side of last inequality above is greater than 1 for values of $q \leq 0.7$. Thus fusion is beneficial despite duplicated data movement for one of the arrays even when up to 70% of the memory is consumed by the remaining contractions. Note that this condition can be quickly verified at runtime once the problem size is known.

**Two-Level Memory Hierarchy**

Based on the discussion above, we assume that for a given sequence of tensor contractions to be fused, the non-intermediate arrays in all contractions, except for the first and last contraction, must fit in the memory. For simplicity, we further assume that all such arrays
Algorithm 1 Sample loop structure with allocation for $I_i$, $I_{i+1}$, and $I_{i+2}$

\[
\begin{align*}
S_1 &= I_i \cap I_{i+1} \cap I_{i+2} \\
S_2 &= I_i \cap I_{i+1}, \quad S_3 = I_{i+2} \\
S_4 &= I_i
\end{align*}
\]

\begin{algorithm}
\begin{align*}
\text{for } s_x \in S_1 & \text{ do} \\
& \{\text{Allocate } I_{i+1}[s_x]\} \\
\text{for } s_y \in S_2 - S_1 & \text{ do} \\
& \{\text{Allocate } I_i[s_y]\} \\
& \text{for } s_z \in S_4 - S_2 \text{ do} \\
& \{\text{Produce } I_i[s_z]\} \\
& \{\text{Update } I_{i+1}[s_y]\} \\
\text{end for} \\
\text{end for} \\
\text{for } s_w \in S_3 - S_1 & \text{ do} \\
& \{\text{Allocate } I_{i+2}[s_w]\} \\
& \{\text{Produce } I_{i+2}[s_w]\} \\
\text{end for} \\
\text{end for}
\end{align*}
\end{algorithm}

Together fit in the available memory. Based on this assumption, we will first present a solution to the problem of identifying a loop structure, with placement of data movement statements for all the intermediate arrays and one of the input arrays in the first contraction and the output array in the last contraction, assuming the remaining arrays fit in memory. This will later be extended to support the scenario in which the first or the last contraction require redundant data movement.

We simplify our presentation by assuming that all dimensions are similar in size. Consider a contraction list, referred to as a fusible list, in which all contractions are fusible according to the analysis presented in the previous subsection. We now determine the actual fusion chain and locations of the data allocation and movement statements. Let the fusible list be $I_0, \ldots, I_n$. For any three tensors $I_i$, $I_{i+1}$, and $I_{i+2}$, a sample loop structure is as shown in Algorithm 1, where the intersection operation denotes the intersection of indices of two tensors. The key observation here is that the loop nest producing $I_{i+1}$ must be enclosed by a memory allocation for $I_i$, since the loop nest consumes $I_i$. However, the allocation statements for tensors $I_i$ and $I_{i+2}$ need not enclose each other, since they do not have an immediate producer-consumer relationship. The data movement statements can be in arbitrary order as long as this condition is satisfied.

In this sample loop structure, the total memory cost is given by $\text{mem}(I_{i+1}) + \max(\text{mem}(I_i)$,
mem(I_{i+2})$. Since the unfused loop nest for each contraction, with one of the arrays fitting in memory, does not incur any redundant data movement, we consider only loop nests with no redundant data movement. The total memory cost can be determined through the recurrence relation shown below:

\[
\begin{align*}
  f(i, j) &= 0, \text{ if } j < i \\
  &= \min_{k=i}^{j} \frac{|I_k|}{|Common_{i,j}|} + \max(f(i, k - 1), f(k + 1, j)), \text{ otherwise}
\end{align*}
\]

\[Common_{i,j} = \bigcap_{k=i}^{j} I_k\]

The recurrence is evaluated as \(f(1, n)\) to compute the values of all \(f(i, j)\) through dynamic programming. \(Common_{i,j}\) denotes all the loops that are common to tensor contractions \(i\) through \(j\). The procedure to determine the loop structure first involves computing the memory required by arbitrary sub-chains of the “fusable list” memoized in a matrix. The function computes the data movement cost, in terms of volume, of executing a given chain of tensor contractions, ignoring the data movement for the first input and the last output. If the entire chain can be fused with the memory available, the data movement cost is evaluated to be zero. If not, various choices of splitting the chain into sub-chains are considered, and the choice that minimizes the data movement cost is determined recursively.

For each split the array at the split incurs the data movement cost equal to twice its size, to write it as output from the first sub-chain and read it back in the next sub-chain. Note that the algorithms shown only compute the memory and data movement costs. Determining the actual loop structure including the loop nesting to obtain the memory cost and splits to achieve the data movement cost can be determined by keeping track of the choices made while minimizing the cost function. The procedure is similar to that employed in the dynamic programming solution to matrix-chain multiplication [46].

When either the first or the last contraction involves large tensors necessitating duplicated data movement, we determine the optimal fusion structures for the fusible list excluding the contractions requiring such duplicated data movement. Let us consider the case with \(I_0, B_0,\) and \(I_1\) being too large to fit in memory. As discussed in Section 9.2.4,
the loops in the second contraction consuming the output of the first contraction should be enclosed by the memory allocation statement for the output tensor of the first contraction. In the running example, the allocation statement for $I_1$ must enclose the production of $I_2$. Positions in the loop structure determined for contractions 2 through $n$ that satisfy this requirement are determined. If the memory consumed by all live tensors at any of these positions is small enough to enable profitable fusion, as expressed by the condition 5.4, the first contraction is fused. This procedure is repeated for the last contraction in the list when it incurs duplicated data movement.

Multi-Level Memory Hierarchies

In the previous section, we presented the steps to determine the fusion structure that minimizes the data movement cost given a memory constraint. Here we show how the procedure naturally extends to handle multi-level memory hierarchies. We have considered fusion as a transformation that combines the loop corresponding to the producer and consumer such that the data movement cost for the associated intermediate tensor is eliminated. Effective fusion of a sequence of contractions for data movement between the $\beta\gamma$ levels implies that all contractions in the chain can be executed without intervening $\beta\gamma$ data transfer, with sufficient space in $\gamma$ to fit all necessary intermediates. A sub-chain can be fused at the $\gamma$ level if the memory required by it, as computed by this expression, is less than $M_\gamma$. Given the memory cost for each sub-chain at $\gamma$, the memory cost at the $\beta$ level is calculated using the following modified recurrence expression:

$$f(i, j) = \begin{cases} 0 & \text{if } j < i \\ \frac{|I_i| + |I_j|}{|I_i \cap I_j|} & \text{if } \text{memory}_{\gamma}(i, j) \leq M_\gamma \\ \min_{k=i}^{j} \frac{|I_k|}{|\text{Common}_{i,j}|} + \max(f(i, k - 1), f(k + 1, j)) & \text{otherwise} \end{cases}$$

$$\text{Common}_{i,j} = \bigcap_{k=i}^{j} I_k$$

This expression takes into account the fact that when a sub-chain is fused at the $\gamma$ level, only the first and last tensor in that sub-chain consume memory at the $\beta$ level. The total
Expression A (Small Input):
\[ C[h_1, h_2, h_3, h_4, h_5, h_6] = A[h_1, h_2, h_3, h_7] \times B[h_4, h_5, h_6, h_7] \]
\[ I[h_1, h_2, h_3, h_4, h_5, h_8] = B[h_8, h_6] \times C[h_1, h_2, h_3, h_4, h_5, h_6] \]

Expression B (Large Common Index):
\[ C[h_6, h_5, h_4, h_3, h_2, h_1] = A[h_7, h_5, h_4, h_3, h_2, h_1] \times B[h_7, h_6] \]
\[ I[h_8, h_2, h_1] = B[h_8, h_6, h_5, h_4, h_3] \times C[h_6, h_5, h_4, h_3, h_2, h_1] \]

Expression C (Large Input):
\[ C[h_8, h_7, h_6, h_5, h_4, h_3, h_2, h_1] = A[h_9, h_4, h_3, h_2, h_1] \times B[h_9, h_8, h_7, h_6, h_5] \]
\[ I[h_10, h_7, h_6, h_5, h_4, h_3, h_2, h_1] = B[h_10, h_8] \times C[h_8, h_7, h_6, h_5, h_4, h_3, h_2, h_1] \]

Expression D (Long Chain):
\[ M_1[h_1, h_2, h_3, h_4, h_5, h_7] = M_0[h_1, h_2, h_3, h_4, h_5, h_6] \times B_1[h_6, h_7] \]
\[ M_2[h_1, h_2, h_3, h_4, h_5, h_8] = M_1[h_1, h_2, h_3, h_4, h_5, h_7] \times B_2[h_5, h_8] \]
\[ M_3[h_1, h_2, h_3, h_4, h_5, h_8] = M_2[h_1, h_2, h_3, h_4, h_5, h_8] \times B_3[h_4, h_9] \]
\[ M_4[h_1, h_2, h_3, h_4, h_5, h_8] = M_3[h_1, h_2, h_3, h_4, h_5, h_8] \times B_4[h_3, h_10] \]
\[ M_5[h_1, h_11, h_10, h_9, h_7, h_8] = M_4[h_1, h_2, h_10, h_9, h_7, h_8] \times B_5[h_2, h_11] \]

Expression E (Fewer Dimensions and Large Dimension Size):
\[ C[h_1, h_2, h_3, h_4] = B[h_3, h_5] \times A[h_1, h_2, h_4, h_5] \]
\[ I[h_1, h_2, h_3, h_6] = B[h_6, h_4] \times C[h_1, h_2, h_3, h_4] \]

Figure 9.2: Tensor contraction expressions used as benchmarks

data movement cost for fusion in the presence of a multi-level memory hierarchy is computed using a modified version of procedure employed for a two-level memory hierarchy. If the memory required at the \( \beta \) level to execute a sub-chain is less than \( M_\beta \), the corresponding data movement cost is determined. The data movement cost for a sub-chain fusible at the \( \gamma \) level is given as the cost to transfer the first and last tensor in the chain between the \( \beta \gamma \) levels. We assume that despite fusion of some of the loops, the dimensions of the remaining loops are large enough for tiling all dimensions at the \( \gamma \) level. When a sub-chain involves a single contraction, due to the need for redundant data movement, its total data movement cost is computed as discussed in Section 9.2.3.

9.3 Experimental Evaluation

We evaluate our methods by executing tensor contractions on a system with an explicitly managed, multi-level, memory hierarchy. Particularly, the system consisted of the following three levels in the memory hierarchy: disk, global memory, and the local or device memory.
The global memory corresponds to the use of *global arrays* [126] on the cluster. The device memory is the memory on the GPU, which had a similar size as the local memory on each node. Since all processing is done on the GPU, processor cache is not an important component of the memory hierarchy. Within the GPU, the programmable cache or the *shared memory* reflects another level in the hierarchy, but did not turn out to be important for our target class of applications. In particular, the shared memory in the GPUs we used was too small to enable fusion. The impact of tiling on kernels such as matrix-matrix multiplication has been extensively evaluated in a variety of contexts [120, 142, 81, 45]. While the tile sizes are derived as discussed earlier, we focus on demonstrating the impact of our approach to fusion.

The specific configuration used was a cluster environment where each computing node has two quad-core Intel Xeon X5560 CPUs, running at 2.80 GHz. Every pair of nodes shares a Tesla 1070 GPU box, which means every node has two GPUs available as accelerators. For simplicity, in our experiments, we launch two processes on each node, so that every process can use one GPU for acceleration. Our analytical models assume that there is a 32 GB global memory and 1 GB local memory for the application's data structures. In all the experiments, the input data and the final output are stored on the disk. Input files are first loaded into the global memory, then moved to the local memory. Storage of output file follows the reverse order. Communication between disk and global memory is *collective*, whereas, the data movement between the global memory and the local memory is *one-sided* (each processor transfers data independently).

Our evaluation was based on five candidate sets of contractions, chosen to reflect the nature of the contractions in quantum chemistry. These benchmark expressions are listed in Figure 9.2. For each expression, we evaluate the benefit of fusion at both the global memory level and the disk level. Fusion at the global memory level implies that the intermediate data resides in the device memory, whereas fusion at the disk level implies that the intermediate data is kept in the global memory. As a baseline, we also created a non-fused version which refers to the case when all intermediate results are copied to the disk and need to be loaded for the next contraction. Our approach predicts that fusion is beneficial at the disk level
for all five expressions, while fusion at the global memory level is beneficial for expressions A, B, C, and D.

**Effectiveness on Different Expressions**

We compare the fused and non-fused versions in terms of computation cost, the transfer time between global and local memory levels, and disk I/O cost. In all the charts, LM-GM implies data transfer time between the local memory and the global memory. In Figures 9.3, 9.5, and 9.7, the left bar in each bar cluster denotes the time with fusion, and the right bar denotes the time without fusion. Computation time reported here includes time spent in floating point operations and data movement between local memory and the GPU device memory. Therefore, in some cases, the computation time reported is shorter in the version with fusion. The disk I/O times, which could be much higher than the other components, are shown in Figures 9.4, 9.6, and 9.8.

**Expression A – Fusion with small input matrices:** The Expression A involves two contractions, where all input matrices fit in local memory. Data transfers for the matrices C and I are the dominating factors in this computation, and fusion eliminates the overhead of moving C. The first cluster of bars in Figure 9.3 and Figure 9.4 show the benefit achieved
by fusion at the disk and global memory levels, respectively. The experiments were done with the dimension size 40, on four computing nodes. We can see that the fusion at the GM level reduces the sum of computation and the local memory transfer time by about 40%. On the other hand, fusion at disk to global memory level is playing an even more important role, since the execution is dominated by this part. From Figure 9.4, we can see that fusion results in speedup of about 2.4.

**Expression B – Fusion with large common index in the last contraction:** In this expression, the input $B_2$ in the second contraction is large, so tiling is done on the dimensions $h_5, h_4, h_3,$ and $h_2$. Though it introduced replicated accumulation for $I$, the overhead is still much smaller than the overhead of communication for $C$ and $B_2$. Fusion turns out to be beneficial as it removes communication for $C$. The performance with and without fusion at the two levels is shown in the second clusters from the left in Figures 9.3 and 9.4. We have experimented with the size of each dimension being 40. For the sum of computation and local memory transfer times, fusion resulted in a 45% reduction. At the disk level, the speedup by fusion is about 3.7, because the large amount of time spent in writing $C$ is avoided.

**Expression C – Fusion for large inputs in the first contraction:** As stated in Section 9.2.4, when the two input matrices in the first contraction are large enough to exceed the memory limit at a certain level of memory, fusion is still beneficial because the intermediate result is still the dominant part. Expression C represents this case, under the assumption that the matrix $C$ is very large and does not fit in the local memory. Therefore, tiling has to be performed on the *non-common index* of the input matrices $A$ and $B$.

We experimented with the dimension size being 16. From the third bar cluster in Figure 9.3, we can see that fusion saved 75% of the sum of computation and the local memory transfer times. Again, the benefit of fusion at the disk level is more significant. By fusion, the total disk I/O has a speedup of about 2.5 by removing matrix $C$, the dominating factor in the non-fused version.

**Expression D – Fusion for a list of five contractions:** The expression D was particularly chosen to test the fusion algorithm we had presented for a *fusion chain*. Every contraction here has one small input ($B_1, B_2, B_3, B_4,$ and $B_5$), which could reside in the
local memory. Fusion of all five contractions requires memory size $N^5$, where $N$ is the size of one dimension. Therefore, in the case where the size of the local memory is 1 GB, we cannot fuse all the five contractions at global memory level when tile size is 40 or more.

According to our fusion algorithm, in the fused version at the global memory level, if the tile size is smaller than 40, we fuse all the five contractions. However, when tile size is larger, we break the fusion chain and fuse the two shorter chains separately.

The performance for this expression with the dimension size 40 is shown as the fourth bar cluster in Figures 9.3 and 9.4. Moreover, as we have analyzed above, this expression exhibits interesting behavior as problem size is changed. Therefore, we analyzed the performance with varying dimension sizes, and the results are shown in Figures 9.5 and 9.6. For dimension size smaller than 40, the speedup of fusion is about 2~2.5, and drops to 1.4~1.7 when $N \geq 40$. At the disk level, fusion of five contractions is always feasible with problem sizes we have experimented with, resulting in a speedup of 4~5 due to fusion.

**Expression E – A case for which fusion does not have significant benefits:** Our
fusion condition predicts that fusion does not have significant benefit under certain conditions. The right most bar cluster in Figure 9.3 shows the performance with and without fusion at global memory level for the Expression E. We used a dimension size of 256. The execution time here is dominated by the computation time. Speedup by fusion is only about 1.02 ∼ 1.09, which is consistent with our prediction. This shows that our models are capable of correctly predicting when fusion is not be beneficial, or result in small improvements.

**Scalability Study**

We studied the benefit of our optimizations as the number of nodes in the cluster increases. We only present detailed results from the Expression A – the results from other cases are similar. Figures 9.7 and 9.8 show the performance on a relatively large problem size ($N = 48$), with different numbers of nodes. For the sum of computation and local memory transfer times, the speedup is about 1.8 ∼ 2.3. At disk level, the speedup due to fusion ranges between 2.3 and 2.5. It can be seen that the computation time decreases with increasing number of nodes, i.e., the computation is being parallelized effectively. However,
the communication time becomes relatively large, as data may be retrieved from different parts of the global memory. Thus, fusion becomes increasingly important at larger node counts. At the disk level, the I/O time is relatively independent of the number of nodes, resulting in relatively stable performance improvement from fusion.

9.4 Summary

Emerging high-end architectures are bringing new challenges for compilers and code generation systems. Particularly, deeper, explicitly controlled memory hierarchies are becoming common and need to be optimized for. We have considered tiling, loop fusion, and placement of data movement operations for tensor contractions, which is an important class of scientific applications, for systems with more than two levels in the memory hierarchy. We have developed practical techniques focusing on the dominant cost factors. Experimental evaluation has shown that loop fusion for multiple levels improves performance and our methods can correctly predict cases where loop fusion is not advantageous. In addition to being effective, we believe these low complexity approaches serve to reduce the search space and provide effective starting points for empirical search in the “neighborhood” of the candidate loop structures chosen.
CHAPTER 10

CONCLUSION

This chapter is a brief summary of the contribution of our existing research and future work, with the time line of our work plan.

10.1 Contribution

The usage of accelerators like GPUs and FPGA has introduced a new research field in parallel and high performance computing. However, writing algorithms on these devices require knowledge about the architecture and programming model. To provide an interface for programs in high level languages with computing power from these accelerators, we construct a transformation system. This system has the following contribution.

• We constructed a code transformation system GREENRIDE. In this system, the programmers simply need to specify the sequential reduction loop(s) with some additional information about the variables. Program analysis and code generation are used to map the applications to a GPU. We apply several additional optimizations when the code is being generated. The programmers can also provide other functions and annotation to help achieve better performance. In the experiments, we observed good speedup, while using very simple program analysis and code generation techniques.

• We developed a system GMAT-DM (Gpgpu from MATLAB for Data Mining applications), which is an automatic translation tool to convert data mining applications written in MATLAB to CUDA, for execution on GPUs. The main focus of this system is making efficient usage of memory, thus enabling high performance on GPU. By adapting code analysis
on high level source code and incorporating the CUDA code generator we developed previously, we were able to successfully speedup some data mining applications in MATLAB, and shows the potential of the system to deal with complicated MATLAB programs.

- We also provided parallelization of data mining applications on GPU clusters, by building AUTO-GC. This is based on previous work in our group, FREERIDE, which was a middleware for parallel data mining [86, 87]. Similar to GREENRIDE, AUTO-GC only requires the programmers to specify the sequential reduction loop(s) with some additional information about the variables. Good performance was also observed in the automatically generated programs.

- We developed a shared memory arrangement strategy based on integer programming. By analysing the kernel function and basic data information, the shared memory on GPUs is used more efficiently with this approach. It also suggests certain loop transformations which could optimize the usage of shared memory and further improve the performance.

- Auto-tuning is an effective way to provide good compile-time and run-time performance for applications with different data features on various systems. We developed an auto-tuning system for tensor contractions on different GPUs, which helps select the better algorithm and parameters to improve the performance.

- By analysing tensor contraction expressions, we provided a framework to optimize loop fusion for this class of applications on multi-level memory hierarchy. Our approach uses a simple evaluation method to determine loop fusion schemes, which reduces the overhead of data movement significantly.

10.2 Future Work

In future, we plan to extend our work in the following directions.

- The code generation system for data intensive applications with reduction patterns could be applied and optimized for other classes of applications, and be combined with some existing work, such as reduction operations with different locking mechanisms [82]. The integer programming model could also be used for other architectures, including future accelerators, and could also be used to determine shared memory configuration in a configurable system such as Fermi GPU.
• We would like to consider heterogeneous systems for the loop transformation approach with tensor contraction sequences. Currently, each memory layer has a single architecture. In some systems, different threads/processes might run on different types of memory, even at the same level. For example, in a heterogeneous system which consists of both GPU and multi-core CPU, a GPU process will need one more level of data movement, which is between host memory and GPU memory. The problem of loop transformation gets more complicated in this situation, which needs more effort to optimize.

• The auto-tuning framework will be extended to include more parameters, enabling better performance gain. For example, on T10 GPU, we use streaming inside a single task to pipeline computation and data movement. The selection of stream numbers is a potentially tunable parameter. Also, when “with data copy” and “without data copy” schemes are mixed, more complication will come out, which needs to be taken care of.
BIBLIOGRAPHY


[53] Boubacar DIOUF, Ozcan OZTURK, and Albert COHEN. Optimizing local memory allocation and assignment through a decoupled approach. 2009.


