INTEGRATED COMPILER OPTIMIZATIONS FOR TENSOR CONTRACTIONS

DISSERTATION

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

By

Xiaoyang Gao, B.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:
Dr. P. Sadayappan, Adviser
Dr. Gagan Agrawal
Dr. Atanas Rountev

Approved by

Adviser
Graduate Program in
Computer Science and
Engineering
ABSTRACT

This dissertation addresses several performance optimization issues in the context of the Tensor Contraction Engine (TCE), a domain-specific compiler to synthesize parallel, out-of-core programs for a class of scientific computations encountered in computational chemistry and physics. The domain of our focus is electronic structure calculations, where many computationally intensive components are expressible as a set of tensor contractions. These scientific applications are extremely compute-intensive and consume significant computer resources at national supercomputer centers. The manual development of high-performance parallel programs for them is usually very tedious and time consuming. The TCE system is targeted at reducing the burden on application scientists, by having them specify computations in a high-level form, from which efficient parallel programs are automatically synthesized.

The goal of this research is to develop an optimization framework to derive high-performance implementations for a set of given tensor contractions. In particular, the issues investigated include:

1. *Development of an efficient in-memory parallel algorithm for a tensor contraction:* A tensor contraction is essentially a generalized matrix multiplication involving multi-dimensional arrays. A novel parallel tensor contraction algorithm is developed by extending Cannon’s memory-efficient parallel matrix multiplication algorithm.
2. **Design of a performance-model driven framework for a parallel out-of-core tensor contraction:** For a parallel out-of-core tensor contraction, besides the in-core parallel algorithm used, several other factors can affect the overall performance, such as the nested-loop structure (permutation), tile size selection, disk I/O placement and the data partitioning pattern. The best choice here depends on the characteristics of the target machine and the input data. We develop performance models for different parallel out-of-core alternatives and use predicted information from these performance models to drive our optimization techniques.

3. **Design of an integrated optimization framework for a set of tensor contractions:** In order to improve locality and parallelism, many high-level program transformation techniques are incorporated in the TCE framework, such as loop fusion, fission, permutation, and tiling. Given a set of tensor contractions, the number of possible combinations of program transformations to consider is too large for exhaustive enumeration. A primary goal of our research is to: 1) model interactions between different transformations and assess their impact on the overall performance, 2) use a search-based strategy to explore the combined space of transformations, and provide efficient pruning methods to reduce the search space, 3) use performance-driven models to identify the best combination and generate high-performance code.

4. **Incorporation of domain-specific optimizations:** Tensors in the class of computations considered have a number of symmetry properties, which must be utilized in order to generate efficient code. However, tensor symmetry restricts the applicability and effectiveness of some compiler optimizations. We study the effects of different symmetry properties on loop transformations and code generation, and adapt the
optimization approaches to make them applicable to computations on tensors with symmetry properties.

We have implemented and evaluated most of the proposed algorithms and optimizing strategies. Experimental results show that they work effectively in practice.
This is dedicated to my parents
ACKNOWLEDGMENTS

I would like to express my gratitude to everyone who helped me during my graduate study. It is hard to name them all here, and thank them enough.

First and foremost, I want to thank Prof. P. Sadayappan, my thesis adviser. He has offered invaluable suggestions and advice to me about how to identify challenging problems and conduct research. His visionary thoughts and energetic working style have influenced me greatly as a researcher. I’m fortunate to have had the opportunity to work with and learn from him during the past several years.

Secondly, I want to express my sincere gratitude to Prof. Gerald Baumgartner and Dr. Chi-chung Lam. They give me valuable advice and shared their insights and thoughts with me. Many of the research findings in my work originated from discussions and collaboration with them. I also thank Dr. Agrawal and Dr. Rountev for spending time to serve on my committee, and providing valuable feedback to improve this dissertation.

I would also like to acknowledge the support of my colleagues on the TCE project, who helped me greatly during my Ph.D study. They are Albert Hartono, Sriram Krishnamoorthy, and Sandhya Krishnan. Special thanks to Qingda Lu for helping me on various errands when I was away from campus. You all made my study at the Ohio State University both interesting and very enjoyable.

Finally and most importantly, I thank my entire family for their love and support. Thank you Xiaogang! I couldn’t have accomplished all this work without your help.
VITA

Jan 01, 1974 .......................... Born - Chongqing, China

1997 ................................. B.S. Computer Science & Technology
Beijing University, China

2001-present ........................ Graduate Research Associate,
The Ohio State University.

PUBLICATIONS

Research Publications


2. Performance modeling and optimization of parallel out-of-core tensor contractions, Xiaoyang Gao, Swarup Kumar Sahoo, Qingda Lu, Chi-Chung Lam, J. Ramanujam, Gerald Baumgartner, P. Sadayappan. In proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in:

<table>
<thead>
<tr>
<th>Area</th>
<th>Professor</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Performance Computing</td>
<td>Prof. P. Sadayappan</td>
</tr>
<tr>
<td>Software Engineering</td>
<td>Prof. Atanas Rountev</td>
</tr>
<tr>
<td>Computational Theory</td>
<td>Prof. Ten-Hwang Steve Lai</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xvii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Tensor Contraction Engine</td>
<td>2</td>
</tr>
<tr>
<td>1.2 The Computational Context</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Related work</td>
<td>10</td>
</tr>
<tr>
<td>1.5 Dissertation Outline</td>
<td>12</td>
</tr>
<tr>
<td>2. Elaboration of Data Structures</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Operation Tree</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Loop Nesting Tree</td>
<td>15</td>
</tr>
</tbody>
</table>
5.6 Conclusions ....................................................... 88

6. Efficient Search for Loop Transformations to Optimize Data Movement ........ 92
   6.1 Preliminaries ...................................................... 94
   6.2 Fusion Structure and Disk I/O cost ................................ 96
      6.2.1 Maximally-Fused Fusion Structure ......................... 96
      6.2.2 Loop Nesting Tree ........................................... 97
      6.2.3 Fusion Structure and Disk I/O Cost ....................... 99
   6.3 Optimizing Fusion Structure for a Single Disk Access Operation .......... 105
      6.3.1 Candidate Location Sequence .............................. 105
      6.3.2 Weight of Loop Fission ..................................... 107
      6.3.3 Optimizing Loop Structure for a Single Disk Access Operation 110
   6.4 Optimizing Fusion Structure for a Set of Disk Access Operation ........... 113
      6.4.1 Algorithm ..................................................... 113
      6.4.2 Integrated Framework ....................................... 115
   6.5 Experiments ..................................................... 116
   6.6 Conclusion ...................................................... 118

7. Optimization for Tensor Contractions with Symmetry Properties ................. 120
   7.1 Spatial Symmetry .................................................. 120
   7.2 Permutation Symmetry ........................................... 125

8. Conclusion .......................................................... 136
   8.1 Contributions ..................................................... 136
   8.2 Future Work ....................................................... 138

Bibliography .......................................................... 140
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Per-processor costs incurred by the rectangular cannon algorithm’s for different logical forms of a 16-processor grid</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Configuration of the Itanium 2 cluster at OSC</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Performance of rectangular cannon’s algorithm for different processor grids</td>
<td>38</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance of rectangular cannon’s algorithm for different rotation index</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Performance results of unoptimized and optimized versions for 4-index transform on 4 processor</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Communication patterns used 4-index transform expression of unoptimized and optimized versions</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Arrays distribution constraint for different parallel algorithms</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Communication and Disk Access Volume for the Outside Communication pattern</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Communication and Disk Access Volume for the Inside/Rotation pattern</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Effective Communication and Disk Access Volume for the Inside Communication pattern with replication/accumulation parallelism</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Predicted and Empirical results</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Predicted performance results on 4 processors for ccsd and 4index-transform</td>
<td>66</td>
</tr>
<tr>
<td>4.7</td>
<td>When $B_d=10$MB/s, predicted best disk/communication overheads (in sec.)</td>
<td>67</td>
</tr>
</tbody>
</table>
5.1 Effectiveness of pruning of loop structures. ............................. 87

6.1 I/O time and total execution time for 4index-transform: maximally-fused loop structure versus optimized loop structure ........................................ 118

6.2 I/O time and total execution time for three implementations of 4index-transform ................................................................. 118
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Using fusion to reduce memory requirement</td>
<td>7</td>
</tr>
<tr>
<td>1.2 Using tiling and partitioning to improve data locality</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Decoupled TCE Synthesis System</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Operation tree and unfused loop structure for the four-index transform</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Representations involved in generation of a fused code structure</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Traditional Cannon’s algorithm</td>
<td>19</td>
</tr>
<tr>
<td>3.2 The variant communication pattern with array $B$ fixed</td>
<td>21</td>
</tr>
<tr>
<td>3.3 The variant communication pattern with array $A$ fixed</td>
<td>22</td>
</tr>
<tr>
<td>3.4 An instance of the generalized Cannon’s algorithm</td>
<td>23</td>
</tr>
<tr>
<td>3.5 The distribution method under the rectangular cannon’s algorithm with a $2 \times 3$ processor grid</td>
<td>26</td>
</tr>
<tr>
<td>3.6 The pseudo code for rectangular cannon’s algorithm for $R \times S$ processor grid</td>
<td>28</td>
</tr>
<tr>
<td>3.7 The init phase of the rectangular cannon’s algorithm with a $2 \times 3$ processor grid</td>
<td>29</td>
</tr>
<tr>
<td>3.8 Rotation phase of the rectangular cannon’s algorithm with a $2 \times 3$ processor grid</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Pseudo code of Inside Communication Method</td>
<td>45</td>
</tr>
</tbody>
</table>
7.8 Apply loop fusion for a symmetrization contraction . . . . . . . . . . . . . 130
7.9 Fusion structure with disk I/O for a symmetrization contraction . . . . . . 131
7.10 Element access pattern in iteration \( a = x \) of code1 in Figure 7.5(b) . . . . 132
7.11 Element access pattern in iteration \( a = x \) of code2 in Figure 7.5(b) . . . . 133
7.12 Apply loop fusion for a decompression contraction . . . . . . . . . . . . . 134
7.13 Fusion structure with disk I/O statements for a decompression contraction . 134
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SearchOptimalLoopStructure($t$: the root of a subtree)</td>
<td>74</td>
</tr>
<tr>
<td>2 EnumerateTopSubtrees($t$: the root of a sub-tree) returns $TreeSet$</td>
<td>89</td>
</tr>
<tr>
<td>3 Construction($P$)</td>
<td>90</td>
</tr>
<tr>
<td>4 Parenthesize($\text{int}$)</td>
<td>91</td>
</tr>
<tr>
<td>5 Weight($I,L,F$)</td>
<td>109</td>
</tr>
<tr>
<td>6 SingleDiskIO($D,F,M,\text{OptMemCost},\text{OptDiskCost}$)</td>
<td>112</td>
</tr>
<tr>
<td>7 MultiDiskIO($DS,F,M,\text{OptDiskCost}$)</td>
<td>115</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The development of effective performance-model driven program transformation strategies for optimizing compilers is a challenging problem. We face this problem in the context of a domain-specific compiler targeted at a class of computationally demanding applications in quantum chemistry. A synthesis system is being developed to automatically generate efficient parallel programs from a high-level mathematical specification of a computation expressed as tensor contraction expressions. As the optimization of such complex computations requires a number of program transformation strategies, it is difficult to determine the best combination of them. This becomes even more challenging on parallel and distributed systems due to the wide variety of system architectures.

The main goal of our research is to develop efficient and effective optimizing algorithms for a set of tensor contraction expressions. There are several factors that have impacts on the performance of a parallel out-of-core implementation. Our work especially focuses on the reduction of significant overheads involved in disk I/O and inter-processor communication. To address this problem, we first consider the optimization of the core computational component, a single tensor contraction. A memory-efficient parallel algorithm is provided for the in-memory computations, where communication cost is reduced by selecting a proper logical processor grid and data distribution schema. When the input/output tensors are too
large to fit in physical memory, out-of-core algorithms will be applied to improve locality. In such a context, a number of implementation parameters can affect the overhead cost considered, such as loop structure, tile size, and placements of disk I/O and communication statements. We provide an optimization system to analyze their interactions and effects on performance, and automatically optimize these parameters according to the target machine architecture and input data characteristics. The optimization procedure uses execution time as the evaluation metric; hence accurate performance models for both disk I/O costs and inter-processor communication costs are required. After that, we extend our optimization approach to more general and complicated situations, such as tensor contraction sequences and calculations with tensors with symmetry properties.

In the following section, we first provide an overview of the domain-specific compiler, followed by open issues not resolved previously. Then, we discuss our contribution and some related work. An outline of the thesis is given at the end of this section.

1.1 Tensor Contraction Engine

Our primary motivation arises from the domain of electronic structure calculations using ab initio quantum chemistry models such as coupled cluster methods [15], in which many computationally intensive components are expressible as a set of tensor contractions. This computational domain is extremely compute-intensive and consumes significant computer resources at national supercomputer centers. Currently, manual development of accurate quantum chemistry models in this domain is very tedious and may take an expert several months to years to develop and debug. In order to relieve chemists from the tedious programming tasks, we developed a synthesis tool to facilitate the rapid development
of high-performance parallel programs for the class of scientific computations described above.

Our program synthesis tool is called the tensor contraction engine (TCE), which takes the equations specified by chemists in a high-level form and acts like an optimizing compiler, automatically producing an effective parallel program tuned to the characteristics of the target machine and input data. A number of compile-time optimizations are being incorporated into TCE. These include algebraic transformations [30, 20] to minimize the number of algebraic operations, loop fusion and array contraction [29] for memory space minimization, tiling [26] and loop permutation for data locality optimization, computation and data distribution [11] for communication minimization, and etc.

The input of the TCE system is a tensor contraction expression over several input arrays. Instead of directly computing the result by the input arrays, the number of operations can often be significantly reduced by suitable choice of some intermediate arrays. However, these arrays that help in reducing the computational cost will cause a problem with the memory requirement. Although loop fusion can be effective in reducing the total size of intermediate arrays, they may still be too large to fit into memory after the fusion. Therefore, disk based data and out-of-core operations are required. To improve the performance of out-of-core operations, tiling strategy and loop permutation are applied to decrease the disk access cost. Moreover, for a parallel architecture, we need to consider distribution and parallelism strategies to reduce the communication cost. We provide an example later to illustrate each phase of the synthesis system, focusing on performance optimization issues that they address.
1.2 The Computational Context

In the class of computations we considered, the final result to be computed can be expressed using a collection of multi-dimensional summations of the product of several input arrays. As an example, we consider a transformation often used in quantum chemistry applications to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

\[
B(a, b, c, d) = \sum_{p,q,r,s} C_1(d, s) \times C_2(c, r) \times C_3(b, q) \times C_4(a, p) \times A(p, q, r, s) \quad (1.1)
\]

Here, \(A(p, q, r, s)\) is an input four-dimensional array, and \(B(a, b, c, d)\) is the output transformed array, both need to be placed on disk before and after calculation. The arrays \(C_1\) through \(C_4\) are called transformation matrices. The indices \(p, q, r,\) and \(s\) have the same range \(N\). Typical values for \(N\) range from 60 to 1300. Likewise, the indices \(a, b, c,\) and \(d\) have the same range \(V\). The number for \(V\) is usually between 50 and 1000. This expression is referred to as the 4-index transform in the rest of the dissertation.

The calculation of \(B\) is done in four steps to reduce the number of floating point operations from \(O(V^4N^4)\) in the initial formula (8 nested loops, for \(p, q, r, s, a, b, c,\) and \(d\)) to \(O(VN^4)\):

\[
B(a, b, c, d) = \sum_{s} C_1(d, s) \times \left( \sum_{r} C_2(c, r) \times \left( \sum_{q} C_3(b, q) \times \left( \sum_{p} C_4(a, p) \times A(p, q, r, s) \right) \right) \right)
\]

This operation-minimization approach results in the creation of three temporary intermediate arrays \(T1, T2,\) and \(T3:\)

\[
T1(a, q, r, s) = \sum_{p} C_4(a, p) \times A(p, q, r, s)
\]
\[ T_2(a, b, r, s) = \sum_q C_3(b, q) \times T_1(a, q, r, s) \]
\[ T_3(a, b, c, s) = \sum_r C_2(c, r) \times T_2(a, b, r, s) \]
\[ B(a, b, c, d) = \sum_s C_1(d, s) \times T_3(a, b, c, s) \]

Assuming that the available memory limit on the machine running this calculation is less than \( V^4 \) (which is 3TB for \( V = 800 \)), any of the logical arrays \( A, T_1, T_2, T_3, \) and \( B \) is too large to entirely fit within its memory. Therefore, if the computation is implemented as a succession of four independent steps, such as in Fig. 2.1(a), the intermediates \( T_1, T_2, \) and \( T_3 \) have to be written on disk after they are produced, and read from disk before they are used in the next step. Furthermore, since none of these array can be fully stored in memory, it may not be possible to read each element only once from disk. The amount of disk access volume could be much larger than the total volume of the data on disk. Suitable fusion of the common loops involved in the contractions that produce and consume an intermediate can reduce the size of the intermediate array, making it feasible to be retained it in memory.

For the example considered, the application of fusion is illustrated in Fig. 1.1(a). It can be seen that, by the use of loop fusion, \( T_1 \) can actually be reduced to a scalar, \( T_2 \) to a one-dimensional array and \( T_3 \) to a two-dimensional array. Then, all intermediate arrays can be completely stored in memory, without the need for any disk I/O for them.

Although loop fusion reduces the total memory requirement, input array \( A \) and output array \( B \) are still too large to be entirely loaded into memory. During the computation, they must be partially read from or write to disk through a memory buffer. The size of the buffer is in correspondence with the physical memory limit. The number of how many times an array is loaded from disk to memory is dependent on the local buffer size and disk I/O placement. Fig. 1.1(b) shows one possible way to insert corresponding disk I/O statements into the fused code specification in Fig. 1.1(b), where the buffer of \( A \) contains \( N \times N \)
words, and each element in array $A$ is read in memory for $V$ times. Here tiling is applied
to reduce the disk I/O cost. For example, in Fig. 1.2(a), loop $a$ is divided into two parts $aT$
and $aI$. Then, the amount of disk I/O required by array $A$ is decreased by $aI$ times. The
optimal tile size and disk I/O placement for each disk array would be chosen to minimize
the disk I/O cost while satisfying the memory constraints.

Since our ultimate aim is to generate code for parallel architectures, we must con-
sider distributed computing aspects, such as data partitioning and communication methods.
In the current system, we focus on one-dimensional or two-dimensional data distribution
patterns, and load-balanced computation decomposition. The computation of tensor con-
traction required complicated interactions between data elements. It is not easy to find
an effective method to partition data and computations to save communication costs. For
example, a natural approach to parallelize the program in Fig. 1.2(a) is to distribute the out-
nermost loop $aT$ to get the coarsest granularity of parallelism. Each processor then accesses
the partial $B$ and $C4$, and the complete $C2$, $C3$, $C4$ and $A$. In this method, array $C1,C2,$
$C3$ and $A$ must be replicated on all processors. In addition, redistribution may occur when
array $B$ and $C4$ are not distributed by dimension $a$ at the beginning. Both replication and
redistribution will incur a significant amount of inter-processor communications. The cor-
responding pseudo-code is shown in Fig. 1.2(b), when loop $aT''$ has a different range on
different processors.

For all optimization techniques discussed above, the operation minimization is inde-
pendent from others and can be applied separately. After the operation minimal form is
decided, other optimization techniques are used to seek high performance by transforming
the program structure. In this dissertation, we will provide a framework which takes the result of operation minimization (a set of tensor contraction expressions) as input, and decide the optimal program structure by applying a sequence of optimization techniques. As we discussed earlier, a lot of program parameters can affect the performance of a parallel program, such as loop structure, tile size selection, disk I/O placement, data partitioning and communication methods. These parameters are inter-related, and can’t be determined independently. However, due to the large amount of possible combinations, the simultaneous search for all parameters is not feasible either. The first prototype of TCE system adapts a decoupled optimization framework, which can be decomposed into several phases. At each phase, it determines some program parameters to optimize different performance metrics, while either assuming the optimal value of other parameters or using the result generated
practical and efficient system, but may generate sub-optimal solutions.

1.3 Contributions

The tensor contraction expressions involved in electronic structure calculations are extremely compute-intensive and need to operate on very large data sets. In this context, parallel and out-of-core implementations are required. For efficient execution of parallel out-of-core programs, two dominant overhead costs, inter-processor communication cost

Figure 1.2: Using tiling and partitioning to improve data locality

by previous stages. The overview of the decoupled system is shown in Fig. 1.3. This is a practical and efficient system, but may generate sub-optimal solutions.

The tensor contraction expressions involved in electronic structure calculations are extremely compute-intensive and need to operate on very large data sets. In this context, parallel and out-of-core implementations are required. For efficient execution of parallel out-of-core programs, two dominant overhead costs, inter-processor communication cost
Figure 1.3: Decoupled TCE Synthesis System

and local disk access cost, need to be reduced. Many program parameters affect these two costs, which requires the optimizer to apply different program transformation strategies. Here, the main challenge is how to select the best combination of these transformations to minimize the overall execution time. Because the core computations we are targeting are
domain-specific, we are not aware of any previous research work that can be applied here. We need to design novel optimization algorithms based on the special characteristics of this kind of computations. So far, our research includes the following:

- With Cannon’s memory efficient parallel matrix multiplication algorithm as a template, we develop a novel parallel algorithm for a single tensor contraction. The new algorithm retains the positive features of the old one, while overcoming its limitations, as it can deal with non-square processor grids.

- For a single parallel out-of-core tensor contraction, we provide a model driven optimization approach to reduce the total overheads. In this approach, we analyze the relationship between factors that affect performance, generate different solutions of the optimized code using accurate a performance model to estimate their execution times, and choose the optimal one with the minimal execution time.

- For a set of tensor contractions, intermediate arrays are generated. The program structure becomes more complicated, and so do the interactions between various transformation strategies. Loop fusion is introduced into the optimization system for the purpose of reducing the memory requirement of intermediate arrays. It also brings up imperfectly nested loop structures. We provide a framework, which can model the relationship between fusion, tiling and disk I/O placements, and efficiently prune the search space to find optimal solutions in reasonable time.

1.4 Related work

The issues arising in optimizing locality in the context of tensor contractions have been addressed in our earlier work, mainly focusing on minimizing memory-to-cache traffic.
This approach has been extended to address efficient disk-to-memory movement in [26], where a greedy approach seeking placement of disk read/write operations has been taken. Loop fusion is exploited as a powerful tool to reduce memory requirement in [29]. In [28], a set of candidate fusion structures with disk I/O placements is used as input and the tile size search space is explored. The search space is divided into feasible and infeasible solution spaces and their boundary is shown to contain the optimal solution. An algorithm has been developed to locate the boundary efficiently and a steepest ascent hill-climbing used to determine an efficient solution for the tile sizes.

There is some work in the area of software techniques for optimizing disk I/O, which includes parallel file systems, compile time [6, 7, 4, 22, 23, 21, 36, 37] and runtime libraries and optimizations [46, 10]. Bordawekar et al. [6, 4] discuss several compiler methods for optimizing out-of-core programs in High Performance Fortran. Our classification of communication into inside and outside methods is similar to the inside and outside communication strategies discussed in [5].

A special case of tensor contraction, the matrix multiplication problem has been extensively studied. Many parallel algorithms have been developed for matrix multiplications, such as SUMMA, DIMMA, SRUMMA, etc. Most of these research efforts focus on in memory implementation, with limited work on the design of parallel out-of-core linear algebra libraries [48, 47]. However, we are not aware of any work that addresses the detailed modeling of disk I/O and inter-processor communication costs, besides evaluation and optimization of the overall performance of parallel out-of-core computations.

Some recent work has explored the use of loop transformations for locality in nested loops, such as [14, 35, 50]. Nevertheless, a performance-model-driven approach for the integrated use of loop fusion and loop tiling for imperfectly nested loops has not been
presented in these works. Lim et al. develop a framework based on affine partitioning and blocking to reduce synchronization and improve data locality [1, 34, 44]. However issues related with locality enhancement, I/O placement and optimization, and automatic tile size selection for imperfectly nested loops are not considered in their work.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows. In the next chapter, some data structures and key terms used in our system are explained. In Chapter 3, we introduce the novel parallel algorithm designed for a single tensor contraction. Chapter 4 presents our performance-model driven optimization algorithm for a single parallel out-of-core tensor contraction. The integrated system to optimize a set of tensor contractions is introduced in Chapter 5 and Chapter 6. Chapter 7 discusses some domain-specific optimizations for computations arising in chemistry and physics, and how to combine them with general compiler optimizations. In Chapter 8 we summarize the contributions of this dissertation and discuss follow on work.
CHAPTER 2

ELABORATION OF DATA STRUCTURES

As we have seen in the previous chapter, once a tensor contraction expression is in an operation minimal form, we need to find an optimal evaluation for it. A lot of program transformation strategies cooperate in our optimization system. Different representations of the program structure are used in different optimizing phases to facilitate the analysis of effects of program transformations. In this chapter, we will describe data structures and related notations used in the later discussions.

2.1 Operation Tree

As an example, consider the tensor contraction expression 1.1, which can be computed by the following sequence of contractions:

\[ T_1(a, q, r, s) = \sum_p C_4(a, p) \times A(p, q, r, s) \]
\[ T_2(a, b, r, s) = \sum_q C_3(b, q) \times T_1(a, q, r, s) \]
\[ T_3(a, b, c, s) = \sum_r C_2(c, r) \times T_2(a, b, r, s) \]
\[ B(a, b, c, d) = \sum_s C_1(d, s) \times T_3(a, b, c, s) \]

The sequence of contractions can be directly translated into a collection of perfectly nested loops, as shown in Fig. 2.1(b). It can also be represented by an operation tree
in Fig. 2.1(a). In an operation tree, leaves correspond to the input arrays and the root to the output array. The intermediate nodes and output node are also called **contraction nodes**, which are produced by the tensor contraction of their immediate children. The edges in the operation tree represent the **producer-consumer** relationship between the different contractions. Henceforth, the term **intermediate node** will be used to refer to both the intermediate array produced in the corresponding contraction node of the operation tree, and the contraction that produces it. The reference shall be clear from the context.

An operation tree represents the calculation dependence of the contraction sequence. By adding more information to nodes, it can correspond to other kinds of relationships. For example, in the next chapter, we will put the distribution method in each node, and model the redistribution processes during the computation by the expanded tree.
2.2 Loop Nesting Tree

As we said before, the directly implementation of an operation tree is to create a perfectly nested loop for each contraction node. Actually, when the physical memory has enough space to hold all intermediate arrays, this is the best solution to achieve optimal performance. Otherwise, we employ loop fusion to merge the same loops between contractions with a producer-consumer relationship, then reduce the size of corresponding intermediate arrays. The merged index(dimension) in an intermediate array is said to be fused. The loop nesting tree is introduced to show the loop fusion structure of an imperfectly nested loop structure.

Given a choice of fusion, an intermediate node not fused with its parent divides the operation tree into two parts, both of which can be evaluated independently. Such an unfused intermediate node, is said to be a cut-point in the operation tree. A connected operation tree without any interior cut-points is called a fused sub-tree. The divided operation tree for the four-index transform corresponding to $T_1$ being a cut-point is shown in Fig. 2.2(a). The cut-point divides the operation tree into two fused sub-trees, one of which produces $T_1$, and the other which consumes it.

The loop nesting tree (LNT) represents a loop structure mapping to a fused sub-tree. Each node in a LNT is labeled by the indices of a set of fully permutable loops appearing together at some level in the imperfectly nested loop structure which resulted from applying loop fusion to the contraction computations in the sub-tree. The leaves represent the innermost loops, while the root represents the outermost loops. Fig. 2.2(b) shows two possible LNT’s for the two fused subtrees in Fig. 2.2(a), respectively. The corresponding code fragment is presented in Fig. 2.2(c).
T1 = SUM(A*C4)

A  C4

T2 = SUM(T1*C3)

T1  C3

T3 = SUM(T2*C2)

C2  T2

B = SUM(T3*C1)

C1  T3

(a) Divided operation trees

a, r, q, s, t (T1)

a, b

r, s

c, d, s (B)

q (T2)

c (T3)

(b) Loop nesting trees

loopNest1:
for a, r, q, s, p
\[ t_{1,a,q,r,s} += A_{p,q,r,s} \times C_{4a,p} \]

loopNest2:
for a, b
for r, s
for q
\[ t_{2,r,s} += t_{1,a,q,r,s} \times C_{3b,q} \]
for c
\[ t_{3,c,s} += t_{2,r,s} \times C_{2c,r} \]
for c, d, s
\[ B_{a,b,c,d} += t_{3,c,s} \times C_{1d,s} \]

(c) Corresponding loop structure

Figure 2.2: Representations involved in generation of a fused code structure.
CHAPTER 3

GENERALIZATION OF CANNON’S ALGORITHM

Since primitive tensor contractions are essentially generalized multi-dimensional matrix multiplications, we choose to use the memory efficient Cannon’s algorithm [8] as the primary template. However, Cannon’s algorithm is designed for two-dimensional arrays, and is restricted to a square distribution of arrays among processors. In this chapter, we present a generalization of Cannon’s algorithm, which can be applied to multi-dimensional arrays, and not requiring the number of processors to be a perfect square.

3.1 Cannon: Memory Efficient Matrix Multiplication Algorithm

The traditional Cannon’s algorithm is designed for parallel matrix multiplication. A logical view of the \( P \) processors as a two-dimensional \( \sqrt{P} \times \sqrt{P} \) grid is used (\( P \) is a perfect square number). Each matrix is block distributed along the two processor dimensions. As will be clear later on, the logical view of the processor grid does not impose any restriction on the actual physical interconnection topology of the processors, since an empirical characterization of the cost of redistribution between different distributions is performed on the target system. To simplify the problem, we assume that the time required for a processor node to send a message of \( m \) words to another processor node is modeled as \( ts + tw \times m \), where \( ts \) is the message setup time and \( tw \) is the data transmission time per
word. As an example, the matrix multiplication is:

\[ C(m, n) = \sum_k A(m, k) \times B(k, n) \]  

(3.1)

Before the computation starts, all matrices should be fully distributed among the square processor grid. Let \( C_{i,j} \) denote the sub-block of \( C \) specified by \( C(\text{range}(i, L_m, \sqrt{P}), \text{range}(j, L_n, \sqrt{P})) \), where \( L_m \) is the size of \( m \) dimension, and \( \text{range}(x, L, \sqrt{P}) \) is the range \( i \times L/\sqrt{P} \) to \((i + 1) \times L/\sqrt{P} - 1\). The sub-block \( A_{i,j} \) and \( B_{i,j} \) are defined in the same fashion. \( C_{i,j}, A_{i,j} \) and \( B_{i,j} \) are mapped onto processor \( P_{i,j} \), the processor in the \( i^{th} \) row and \( j^{th} \) column, \( 0 \leq i, j < \sqrt{P} \), of the two-dimensional processor grid. The algorithm executes in two phases: \textit{init} and \textit{rotation}. The \textit{init phase} essentially skews the matrices \( A \) and \( B \) to align them appropriately. In this phase, sub-blocks of \( A_{i,j} \) are shifted left circularly by \( i \) positions along the row of processor grid; sub-blocks of \( B_{i,j} \) are shifted up circularly by \( j \) positions along the column of processor grid. Thus, the processor \( P_{i,j} \) can receive sub-blocks \( A_{i,(j+i)\mod \sqrt{P}} \) and \( B_{(i+j)\mod \sqrt{P},j} \), and perform the first sub-matrix multiplication on them. The \textit{rotation phase} consists a sequence of \( \sqrt{P} - 1 \) rotate-compute operations. During each step, sub-blocks \( A \) (\( B \)) are shifted left (up) circularly by one processor and each processor multiplies the newly acquired sub-blocks of \( A \) and \( B \) and adds the result to the sub-block of \( C \) being maintained. The pseudo code of Cannon’s algorithm is presented in Fig.(3.1).

Since there is no replicated data in the algorithm, each processor requires \( A.size/P + B.size/P + C.size/P \) words of memory to store sub-blocks of \( A, B, \) and \( C \). The total communication time required for the initial alignment and consecutive rotations is:

\[ \text{CommCost} = 2ts \times \sqrt{P} + (A.size + B.size) \times tw/\sqrt{P} \]
Predistribution:
Distributed $A, B$ and $C$ on the logical processor grid

Init phase:
$P$ = The number of processors
$A_{i,j}$ = The sub-block on array $A$ on processor $P_{i,j}$
$B_{i,j}$ = The sub-block on array $B$ on processor $P_{i,j}$
Left circularly shift sub-block $A_{i,j}$ by amount $i$
Up circularly shift sub-block $B_{i,j}$ by amount $j$
for $m = 0$ to $\frac{L_n}{\sqrt{P}} - 1$, $n = 0$ to $\frac{L_m}{\sqrt{P}} - 1$
  for $k = 1$ to $\frac{L_k}{\sqrt{P}} - 1$
    $C(m,n) + = A(m,k) \times B(k,n)$

Rotation phase:
for $Rotation = 1$ to $\sqrt{P} - 1$
  Left circularly shift sub-block $A_{i,j}$ by 1
  Up circularly shift sub-block $B_{i,j}$ by 1
  for $m = 0$ to $\frac{L_n}{\sqrt{P}} - 1$, $n = 0$ to $\frac{L_m}{\sqrt{P}} - 1$
    for $k = 1$ to $\frac{L_k}{\sqrt{P}} - 1$
      $C(m,n) + = A(m,k) \times B(k,n)$

Figure 3.1: Traditional Cannon’s algorithm

3.2 Generalizing Cannon’s Algorithm for Multi-dimensional Tensor Contraction

A multi-dimensional tensor contraction formula can be expressed as a generalized two-dimensional matrix multiplication $C(M, N) = \sum_K A(M, K) \times B(K, N)$, where $M$, $N$, and $K$ represent index sets instead of single indices. Considering the following tensor contraction,

$$C(a, b, c, d) = \sum_{f,l} A(a, b, f, l) \times B(f, l, c, d)$$  \hspace{1cm} (3.2)

It is characterized by the index sets $M = \{a, b\}$, $N = \{c, d\}$, and $K = \{f, l\}$. Here, we call $M$ and $N$ non-summation indexsets, and $K$ the summation indexset.
The generalization of Cannon’s algorithm for multi-dimensional arrays is illustrated in the following. The \( P \) processors form the same two-dimensional logical grid as before. For a multi-dimensional array, we use a pair of indices to denote its partitioning or distribution on the processor grid. Consider array \( C(a, b, c, d) \), the index pair \( (a, c) \) specifies that the first \((a)\) and the third \((c)\) dimensions of \( C \) are distributed along the first and second processor dimensions respectively; and that the second \((b)\) and fourth \((d)\) dimensions of \( C \) are not distributed. Thus, a processor \( P_{i,j} \) will be assigned a portion of \( C \) specified by \( C(\text{range}(i, L_a, \sqrt{P}), 0 : L_b - 1, \text{range}(j, L_c, \sqrt{P}), 0 : L_d - 1) \). This sub-block of \( C \) is referred as \( C_{i:a,j:c} \). Following the same fashion, array \( A \) and array \( B \) are distributed by a pair of indices, one from the non-summation set and the other from the summation set. For example, \( A \) is distributed by index \( (a, f) \) and \( B \) is distributed by index \( (f, c) \). Then, the sub-blocks of \( A \) and \( B \) on processor \( P_{i,j} \) are denoted as \( A_{i:a,j:f} \) and \( B_{i:f,j:c} \).

To describe the distribution and communication methods used in our algorithm, we define the following terms:

**Definition 3.2.1** A triplet \( \{m, n, k\} \) formed by one index from each index set \( M, N, \) and \( K \) defines a distribution \( \langle m, n \rangle \) for the result array \( C \), and distributions \( \langle m, k \rangle \) and \( \langle k, n \rangle \) for the input arrays \( A \) and \( B \), respectively. Therefore, we call it distribution triple.

**Definition 3.2.2** One of the three indices in the distribution triple can be chosen as the rotation index, along which the processor communication takes place. The other two indices in the distribution triple are called fixed indices. The same fixed index in different arrays is distributed among the same processor dimension.
Due to the symmetry of the problem, any of the three indices \( \{m, n, k\} \) in the distribution triple can be chosen as the rotation index, so that it is possible to keep any one of the arrays in a fixed distribution and communicate ("rotate") the other two arrays. In the traditional Cannon’s algorithm, the summation index \( k \) is the rotation index. Sub-blocks of \( A \) and \( B \) are rotated among processors, then each processor holds a different sub-block of \( A \) and \( B \) after each rotation step. Array \( C \) is fixed from the init phase. The same sub-block of \( C \) is used in every computation on each processor. There are two other variants of Cannon’s algorithm that keep either array \( A \) or array \( B \) in a fixed distribution. The communication patterns of these two variants are slightly different than the traditional version. For example, in the Expression (3.2), assuming the distribution triple is \( \{a, c, f\} \), the communication patterns of two variants of Cannon’s algorithm are shown in Figure 3.2 and Figure 3.3.
(a) Init phase  

(b) Rotation phase

Figure 3.3: The variant communication pattern with array $A$ fixed

In the first pattern, rotation index is $a$; array $B$ is fixed among processors. Array $A$, $B$ and $C$ are distributed by $\langle a, f \rangle$, $\langle c, f \rangle$ and $\langle c, a \rangle$, respectively. In the init phase, array $A$ and $C$ are skewed and aligned properly. During the rotation phase, sub-blocks of $A$ ($C$) are shifted up (left) circularly. For each step, a processor multiplies the newly acquired sub-block of $A$ and the maintained sub-block of $B$, then adds the result to the new sub-block of $C$. The pseudo code of the first pattern is presented in Fig. 3.4.

In the second pattern, rotation index is $c$; array $A$ is fixed among processors. Array $A$, $B$ and $C$ are distributed by $\langle a, f \rangle$, $\langle c, f \rangle$ and $\langle a, c \rangle$, respectively. In the init phase, array $B$ and $C$ are skewed and aligned properly. During the rotation phase, sub-blocks of $B$ ($C$) are shifted up (left) circularly. For each step, a processor multiplies the maintained sub-block
Predistribution:
Distributed $A$ on index $a$ and $f$
Distributed $B$ on index $c$ and $f$
Distributed $C$ on index $c$ and $a$

Init phase:
$P$ = # of processors
$A_{i,a;j,f}$ = The sub-block on array $A$ on processor $P_{i,j}$
$C_{i;c,j;a}$ = The sub-block on array $B$ on processor $P_{i,j}$
Up circularly shift sub-blocks $A_{i,a;j,f}$ in each column by amount $i$
Left circularly shift sub-blocks $C_{i;c,j;a}$ in each row by amount $j$
for $a = 0$ to $\frac{L_a}{\sqrt{P}} - 1$, $b = 0$ to $\frac{L_b}{\sqrt{P}} - 1$, $c = 1$ to $\frac{L_c}{\sqrt{P}} - 1$, $d = 1$ to $\frac{L_d}{\sqrt{P}} - 1$
for $f = 0$ to $\frac{L_f}{\sqrt{P}} - 1$, $l = 0$ to $\frac{L_l}{\sqrt{P}} - 1$
\[ C(a,b,c,d)+ = A(a,b,f,l) \times B(f,l,c,d) \]

Rotation phase:
for Rotation = 1 to $\sqrt{P} - 1$
Up circularly shift sub-block $A_{i;a,j,f}$ by 1
Left circularly shift sub-block $C_{i;c,j;a}$ by 1
for $a = 0$ to $\frac{L_a}{\sqrt{P}} - 1$, $b = 0$ to $\frac{L_b}{\sqrt{P}} - 1$, $c = 1$ to $\frac{L_c}{\sqrt{P}} - 1$, $d = 1$ to $\frac{L_d}{\sqrt{P}} - 1$
for $f = 0$ to $\frac{L_f}{\sqrt{P}} - 1$, $l = 0$ to $\frac{L_l}{\sqrt{P}} - 1$
\[ C(a,b,c,d)+ = A(a,b,f,l) \times B(f,l,c,d) \]

Figure 3.4: An instance of the generalized Cannon’s algorithm

of $A$ and the newly acquired sub-block of $B$, then adds the result to the new sub-block of $C$.

Therefore, the number of distinct communication patterns within the generalized Cannon’s algorithm framework is given by $3 \times |M| \times |N| \times |K|$, where $|M|$, $|N|$, and $|K|$ are defined as the number of indices in the index set $M$, $N$, and $K$, respectively. The communication cost involved in the tensor contraction depend on the size of arrays, the number of processors, the inter-processor network configurations $ts$ and $tw$, and the selection of rotation index $ri$: 

23
\[
M_{\text{CommCost}}(A, B, C, P, ts, tw, ri) = \begin{cases} 
2ts \times \sqrt{P} + (B.\text{size} + C.\text{size}) \times tw/\sqrt{P} & \text{if } ri \in M \\
2ts \times \sqrt{P} + (A.\text{size} + C.\text{size}) \times tw/\sqrt{P} & \text{if } ri \in N \\
2ts \times \sqrt{P} + (A.\text{size} + B.\text{size}) \times tw/\sqrt{P} & \text{if } ri \in K 
\end{cases}
\] (3.3)

In addition to the communication involved in the init and rotation phases, array redistribution may be necessary before or after the algorithm. For instance, when evaluating Expression (6.1), the arrays \(A(a, b, f, l)\) and \(B(f, l, c, d)\) have initial distributions \(\langle a, b \rangle\) and \(\langle f, c \rangle\) respectively. \(A\) would have, for example, to be re-distributed from \(\langle a, b \rangle\) to \(\langle a, f \rangle\) for the generalized Cannon’s algorithm to be possible. Since the initial distribution \(\langle f, c \rangle\) of \(B\) is the same as the distribution required to perform the Cannon’s rotations, no redistribution is necessary for array \(B\). However, if we want \(C\) to have the distribution \(\langle a, d \rangle\), we would either re-distribute \(B\) from \(\langle f, c \rangle\) to \(\langle f, d \rangle\) before the algorithm, or re-distribute \(C\) from \(\langle a, c \rangle\) to \(\langle a, d \rangle\) after the algorithm. The selection of distribution triple will affect the redistribution cost. The redistribution cost would be estimated differently for the different cases. In the general cases, the redistribution can be modeled as an all-to-all scatter operation, and the optimal cost to re-distribute array \(A\) on \(P\) processors is:

\[
\text{RedistCost}(A, P) = ts \times P + tw \times \text{A.size}/P
\] (3.4)

Henceforth, we will use the formula to calculate the redistribution cost.

### 3.3 Generalizing Cannon’s Algorithm for Non-square Processors Grid

The advantage of Cannon’s algorithm lies in its memory and communication efficiency. But, it is only suitable for a square grid of processors and square matrices. In this section, we introduce an extension of Cannon’s algorithm for dealing with non-square processors grid and non-square arrays, which will be called as the **rectangular cannon’s algorithm**.
This algorithm can be applied on any shape of rectangular processors grid. Therefore, given a set of processors and three operand arrays, we can choose the optimal shape of the two-dimensional processors grid which is specified for the size of input arrays. Here, we introduce a new term for the logical processor grid.

**Definition 3.3.1** The $P$ processors form a logical rectangular grid. A pair of numbers $(x, y)$ represents the number of processors assigned to the $x$ dimension and $y$ dimension in the grid, respectively. We call it the **logical form** of the processor grid.

Since the logic processor grid is not always square, we add a pair of integers to describe the array distribution schema, such as $(< m, k >, < x, y >)$, where $< m, k >$ are distribution indices, $< x, y >$ is the logical form of processor grid. Two distribution schemas $d$ and $d'$ are same as each other, if and only if $< m, k > = < m', k' >$ and $< r, s > = < r', s' >$.

In order to simplify the description, we use the matrix multiplication in Expression (6.1) as the example, and assume that the rotation index is always $k$. Nevertheless, with slightly modification, the same algorithm can be used in other cases. Assuming the logical form of processor grid is $(R, S)$. The least common multiple of $R$ and $S$ is denoted as $LCM(R, S)$. Furthermore, we have:

\[ lr = \frac{LCM(R, S)}{R} \]

\[ ls = \frac{LCM(R, S)}{S} \]

To apply the Cannon’s algorithm, the sub-blocks of $A$ and $B$ on the same processor must be matched in the summation dimension. It means that, at step $t$, if a processor wants to perform computation on sub-block $A_{i,j}$ and $B_{p,q}$, then the range of $j$ and $p$ would be same.
Based on this rule, we divide the common dimension ($k$ in this example) of $A$ and $B$ into $\text{LCM}(R, S)$ parts. Therefore, array $A$ is divided into $R \times \text{LCM}(R, S)$ sub-blocks, array $B$ is divided into $\text{LCM}(R, S) \times S$ sub-blocks, and array $C$ is divided into $R \times S$ sub-blocks. Each processor in the grid holds $ls$ sub-blocks of $A$, $lr$ sub-blocks of $B$, and one sub-block of $C$. Each step operates on a sub-block and not on the entire data local to a processor.

Let $SA_{i,j}$ denote the sub-block of $A$ specified by $A(\text{range}(i, L_m, R), \text{range}(k, L_k, \text{LCM}(R, S)))$, and $SB_{i,j}$ denote the sub-block of $B$ specified by $B(\text{range}(i, L_k, \text{LCM}(R, S)), \text{range}(j, L_n, S))$, and $C_{i,j}$ denote the sub-block of $C$ specified by $C(\text{range}(i, L_m, R), \text{range}(j, N_n, S))$, where $0 \leq i < R$, $0 \leq j < S$, and $0 \leq k < \text{LCM}(R, S)$. For example, on a $2 \times 3$ processor grid, the distribution of array $A$, $B$, and $C$ are shown in Figure 3.5.

As we discussed in the previous section, the algorithm executes in two phases. At the init phase, the block $SA_{i,j}$ is circularly shifted $lr \times i$ positions left, and the block $SB_{i,j}$ is circularly shifted $ls \times j$ positions up. The rotation phase consists of $\text{LCM}(R, S) - 1$ steps.
At step $t$, $1 \leq t < LCM(R, S)$, the processor $P_{i,j}$ multiplies the sub-block $S_{A_{i,k1}}$ with $S_{B_{k2,j}}$, and adds the result to the sub-block $C_{i,j}$, where $k1$ and $k2$ are:

$$k1 = (lr \times i + ls \times j + t) \mod LCM(R, S)$$

$$k2 = (ls \times j + lr \times i + t) \mod LCM(R, S)$$

If $t + 1$ is an integer multiple of $ls$, each processor circularly shifts all sub-blocks of $A$ on it to its left neighbor in the same row. If $t + 1$ is an integer multiple of $lr$, each processor circularly shifts all sub-blocks of $B$ on it to its upper neighbor in the same column. The pseudo code of the rectangular cannon’s algorithm is shown in Fig. 3.6. For a $\langle R, S \rangle$ logical form, the communication cost involved in the algorithm can be estimated by the following formula:

$$Rec_{CommCost}(A,B,R,S) = \begin{cases} 
S \times (ts + \frac{A_{\text{size}} \times tw}{R \times S}) + R \times (ts + \frac{B_{\text{size}} \times tw}{R \times S}) & \text{if } R > 1, S > 1 \\
(S - 1) \times (ts + \frac{A_{\text{size}} \times tw}{R \times S}) & \text{if } R = 1 \\
(R - 1) \times (ts + \frac{B_{\text{size}} \times tw}{R \times S}) & \text{if } S = 1
\end{cases}
$$

(3.5)

Note that, the function above only works when $ri$ is belonging to indexset $K$, but, it can easily be modified to calculate the communication cost for $ri$ in $M$ or $N$.

For further explanation, we use the following example. Assuming the logical form of processor grid is $\langle 2, 3 \rangle$, at first, the distribution of array $A$, $B$ and $C$ are shown in Figure 3.5. In the init phase, array $A$ and array $B$ are skewed for alignment. The distributions after alignment are shown in Figure 3.7, where the shadowed blocks are the sub-matrix used in the first computation. The rotation phase, which includes 5 steps, is illustrated in Figure 3.8. It shows in shadow the sub-blocks of array $A$ and $B$ accessed in every step. At the end of the second and fourth steps, all sub-blocks of array $A$ are circularly shifted 2 position left. At the end of the third step, all sub-blocks of array $B$ are circularly shifted 3 position up.
Definition:

\[ R = \text{Number of processors in the } x\text{-axis} \]
\[ S = \text{Number of processors in the } y\text{-axis} \]
\[ \text{LCM} = \text{The least common multiple of } R \text{ and } S \]
\[ lr = \text{LCM}/R \]
\[ ls = \text{LCM}/S \]

Predistribution:

Divide \( A \) into \( R \times \text{LCM} \) sub-blocks and distribute them on \( R \times S \) processor grid
Divide \( B \) into \( \text{LCM} \times S \) sub-blocks and distribute them on \( R \times S \) processor grid
Divide \( C \) into \( R \times S \) sub-blocks and distribute them on \( R \times S \) processor grid

Init phase:

\[ SA_{i,j} = \text{The sub-block on array } A \text{ on row } i \text{ and column } j \]
\[ SB_{i,j} = \text{The sub-block on array } B \text{ on row } i \text{ and column } j \]
Left circularly shift sub-block \( SA_{i,j} \) by amount \( lr \times i \)
Up circularly shift sub-block \( SB_{i,j} \) by amount \( ls \times j \)
\[ A_{m,k} = \text{The first sub-block of } A \text{ on the current processor} \]
\[ B_{k,n} = \text{The first sub-block of } B \text{ on the current processor} \]
\[ C_{m,n} = \text{The sub-block of } C \text{ on the current processor} \]
for \( m = 0 \) to \( \frac{Lm}{R} - 1 \), \( n = 0 \) to \( \frac{Ln}{S} - 1 \)
[ for \( k = 0 \) to \( \frac{Lk}{\text{LCM}} - 1 \)
[ \[ C(m,n)+ = A(m,k) \times B(k,n) \]

Rotation phase:

for \( Rotation = 1 \) to \( \text{LCM} - 1 \)
[ \[ k1 = Rotation \mod ls \]
[ \[ k2 = Rotation \mod lr \]
If \( (k1 = 0) \) then
[ \[ \text{Circularly shift all sub-blocks of } A \text{ one processor left} \]
if \( (k2 = 0) \) then
[ \[ \text{Circularly shift all sub-blocks of } B \text{ one processor up} \]
\[ A_{m,k} = \text{The } (k1+1)\text{th sub-block of } A \text{ on the current processor} \]
\[ B_{k,n} = \text{The } (k2+1)\text{th sub-block of } B \text{ on the current processor} \]
for \( m = 0 \) to \( \frac{Lm}{R} - 1 \), \( n = 0 \) to \( \frac{Ln}{S} - 1 \)
[ for \( k = 0 \) to \( \frac{Lk}{\text{LCM}} - 1 \)
[ \[ C(m,n)+ = A(m,k) \times B(k,n) \]

Figure 3.6: The pseudo code for rectangular cannon’s algorithm for \( R \times S \) processor grid
In the traditional Cannon’s algorithm, each processor needs $\sqrt{P}$ sub-blocks of both array $A$ and $B$ from remote processors. In addition, the rotation phase always includes $\sqrt{P} - 1$ steps. But, in the rectangular cannon’s algorithm, the number of rotation steps is given by the logical form of the processor grid. The number of remote blocks of $A$ that are needed by one processor corresponds to the number of processors assigned to $y$ dimension, and that of $B$ correspond to the other dimension. Table 3.1 illustrates the number of rotation steps and communication volume during the rotation phase for each possible logical form with sixteen processors.

It can be seen that different logical forms have different communications for each of the components. When the number of processors assigned to $x$ dimension is increased, so is the communication incurred by array $B$. On the contrary, the communication incurred by array $A$ get decreased. The relative sizes of the arrays $A$ and $B$ determines the optimal distribution pattern. When one array is much larger than the other, the cost can be reduced by skewing the distribution to reduce the number of remote blocks accessed for that array.
### Figure 3.8: Rotation phase of the rectangular cannon’s algorithm with a 2 × 3 processor grid
<table>
<thead>
<tr>
<th>Logical form</th>
<th># of rotation steps</th>
<th>Communication volume per processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 16)</td>
<td>15</td>
<td>( \frac{15A}{16} ) 0</td>
</tr>
<tr>
<td>(2, 8)</td>
<td>7</td>
<td>( \frac{A}{2} ) B/8</td>
</tr>
<tr>
<td>(4, 4)</td>
<td>3</td>
<td>( \frac{A}{4} ) B/4</td>
</tr>
<tr>
<td>(8, 2)</td>
<td>7</td>
<td>( \frac{A}{8} ) B/2</td>
</tr>
<tr>
<td>(16, 1)</td>
<td>15</td>
<td>0 ( \frac{B}{16} )</td>
</tr>
</tbody>
</table>

Table 3.1: Per-processor costs incurred by the rectangular cannon algorithm’s for different logical forms of a 16-processor grid

Notice that, if there is only one processor in the \( x \) dimension, then each processor needs all sub-blocks of array \( A \) and only local sub-blocks of array \( B \) for computation. It is same as the broadcast algorithm with column-major distribution. On the other side, if there is only one processor in the \( y \) dimension, it is same as the broadcast algorithm with row-major distribution. By choosing the proper logical processor grid, we can use the framework of the rectangular cannon’s algorithm to model other one-dimensional or two-dimensional parallel matrix multiplication algorithms.

### 3.4 Communication Minimization for A Sequence of Tensor Contractions

The generalized Cannon’s algorithm introduces a new challenge in how to choose the best distribution pattern in parallel execution to minimize the communication cost. Actually, it includes three sub-problems, the distribution triple, the rotation index and the logical form of processor grid.
For a sequence of tensor contraction expressions, each intermediate array is produced by an contraction and consumed by another contraction. Array redistribution may be required to feed data from one contraction to another, because the way data is distributed in one contraction may not be the most appropriate distribution for the next contraction for algorithmic or efficient reasons. We would consider the communication incurred not only during contraction computations, but also between contraction computations. Thus, the distribution pattern of each individual contraction can not be determined separately. We need to find the global distribution pattern to minimize the total communication cost. In this section, a dynamical programming algorithm is provided to determine the optimal distribution pattern for a sequence of contractions. We begin with the estimation of communication cost for different versions of rectangular Cannon’s algorithm.

The distribution pattern $E$ of a contraction $T$ includes three fields: $\langle m, n, k \rangle$, $ri$ and $\langle r, s \rangle$, which present the distribution triple, rotation index and logical processor form respectively ($P = r \times s$ and $ri \in \{m, n, k\}$). Assuming the calculated contraction is $C = \sum A \times B$, the communication cost during computation is estimated differently from the variances of rotation index:

$$E.cost1 = \begin{cases} 
    s \times (ts + tw \times R_1/P) + r \times (ts + tw \times R_2/P) & \text{if } r, s > 1 \\
    (s - 1) \times (ts + tw \times R_1/P) + (r - 1) \times (ts + tw \times R_2/P) & \text{if } r \vee s = 1 
\end{cases}$$

where

$$R_1 = A.size, \quad R_2 = B.size \quad \text{if } ri \in K$$

$$R_1 = C.size, \quad R_2 = B.size \quad \text{if } ri \in M$$

$$R_1 = A.size, \quad R_2 = C.size \quad \text{if } ri \in N$$
The required distribution of array $A$ during $T$ is denoted as $A.dist$, where $A.dist = \{< m, k >, < r, s >\}$. The previous distribution of $A$ before $T$ is $A.dist' = \{< m', k' >, < r', s' >\}$. If $A$ is an input array, $A.dist'$ is its initial distribution. Otherwise, $A.dist'$ is the output distribution of the contraction producing $A$. If $A.dist'$ is not same as $A.dist$, we need to redistribute array $A$ before computation. The same happens to array $B$. The redistribution cost for $E$ is estimated by:

$$ E.cost2 = \begin{cases} 
2ts \times P + tw \times (A.size + B.size)/P & \text{if } A.dist \neq A.dist', B.dist \neq B.dist' \\
ts \times P + tw \times A.size/P & \text{if } A.dist \neq A.dist', B.dist = B.dist' \\
ts \times P + tw \times B.size/P & \text{if } A.dist = A.dist', B.dist \neq B.dist' \\
0 & \text{if } A.dist = A.dist', B.dist = B.dist' 
\end{cases} $$

The total communication cost to calculate the contraction $T$ under distribution pattern $E$, with input distributions $A.dist'$ and $B.dist'$ is the summation of $E.cost1$ and $E.cost2$. The output distribution $C.dist$ produced under $E$ should be $C.dist = \{< m, n >, < r, s >\}$. For a distribution pattern $E$, if the output distribution is fixed, the first two indices in the distribution triple and the logical form of processor grid of $E$ are decided. Thus, a distribution pattern can also be identified by its output distribution, distributed summation index, and rotation index, such as $E = (C.dist, k, ri)$. In another word, the communication cost of $E$ depends on the output distribution, input distributions, and the distributed summation index and rotation index used in the parallel execution, which can be denoted as $Cost(C.dist, A.dist', B.dist', k, ri)$.

Given a sequence of tensor contractions, first, an operation tree is created. The dynamical programming used to determine the optimal distribution pattern for each contraction node traverses the tree in a bottom-up fashion. For each node $T$, we calculate the set of all feasible output distributions of the relevant contraction, denoted as $T.output$, and find the minimal communication cost to achieve each distribution in $T.output$ as following,
• If $T$ is an input node (leaf of operation tree) with initial distribution $d_0$. It doesn’t contain any computation, the output distribution is same as initial distribution.

\[
T.\text{output} = \{d_0\}
\]

\[
\text{Cost}(T, d_0) = 0
\]

• If $T$ is an intermediate node with two children left and right, then the cost of a particular distribution is calculated as the least cost of its children and subsequently cost to compute it from its children. The set of output distributions includes all combinations of a pair of non-summation indices and a pair of integers whose product is the number of processors. For each output distribution $d_i$ in $T.\text{output}$, the optimal solution that minimizes the cost of $d_i$ is recorded in $\text{opt}(T, d_i)$, which consisting of two links $d_1$ and $d_2$ to the input distributions, summation index $k$, and rotation index $ri$.

\[
T.\text{output} = \{(m,n), (i_1,i_2) \mid \forall m \in M, \forall n \in N, \forall i_1 \times i_2 = P\}
\]

\[
\forall d_i \in T.\text{output}, \text{Cost}(T, d_i) = \min\{\text{Cost}(\text{left}, d_1) + \text{Cost}(\text{right}, d_2) + \text{Cost}(d_i, d_1, d_2, k, ri), \forall d_1 \in \text{left}.\text{output}, \forall d_2 \in \text{right}.\text{output}, \forall k \in K, \forall ri \in \{m,n,k\}\}
\]

\[
\forall d_i \in T.\text{output}, \text{opt}(T, d_i) = (d_1, d_2), \quad d_1 \text{ and } d_2 \text{ used to compute } \text{Cost}(T, d_i)
\]

where

\[
M \equiv \text{Non-summation indexset in left}
\]

\[
N \equiv \text{Non-summation indexset in right}
\]

\[
K \equiv \text{Summation indexset}
\]

\[
P \equiv \text{The number of processors}
\]

• If $T$ is the root of the operation tree with final distribution $d'$, we first calculate the cost of all possible output distributions of $T$ in the same way as in an intermediate node. After that, outputs with a distribution not same as $d'$ will be redistributed, where redistribution cost can be estimated by Expression 3.4. The minimal total cost
for the whole tree is \( \text{finalCost}(T, d') \), which consists of the cost to compute \( T \) and redistribute \( T \).

\[
\text{finalCost}(T, d') = \min \{ \text{Cost}(T, d_i) + \text{Redist}(T, d', d_i), \forall d_i \in T.\text{output} \}
\]

where \( \text{Redist}(T, d', d_i) \) should be zero if \( d' \) is same as \( d_i \).

The output distribution \( d_i \) that minimize the total cost is selected. The distribution pattern leading to \( d_i \) is the optimal distribution pattern for the contraction \( T \). The optimal distribution pattern for other nodes can be obtained by tracing back \( \text{opt}(T, d_i) \) in a top-down manner.

### 3.5 Experiments

We evaluated the generalized cannon’s algorithm on the OSC Itanium-2 cluster whose configuration is shown in Table 3.5. All the experiment programs were compiled with the Intel Itanium Fortran Compiler for Linux.

For the generalized cannon’s algorithm, there are three parameters that can be adjusted according to the size of input data to optimize performance in different cases, the \textit{logical form of processor grid}, \textit{rotation index} and \textit{distribution triple}. The distribution triple has no significant impact on the a single contraction, so, at first, we use a simple matrix multiplication (where the distribution triple is fixed) to show how the logical form and rotation index can affect the performance of the generalized cannon’s algorithm. After that, another example in the chemistry domain is adopted to illustrate the communication minimization algorithm discussed in Section 3.4.
1. **Non-square Matrix Multiplication**:

\[ C(m, n) = \sum_{k} A(m, k) \times B(k, n) \]

In the first experiment, we set \( m = 4800, n = 120, k = 4800 \), then generate different versions of the rectangular cannon’s algorithm using different logical processor grids. To examine the effect of logical form, we fix the rotation index as \( k \) in all implementations. Table 3.5 illustrates the performance of different implementations on 8 processors. From the experiment result, we can see that the communication cost will vary tremendously with different forms of processor grid. By selecting the appropriate logical form, the total execution time of the algorithm can be reduced almost 4 times, then linear speedup is obtained.

The purpose of the second experiment is to evaluate the impact of rotation index, in which \( m = 120, n = 120, k = 400000 \) and \( P = 8 \). Table 3.5 shows the experimental results of rectangular cannon’s algorithms with different rotation index, with the logical form as \( < 1, 8 > \). Obviously, using \( m \) as the rotation index can results in a significant performance improvement over others.

The following observations can be made from the experimental results above.

- Given a non-square matrix multiplication, the relative size of dimension \( m, n \) and \( k \) would determine the optimal logical form and rotation index.

- When the size of involved arrays differ considerably, we would keep the largest array fixed, and rotate smaller arrays among processors by selecting the proper logical form or rotation index.

- Based on the logical form and rotation index used by the parallel algorithm, the communication overhead may become great, leading to an actual decrease in the
overall performance. Thus, our algorithm, which can provide a flexibility in setting these parameters, can outperform the traditional cannon’s algorithm significantly in non-square matrix multiplications.

2. **4-index Transform Expression**: A sequence of tensor contractions introduced in Section 1.2.

\[
B(a, b, c, d) = \sum_s C1(d, s) \times \left( \sum_r C2(c, r) \times \left( \sum_q C3(b, q) \times \left( \sum_p C4(a, p) \times A(p, q, r, s) \right) \right) \right)
\]

In this experiment, we use the dynamical programming algorithm discussed the Section 3.4 to choose the optimal communication pattern for each contraction in the expression. We compare our approach with the baseline implementation in which the distribution triple of each contraction is formed by the first index from set \(M, N, \) and \(K, \) respectively, and a fixed \(\sqrt{P} \times \sqrt{P} \) logical processor form is required throughout the computation. We assume that the distribution of the input and output arrays are unconstrained, and can be chosen by the algorithm to optimize communication time.

The performance results for the unoptimized and optimized version are compared in Table 3.5. In most cases, the optimal version can achieve 60% improvement over the unoptimized version. Table 3.6 shows the communication pattern chosen for each contraction in the experiment \(N = 96, V = 64.\) A first look reveals that the number of redistributions is reduced by effective choice of distribution triple. In addition, the processor forms used in the optimized version help to keep the larger array fixed and rotate the smaller one, so as to reduce the communication occurred during the computation.
Table 3.2: Configuration of the Itanium 2 cluster at OSC

<table>
<thead>
<tr>
<th>Node</th>
<th>Memory</th>
<th>OS</th>
<th>Compilers</th>
<th>Interconnect</th>
<th>Latency</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual 900MHz</td>
<td>4GB</td>
<td>Linux</td>
<td>g77, ifc</td>
<td>Myrinet 2000</td>
<td>17.8 μs</td>
<td>220 MB/s</td>
</tr>
<tr>
<td>Itanium 2</td>
<td></td>
<td>2.4.21smp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Performance of rectangular cannon’s algorithm for different processor grids

<table>
<thead>
<tr>
<th>A.size</th>
<th>B.size</th>
<th>C.size</th>
<th>Logical Form of Processor Grid</th>
<th>Comm. Cost</th>
<th>Total Cost</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>184.32 MB</td>
<td>4.608 MB</td>
<td>4.608 MB</td>
<td>(1,8)</td>
<td>0.78 s</td>
<td>1.05 s</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2,4)</td>
<td>0.396 s</td>
<td>0.659 s</td>
<td>3.278</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(4,2)</td>
<td>0.212 s</td>
<td>0.485 s</td>
<td>4.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(8,1)</td>
<td>0.028 s</td>
<td>0.29 s</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Table 3.4: Performance of rectangular cannon’s algorithm for different rotation index

<table>
<thead>
<tr>
<th>A.size</th>
<th>B.size</th>
<th>C.size</th>
<th>Rotation Index</th>
<th>Comm. Cost</th>
<th>Total Cost</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>384 MB</td>
<td>384 MB</td>
<td>0.1152 MB</td>
<td>k</td>
<td>1.92 s</td>
<td>2.87 s</td>
<td>2.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td>0.2 s</td>
<td>1.18 s</td>
<td>5.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n</td>
<td>1.92 s</td>
<td>3.16 s</td>
<td>2.22</td>
</tr>
</tbody>
</table>

Table 3.5: Performance results of unoptimized and optimized versions for 4-index transform on 4 processor

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>Unoptimized Version</th>
<th>Optimized Version</th>
<th>Factor of Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>40</td>
<td>1.71s</td>
<td>2.383s</td>
<td>0.75s</td>
</tr>
<tr>
<td>80</td>
<td>60</td>
<td>2.944s</td>
<td>4.3s</td>
<td>1.451s</td>
</tr>
<tr>
<td>96</td>
<td>64</td>
<td>4.86s</td>
<td>7.283s</td>
<td>1.633s</td>
</tr>
<tr>
<td>100</td>
<td>72</td>
<td>6.3s</td>
<td>9.73s</td>
<td>2.21s</td>
</tr>
<tr>
<td>Contraction Node</td>
<td>Unoptimized pattern</td>
<td>Optimized pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Input Distrib Triple</td>
<td>Logical Rotation Index</td>
<td>Input Distrib Triple</td>
<td>Logical Grid</td>
</tr>
<tr>
<td>$T1+ = A * C4$</td>
<td>$-$</td>
<td>$s, a, p$</td>
<td>(2, 2)</td>
<td>$p$</td>
</tr>
<tr>
<td>$T2+ = T1 * C3$</td>
<td>$T1$</td>
<td>$a, b, p$</td>
<td>(2, 2)</td>
<td>$q$</td>
</tr>
<tr>
<td>$T3+ = T2 * C2$</td>
<td>$T2$</td>
<td>$b, c, r$</td>
<td>(2, 2)</td>
<td>$r$</td>
</tr>
<tr>
<td>$B+ = T3 * C1$</td>
<td>$T3$</td>
<td>$c, d, s$</td>
<td>(2, 2)</td>
<td>$s$</td>
</tr>
</tbody>
</table>

Table 3.6: Communication patterns used 4-index transform expression of unoptimized and optimized versions
CHAPTER 4

PERFORMANCE MODELING AND OPTIMIZATION OF PARALLEL OUT-OF-CORE SINGLE CONTRACTION

We have described the memory efficient parallel algorithm for a single tensor contraction and the communication optimization algorithm for a sequence of contractions in the previous chapter. These algorithms are used in the case that all tensors can fit into memory entirely. While tensors are too large to fit into memory, data movement between memory and disk, as well as communication among processors, becomes critical on the overall performance. To perform global optimization of a parallel out-of-core tensor contraction, we developed a framework to model both disk I/O cost and inter-processor communication cost and analyze the trade-off between them.

In a parallel out-of-core tensor contraction, many implementation factors affect these two dominant overhead costs, including the communication pattern, the parallel algorithm, data partition, loop permutation, disk I/O placements and tile size selection. These factors are inter-related and cannot be determined independently. The number of possible combinations is exceedingly large and searching them all is impractical. In this chapter, we provide an approach, which can model the relationship between different factors and efficiently prune the search space to find a good combined solution in reasonable time.
This chapter is organized as follows. In the next section, we introduce the main concepts and describe the parallel system supported by the algorithm. Section 4.2 discusses the impact of loop order and the placement of disk I/O statements. Algorithms used with the outside communication pattern and inside communication pattern (defined in Section 4.1.1) are discussed in Section 4.3 and Section 4.4, respectively. Section 6.5 presents results from the application of the new algorithm to an example abstracted from NWCHEM [45]. Conclusions are provided in Section 4.6.

4.1 Preliminaries

As we discussed before, given a tensor contraction, it can be written as

\[ C(I, J) = A(I, K) \times B(J, K), \]  

(4.1)

where \( I, J \) and \( K \) are index sets considered as macro-indices. Consider a distributed-memory computer with \( P \) processors in which every processor has limited local memory and unlimited local disk. If a processor needs data from the local disk of other processors, the required data will be first read by the owner processor and then communicated to the requesting processor. The inter-processor network bandwidth is denoted as \( B_c \), and the local disk to memory bandwidth is \( B_d \). Arrays A, B and C are either evenly distributed or fully replicated among all processors. An index set \( dist \) is used to represent the distribution pattern of an array. For example, if array A is distributed by index \( i \) and \( j \), then \( A.dist = \langle i, j \rangle \). If A is replicated on all processors, then \( A.dist = \langle \rangle \).

The original size of an array is denoted as \( array.size \). In a parallel algorithm, the size of an array required for local computation is denoted as \( array.localsize \). If all required data can fit into memory, there is no disk I/O involved. Otherwise the array whose \( array.localsize \) larger than one-third of local memory will be put on disk.
Because data sets are very large, we assume that both communication cost and disk I/O cost are dominated by the volume of data movement, not the number of data movements. The communication cost and disk I/O cost, can be calculated simply by dividing the transferred volume with the transferring bandwidth.

Three parallel matrix multiplication algorithms, *rotation*, *replication*, and *accumulation* are used. They will be discussed and compared in Section 4.1.2. The choice of the parallel algorithm decides the communication pattern. In the rotation algorithm, computation is executed in several steps. Each processor *circular-shifts* its local data with neighbors between computations. In the replication algorithm, one operand is *broadcasted* to all processors. In the accumulation algorithm, the partial result of the entire target array is *reduced* among all processors. These communication patterns can be implemented by corresponding communication routines. Communication routines on out-of-core data will be carried out in several steps and results in extra disk access cost.

### 4.1.1 Communication Methods

When there is insufficient memory to hold all the remote data for the next computation to be performed locally on a processor, we can use one of two broad approaches to handling the out-of-core communication: 1) first perform disk-to-disk transfer so that all remote data needed by a processor for its portion of the computation is first moved to its local disk, or 2) intersperse disk I/O with computation on in-core sub-arrays. We call the first method as the outside communication method, and the latter as the inside communication method. In the outside communication method, communication and local computation are separated from each other. All remote data for the next computation is fetched before the start of the computation and stored on disk. It may cause redundant disk access, but achieves
minimal communication cost. With the **inside communication method**, communication and local computation are interleaved together. When one block of data is copied into memory, the owner processor performs computations on it, and passes it to other processors requiring it. When other processors receive remote data, they perform computations on it, and discard it without writing it to disk. This approach incurs extra communication cost, but minimizes disk access cost. Examples of these two communication methods are shown in Figure 4.1 and Figure 4.2. The choice of the communication method introduces a trade-off between communication cost and disk access cost. Thus, when available local memory is large enough to hold all the remote data, we can directly select the outside communication method.

### 4.1.2 Parallel Algorithms and Distribution Indices

Many approaches have been proposed for implementing parallel matrix multiplication. In this framework, three simple and common parallel algorithms are considered as the basis for an individual tensor contraction: rotation, replication and accumulation. Implementation details of these parallel algorithms are discussed next.

1. **Rotation**: We use the generalized multi-dimensional Cannon’s algorithm as the primary template, which is described in Section 3.2. In this approach, a logical view of the $P$ processors as a two-dimensional $\sqrt{P} \times \sqrt{P}$ grid is used. Each array is two-dimensional cyclic-block distributed along the two processor dimensions.

2. **Replication**: In this scheme, each processor locally holds one full input array and a strip of the other two arrays. In order to achieve good performance, we always replicate the smaller operand. Without loss of generality, we assume the size of array A is less than the size of B. Thus, to use the replication parallel algorithm, array A
is replicated on all processors, $A.dist = \emptyset$, and arrays B and C are distributed by the same dimensions belonging to the index set J, $B.dist = C.dist = \langle j \rangle, j \in J$. Replication communication can be modeled as an all-to-all broadcast communication operation, whose communication cost is a topology-dependent function. To simplify the problem, we assume that the interconnection network is completely connected in the rest of the chapter. Thus, we use the expression $\text{Replicate}(S) = (S.size) / B_c$ to calculate the replication time.

3. **Accumulation:** In order to apply the accumulation parallel algorithm, two operands are distributed by the same summation indices, $A.dist = B.dist = \langle k \rangle, k \in K$, and the target array is replicated on all processors, $C.dist = \emptyset$. Every processor executes a partial matrix multiplication and accumulates the result at last. The accumulation can be modeled as an all-to-all reduction communication operation, whose communication cost depend on the inter-processor topology. In the completely-connected network, the all-to-all reduction cost is calculated by $\text{Reduce}(S) = (S.size \times \log(P)) / B_c$.

If the distribution of the input or output arrays are not suitable for a specific parallel algorithm, we need to rearrange the data before or after execution. The redistribution procedure is separated from the computation procedure.

The pseudo code of these three parallel algorithms using the inside communication method are shown in Figure 4.1. The corresponding pseudo code for the outside communication method are shown in Figure 4.2. Arrays A, B and C are out-of-core arrays that are distributed using a block-cyclic distribution among $P$ processors in order to render the Collective disk I/O operations load-balanced. Collective disk I/O operations operate on global tiles, which consist of a set of local tiles. The corresponding local disk I/O operation is indicated under the collective disk I/O operation. In the pseudo code, the order of It, Jt
Figure 4.1: Pseudo code of Inside Communication Method

and Kt loops is not determined, all the disk I/O statements and message passing statements are placed inside them. However, after the loop structure is defined, these data movement statements will be inserted at the appropriate places in the actual program.
4.1.3 The Overall Problem Definition

There are many methods to implement a parallel out-of-core tensor contraction. The performance of a method changes according to the hardware execution environment and the tensors’ shape and size. A specific implementation method can be defined by setting following parameters

- the communication method (outside/inside);
• the parallel algorithm and distributed patterns (rotation/replication/accumulation);
• the order of the loops and disk I/O placements; and
• the tile sizes for each dimension

Given a tensor contraction and specific machine parameters, including the number of processors, the amount of physical available memory for every processor, the inter-processor network bandwidth, and the local disk to memory bandwidth, we now address the problem of finding the optimal implementation method, such that the total cost of inter-processor communication and disk I/O is reduced.

For the input and output arrays, the algorithm can be used in either of these modes:

• the distribution of the input and output arrays are unconstrained, and can be chosen by the algorithm to optimize the communication cost; or

• the input and output arrays have a constrained distribution on to the processors in some pre-specified pattern.

The parallel execution can be decoupled into three stages:

1. redistribute the input arrays;

2. compute the tensor contraction expression in parallel; and

3. redistribute the output array.

The total overhead cost for a specific parallel algorithm, which is denoted as $PA$, can be calculated by:

$$\text{Overhead}(PA) = \text{Redist}(A, A.dist1, A.dist2)$$
Table 4.1: Arrays distribution constraint for different parallel algorithms

\[
\begin{align*}
\text{PA} & \quad \text{Distribution Constraints.} \\
\text{Rotation} & \quad A.\text{dist} = \langle i, k \rangle, B.\text{dist} = \langle j, k \rangle, C.\text{dist} = \langle i, j \rangle \\
\text{Replication A} & \quad B.\text{dist} = C.\text{dist} = \langle j \rangle \\
\text{Replication B} & \quad A.\text{dist} = C.\text{dist} = \langle i \rangle \\
\text{Accumulation} & \quad A.\text{dist} = B.\text{dist} = \langle k \rangle 
\end{align*}
\]

where A.dist1 and B.dist1 are the initial distribution of the input arrays A and B, and C.dist2 is the expected distribution of the output array C. A.dist2, and B.dist2 are operand distribution patterns required for PA. C.dist1 is the target distribution pattern generated by PA. A.dist2, B.dist2 and C.dist1 must be compatible with each other by the distribution constraints of PA. The distribution constraints for different parallel algorithms is shown in Table 4.1.

If the initial distribution is the same as the final distribution, data re-arrangement is not necessary, and the redistribution cost is zero. Otherwise, the redistribution cost is the sum of the communication cost and the disk I/O cost, which depend on the redistribution scheme and machine specific inter-processor topology.

In a multi-dimensional tensor contraction expression, many distribution methods can be applied in a specific parallel algorithm. The choice of the distribution method will affect the redistribution cost in stages one and three. However, the overhead of parallel execution in stage two can be calculated independently of the distribution method. Thus,
in the following sections, we present an algorithm to determine all parameters, except for distribution method, which can minimize the overhead cost in stage two. The choice of the distribution method that allows for optimizing the redistribution cost will be discussed later.

4.2 Loop Order and Disk I/O Placements

In this section, we will concentrate on the loop order and the placements of disk I/O statements. We will consider only the order of tiling loops since different orders of the intra-tile loops will not significantly affect the execution time.

Consider the tensor contraction expression given in Expression (6.1). After tiling, the loops $I_t, J_t, K_t$ will be the tiling loops as shown in Figures 4.1 and 4.2. Note that $I_t, J_t, K_t$ are not single indices, but index sets, i.e., they each consist of several loop indices or be empty. Orderings of these tiling loops will depend upon the order of the placement of the disk I/O statements. There are three disk I/O statements corresponding to disk arrays $A, B,$ and $C$. We need to consider six cases for the relative order of these statements as: $ABC, ACB, BAC, BCA, CAB, CBA$.

Consider the case where disk statements are in the order $ABC$ as shown in Figure 4.3. The three disk statements will divide the tiling loops into four parts: $D_1, D_2, D_3,$ and $D_4$. Each of these parts will contain some loops from each of the index sets $I_t, J_t, K_t$. Let $D_i$ contain index sets $I_{t_i}, J_{t_i}, K_{t_i}$ as shown in Figure 4.3(a). Considering the loops in part $D_1$, we note that if $J_{t_1}$ is non-empty, then disk I/O for $A$ will be unnecessarily repeated several times. So $J_{t_1}$ should be moved to part $D_2$ to reduce the total volume of disk access for $A$ without increasing the size of local buffers for $A, B$ and $C$. After putting $J_{t_1}$ to part $D_2$, we can merge index sets $J_{t_1}$ and $J_{t_2}$ together, and re-name the new index set as $J_{t_1}$.

49
Considering the loops in part $D_2$, we note that if $I_{t2}$ is non-empty, then disk I/O for $B$ will be unnecessarily repeated several times. So $I_{t2}$ should be moved to part $D_3$ to reduce the total volume of disk access for $B$ without increasing the size of local buffers. Further, $K_{t2}$ would be empty or moved to part $D_1$ to reduce the memory requirement for the local buffer of $A$ without increasing the volume of disk access for $A$, $B$, and $C$. Similarly, considering the loops in part $D_3$, we note that $K_{t3}$ should be empty or be moved to part $D_4$ to reduce the total volume of disk access for $C$ and that the loops in $J_{t3}$ should be empty or moved to part $D_2$ to reduce the memory requirement for disk access of $B$. Continuing in this fashion, we decide to put loops in $I_{t4}$ in part $D_3$ and loops in $J_{t4}$ in part $D_2$.

The simplified code is shown in Figure 4.3(b). Note, that the particular loops put in index sets will not affect the minimum Overhead cost, but they will determine whether the conditions under which we can achieve the minimum Overhead cost are satisfied or not. This will be explained in detail in later sections.
4.3 Overhead Minimization for the Outside Communication Method

In this section, we analyze each of the three parallel algorithms (rotation, replication and accumulation) with the outside communication pattern and determine the minimal Overhead cost achievable along with the conditions under which this will be possible. In the expressions used in this and the next section, $A$, $B$, $C$ will denote the sizes of arrays $A$, $B$, $C$, respectively; the terms $I$, $J$, $K$ and $It_1$, $Jt_1$, $Kt_1$ will denote the corresponding loop bounds. The total number of processors will be denoted by $P$ and the local memory available for the tiles of each array, which we assume to be one-third of the local memory per processor, is denoted by $M$. The combined memory of all processors is, therefore, $M \times P$.

4.3.1 Rotation

Let us consider the tensor contraction code with disk I/O placement order $ABC$, the outside communication pattern, and rotation type of parallelism as shown in Figure 4.2(a). The tiling loops are ordered as discussed in the previous section. Our goal is to determine the tile sizes (or the number of tiles) that will minimize the Overhead cost, including disk I/O cost and communication cost.

In this case, each of the three arrays are partitioned equally among the $P$ processors. So we have $A.localsize = A/P$, $B.localsize = B/P$, and $C.localsize = C/P$. The communication corresponds to shifting the $A$ and $B$ arrays to adjacent processors. These communications happen $\sqrt{P}$ times and each of these also involves disk operations. Therefore, the total communication volume $V = \sqrt{P} \times (\frac{A}{P} + \frac{B}{P}) = (\frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}})$. The disk access volume during communication $D_1 = 2 \times (\frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}})$, since the disk is accessed twice, once for reading and once for writing. It is clear that these two terms are independent of the tile sizes.
The disk access volume during the computation $D_2 = \sqrt{P} \times (\frac{A}{P} + \frac{B}{P} \times I_t + 2 \times \frac{C}{P} \times K_t) = \frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}} \times I_t + 2 \times \frac{C}{\sqrt{P}} \times K_t$. The total disk access volume $D = D_1 + D_2$. For simplicity, in the calculations below $D$ will include only those two parts that depend on the number of tiles (or tile sizes).

It is clear that this term depends on the number of tiles. To minimize Overhead cost, we will have to minimize the disk access volume during the computation and hence $I_t, K_t$ should be made 1. But this is not possible due to the constraint that the tiles of array $A, B$ and $C$ should fit into memory. Here we assume that each of these array tiles occupies one third of the memory. The constraints involving tiles can be expressed as follows.

\[
I_t \times K_t \geq \frac{A}{M \times P} \quad (4.2)
\]
\[
J_t \times K_t \geq \frac{B}{M \times P} \quad (4.3)
\]
\[
I_t \times I_t \times J_t \geq \frac{C}{M \times P} \quad (4.4)
\]

Note that only Eqn. 4.2 involves both $I_t$ and $K_t$, which we want to be 1. We will try to minimize $D$ under the constraint of Eqn. 4.2. The other two equations can be simultaneously satisfied by using a large value the the unconstrained variables $I_t$ and $J_t$. Since we are trying to reduce the values of $I_t$ and $K_t$ while satisfying Eqn. 4.2, the Eqn. 4.2 can be written as $I_t \times K_t = \frac{A}{M \times P}$. With this modification, we can substitute the value of $K_t$ in the equation for $D$ to get,

\[
B \times I_t^2 - \sqrt{P} \times D \times I_t + \frac{2 \times C \times A}{M \times P} = 0 \quad (4.5)
\]

The above quadratic equation will have a real solution under the condition that the quadratic curve discriminant $P \times D^2 - 4 \times B \times (\frac{2 \times C \times A}{M \times P}) \geq 0$. In other words, for any real value of $I_t$, the minimum achievable value of $D$ is $\frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}$. This minimum value of $D$ can be achieved with $I_t = \sqrt{\frac{2}{M \times P}}$ and $K_t = \frac{K}{\sqrt{2 \times M \times P}}$. In order to satisfy Equations 4.3
and 4.4, we need to choose values of $Jt_1$ and $It_2$ that satisfy the conditions $Jt_1 \geq J \times \sqrt{\frac{2}{M \times P}}$ and $It_2 \geq 1$. Hence, the minimum total disk access volume is

$$2 \times \left( \frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}} \right) + \frac{A}{\sqrt{P}} + \frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}.$$  

(4.6)

If these values are not integers, the number of tiles will be set to the ceiling. There are two special cases if values of $It_1$ or $Kt_1$ are less than 1.

- **Case 1: $I < \sqrt{\frac{M \times P}{2}}$**

  In this case, we select the values as $It_1 = 1$, $Kt_1 = \frac{A}{M \times P}$, $Jt_1 \geq \frac{J}{T}$, $It_2 \geq \frac{I^2}{M \times P}$.

  The minimum total disk access volume during the computation in this case will be

  $$\frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}} + 2 \times \frac{C}{\sqrt{P}} \times \frac{A}{M \times P}.$$  

- **Case 2: $K < \sqrt{2 \times M \times P}$**

  In this case, we select the values as $It_1 = \frac{A}{M \times P}$, $Kt_1 = 1$, $Jt_1 \geq \frac{B}{M \times P}$, $It_2 \geq \frac{M \times P}{K^2}$.

  The minimum total disk access volume during the computation in this case will be

  $$\frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}} \times \frac{A}{M \times P} + 2 \times \frac{C}{\sqrt{P}}.$$  

We performed the analysis for the other five disk placement orders in a similar fashion. The results are shown in Table 4.2 and the details can be obtained from the associated technical report [18].

### 4.3.2 Replication

For this case, let us consider the tensor contraction code with disk I/O placement order $ABC$, outside communication pattern, and replication type of parallelism as shown in Figure 4.2(b). The tiling loops are ordered as discussed in the previous section. As in the case of rotation, our goal is to determine the tile sizes to minimize the Overhead cost.
Without loss of generality, we assume array $A$ is smaller than array $B$. Thus, the arrays $B$ and $C$ are partitioned equally among the $P$ processors whereas $A$ is replicated on all processors. So we have $A.\text{localsize} = A$, $B.\text{localsize} = B/P$, and $C.\text{localsize} = C/P$. In this case, communication corresponds to broadcasting array $A$. Therefore, the total communication volume $\mathcal{V} = A$. The disk access volume during communication $D_1 = A$. Also in this case the above two terms are independent of the tile sizes. The disk access volume during the computation $D_2 = A + \frac{B}{P} \times It_1 + 2 \times \frac{C}{P} \times Kt_1$. The total disk access volume $D = D_1 + D_2$.

It is clear that $D$ depends on the number of tiles. To minimize the Overhead cost, we will have to minimize the disk access volume during the computation and hence $It_1$, $Kt_1$ should be set to 1. But this is not possible due to the constraint that the tiles of arrays $A$, $B$, and $C$ fit into memory. The size constraints involving tiles can be expressed as follows.

\begin{align}
It_1 \times Kt_1 & \geq \frac{A}{M} \quad \text{(4.7)} \\
Jt_1 \times Kt_1 & \geq \frac{B}{M \times P} \quad \text{(4.8)} \\
It_1 \times It_2 \times Jt_1 & \geq \frac{C}{M \times P} \quad \text{(4.9)}
\end{align}

Our analysis here is similar to that for the case of rotation (Section 4.3.1). We will try to minimize $D$ under the constraint of Eqn. 4.7. The other two equations can be simultaneously satisfied by using a large value for the unconstrained variables $It_2$ and $Jt_1$. Since we are trying to reduce the values of $It_1$ and $Kt_1$ while satisfying Eqn. 4.7, the Eqn. 4.7 can be written as $It_1 \times Kt_1 = \frac{A}{M}$. With this modification, we can substitute the value of $Kt_1$ in the equation for $D$ to get

\begin{equation}
B \times It_1^2 - P \times D \times It_1 + \frac{2 \times C \times A}{M} = 0. \quad \text{(4.10)}
\end{equation}
From the above equation, it should be clear that for any real value of $I_1$, the minimum achievable value of $D$ is $\frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}$. This minimum value of $D$ can be achieved with $I_1 = I \times \sqrt{\frac{2}{M}}$ and $K_1 = \frac{K}{\sqrt{2 \times M}}$, $J_1 \geq \frac{I}{P} \times \sqrt{\frac{2}{M}}$ and $I_2 \geq 1$. These values satisfy all the constraints. Hence, the minimum total disk access volume is

$$A + A + \frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}$$  \hspace{1cm} (4.11)

If these values are not integers, the number of tiles will be set to the ceiling. There are two special cases if the values of $I_1$ or $K_1$ are less than 1. The analysis for these cases can be done as shown in Section 4.3.1.

We did the analysis for the other five disk placement orders as above. The results are shown in Table 4.2 and details can be obtained from [18].

### 4.3.3 Accumulation

In this section, we deal with the accumulation type of parallelism. Consider the tensor contraction code with accumulation type of parallelism as shown in Figure 4.2(c). Again our goal is to determine the tile sizes that will minimize the total Overhead cost.

In this case, arrays $A$ and $B$ are partitioned equally among the $P$ processors whereas $C$ is replicated on all processors. So we have $A.localsize = A/P$, $B.localsize = B/P$, $C.localsize = C$. In this case, the communication involves an All-Reduce operation of array $C$. Therefore, total communication volume $V = C \times \log P$. The disk access volume during communication $D_1 = C$. Again the total communication cost is independent of the tile sizes. The disk access volume during the computation $D_2 = \frac{A}{P} + \frac{B}{P} \times I_1 + 2 \times C \times K_1$. The total disk access volume $D = D_1 + D_2$.

As in the previous sections, to minimize the disk access volume during the computation, $I_1$ and $K_1$ should be made 1. But this is prevented by the constraint that the tiles of array
$A$, $B$, and $C$ should fit into memory. The constraints involving tiles in this case can be expressed as follows.

\begin{align*}
    It_1 \times Kt_1 & \geq \frac{A}{M \times P} \quad \text{(4.12)} \\
    Jt_1 \times Kt_1 & \geq \frac{B}{M \times P} \quad \text{(4.13)} \\
    It_1 \times It_2 \times Jt_1 & \geq \frac{C}{M} \quad \text{(4.14)}
\end{align*}

We do the analysis similar to that in the previous subsections. We will try to minimize $D$ under the constraint of Eqn. 4.12. The other two equations are simultaneously satisfied by using a large value for the unconstrained variables $It_2$ and $Jt_1$. As before, the Eqn. 4.12 can be written as $It_1 \times Kt_1 = \frac{A}{M \times P}$. Now substituting the value of $Kt_1$ in the equation for $D$ we get

\[
\frac{B}{P} \times It_1^2 - D \times It_1 + \frac{2 \times C \times A}{M \times P} = 0
\]

(4.15)

From this equation it is clear that, for any real value of $It_1$, the minimum achievable value of $D$ is $\frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}$. This minimum value of $D$ can be achieved with $It_1 = I \times \sqrt{\frac{2}{M}}$, $Kt_1 = \frac{K}{P \times \sqrt{2 \times M}}$, $Jt_1 \geq J \times \sqrt{\frac{2}{M}}$, and $It_2 \geq 1$. Hence, the minimum total disk access volume is

\[
C + \frac{A}{P} + \frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}
\]

(4.16)

If these values are not integers, the number of tiles will be set to the ceiling. There are two special cases if the values of $It_1$ or $Kt_1$ are less than 1. Again, the analysis for these cases can be done as shown in the previous subsections.

We did the analysis for the other five disk placement orders as above. The results are shown in Table 4.2 and details can be obtained from [18].
<table>
<thead>
<tr>
<th></th>
<th>$ABC$ or $ACB$</th>
<th>$BAC$ or $BCA$</th>
<th>$CAB$ or $CBA$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Outside/Rotation</strong></td>
<td>$V = \frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}}$</td>
<td>$V = \frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}}$</td>
<td>$V = \frac{A}{\sqrt{P}} + \frac{B}{\sqrt{P}}$</td>
</tr>
<tr>
<td></td>
<td>$D = \frac{A}{\sqrt{P}} + \frac{\sqrt{8A BC}}{P \sqrt{M}} + \frac{2}{\sqrt{P}}(A + B)$</td>
<td>$D = \frac{B}{\sqrt{P}} + \frac{\sqrt{8A BC}}{P \sqrt{M}} + \frac{2}{\sqrt{P}}(A + B)$</td>
<td>$D = \frac{2C}{\sqrt{P}} + \frac{2\sqrt{A BC}}{P \sqrt{M}} + \frac{2}{\sqrt{P}}(A + B)$</td>
</tr>
<tr>
<td><strong>Outside/Replication</strong></td>
<td>$V = A$</td>
<td>$V = A$</td>
<td>$V = A$</td>
</tr>
<tr>
<td></td>
<td>$D = 2A + \frac{\sqrt{8A BC}}{P \sqrt{M}}$</td>
<td>$D = \frac{B}{P} + \frac{\sqrt{8A BC}}{P \sqrt{M}} + A$</td>
<td>$D = \frac{2C}{P} + \frac{2\sqrt{A BC}}{P \sqrt{M}} + A$</td>
</tr>
<tr>
<td><strong>Outside/Accumulation</strong></td>
<td>$V = C \log P$</td>
<td>$V = C \log P$</td>
<td>$V = C \log P$</td>
</tr>
<tr>
<td></td>
<td>$D = \frac{A}{P} + \frac{\sqrt{8A BC}}{P \sqrt{M}} + C$</td>
<td>$D = \frac{B}{P} + \frac{\sqrt{8A BC}}{P \sqrt{M}} + C$</td>
<td>$D = 3C + \frac{2\sqrt{A BC}}{P \sqrt{M}}$</td>
</tr>
</tbody>
</table>

Table 4.2: Communication and Disk Access Volume for the **Outside Communication** pattern
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Communication Distance ($D$)</th>
<th>Disk Access Volume ($V$)</th>
<th>Lower Bound</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABC</strong></td>
<td>$D = \frac{A}{P} + 3 \sqrt{\frac{ABC}{M^2P}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is same as (3)</td>
<td>Lower bound is higher than (6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same as (1)</td>
</tr>
<tr>
<td><strong>ACB</strong></td>
<td>$D = \frac{B}{P} + 3\sqrt{\frac{ABC}{M^2P^2}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is higher than (3)</td>
<td>Lower bound is same as (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same as (4)</td>
</tr>
<tr>
<td><strong>BAC</strong></td>
<td>$D = \frac{B}{P} + 3\sqrt{\frac{ABC}{M^2P^2}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is same as (6)</td>
<td>Lower bound is higher than (6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same as (2)</td>
</tr>
<tr>
<td><strong>BCA</strong></td>
<td>$D = \frac{2C}{P} + 2\sqrt{\frac{ABC}{M^2P^2}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is same as (5)</td>
<td>Lower bound is higher than (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same as (7)</td>
</tr>
<tr>
<td><strong>CAB</strong></td>
<td>$D = \frac{2C}{P} + 2\sqrt{\frac{ABC}{M^2P^2}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is same as (9)</td>
<td>Lower bound is higher than (6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Same as (8)</td>
</tr>
<tr>
<td><strong>CBA</strong></td>
<td>$D = \frac{2C}{P} + 2\sqrt{\frac{ABC}{M^2P^2}}$</td>
<td>$V = 2\sqrt{\frac{\text{ABC}}{M^2P^2}}$</td>
<td>Lower bound is same as (10)</td>
<td>Lower bound is same as (10)</td>
</tr>
</tbody>
</table>

Table 4.3: Communication and Disk Access Volume for the **Inside/Rotation** pattern
Table 4.4: Effective Communication and Disk Access Volume for the Inside Communication pattern with replication/accumulation parallelism

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Inside/Replication</th>
<th>Inside/Accumulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC or ACB</td>
<td>( \frac{A}{P} + RA + \frac{\sqrt{ABC}}{P\sqrt{M}} )</td>
<td>( \frac{A}{P} + 2\frac{\sqrt{ABC(2+R\log P)}}{P\sqrt{M}} )</td>
</tr>
<tr>
<td>BAC or BCA</td>
<td>( \frac{B}{P} + \frac{\sqrt{ABC(1+RP)}}{MP^4} )</td>
<td>( \frac{B}{P} + 2\frac{\sqrt{ABC(2+R\log P)}}{P\sqrt{M}} )</td>
</tr>
<tr>
<td>CAB or CBA</td>
<td>( \frac{2C}{P} + \frac{\sqrt{ABC(1+RP)}}{MP^4} )</td>
<td>( (2 + R\log P)C + \frac{2\sqrt{ABC}}{P\sqrt{M}} )</td>
</tr>
</tbody>
</table>

4.4 Overhead Minimization for the Inside Communication Method

In this section, we analyze each of the three parallel algorithms possible with the inside communication pattern and determine the minimal Overhead cost achievable along with the conditions under which this will be possible.

4.4.1 Rotation

Consider the tensor contraction code with disk I/O placement order ABC, inside communication pattern, and the rotation type of parallelism as shown in Figure 4.1(a). The tiling loop ordering is decided as before. The goal is to determine the tile sizes (or the number of tiles) that will minimize the total Overhead cost.

In this case, each of the three arrays are partitioned equally among the \( P \) processors in a block-cyclic fashion. So we have \( A_{\text{localsize}} = A/P \), \( B_{\text{localsize}} = B/P \), \( C_{\text{localsize}} = C/P \). The communication corresponds to shifting the A and B arrays to adjacent processors. This communication happens \( \sqrt{P} \) times for each iteration of the tiling loops and each of these also involves disk operations. Therefore, the total communication volume \( V = \sqrt{P} \times \left( \frac{A \times I_1}{P} + \frac{B \times I_1 \times I_2}{P} \right) = \left( \frac{A \times I_1}{\sqrt{P}} + \frac{B \times I_1 \times I_2}{\sqrt{P}} \right) \). Due to in-memory transfer there will not be any disk access as part of the communication. The total disk access
volume $D = \frac{A}{P} + \frac{B}{P} \times It_1 + 2 \times \frac{C}{P} \times Kt_1$. For simplicity in the calculations below, $D$ will not include the component $\frac{A}{P}$, which is independent of the number of tiles.

First we will try to optimize $D$ and $V$ independently. To minimize the communication volume $V$, $It_1$, $It_2$ and $Jt_1$ should be made 1. But this is not possible due to the constraint that the tiles of array $A$, $B$, and $C$ should fit into memory. Again we assume that each of these array tiles occupies one-third of memory. The constraints involving tiles are the same as those shown in the rotation case of outside communication.

Note that only Equation 4.4 involves all the variables whose values we want to reduce namely $It_1$, $It_2$, and $Jt_1$. We will try to minimize $V$ under the constraint of Equation 4.4. The other two equations can be simultaneously satisfied by using a large value for the unconstrained variable $Kt_1$. Since we are trying to reduce the values of $It_1$, $It_2$, and $Jt_1$ while satisfying Equation 4.4, the Equation 4.4 can be written as $It_1 \times It_2 \times Jt_1 = \frac{C}{M \times P}$.

With this modification, we can substitute the value of $Kt_1$ in the equation for $V$ to get

$$B \times (It_1 \times It_2)^2 - \sqrt{P} \times V \times (It_1 \times It_2) + \frac{A \times C}{M \times P} = 0 \quad \text{(4.17)}$$

The above quadratic equation will have a real solution when the condition, quadratic curve discriminant $P \times V^2 - 4 \times B \times \left(\frac{A \times C}{M \times P}\right) \geq 0$ is true. In other words, for any real value of $It_1$ the minimum achievable value of $V$ is $2 \sqrt{\frac{A \times B \times C}{M \times P}}$. This minimum value of $V$ can be achieved with $It_1 = \frac{I}{\sqrt{M \times P}}$, $It_2 = 1$, $Jt_1 = \frac{I}{\sqrt{M \times P}}$, and $Kt_1 \geq \frac{K}{\sqrt{M \times P}}$ which also satisfies the Equations 4.2 and 4.3. Hence, the minimum total communication volume is

$$\frac{2}{P} \sqrt{\frac{A \times B \times C}{M}} \quad \text{(4.18)}$$

Now we will minimize the disk access volume independently. Note that $It_1$ and $Kt_1$ should be made 1 in this case. But this is not possible due to the constraint that the tiles of arrays $A$, $B$, and $C$ should fit into the memory. We will try to minimize $D$ under the constraint
of Eqn. 4.2. The other two equations can be simultaneously satisfied by using a large value for the unconstrained variables $I t_2$ and $J t_1$. The Eqn. 4.2 in this case can be written as $I t_1 \times K t_1 = \frac{A}{M \times P}$. With this modification, we can substitute the value of $K t_1$ in the equation for $D$ to get

$$B \times I t_1^2 - P \times D \times I t_1 + \frac{2 \times C \times A}{M \times P} = 0$$

(4.19)

From this equation we can see that for any real value of $I t_1$ the minimum achievable value of $D$ is $\sqrt{\frac{8 \times A \times B \times C}{M \times P^3}}$. This minimum value of $D$ can be achieved with $I t_1 = I \times \sqrt{\frac{2}{M \times P}}$ and $K t_1 = \frac{K}{\sqrt{2} \times M \times P}$, $J t_1 \geq J \times \sqrt{\frac{2}{M \times P}}$, and $I t_2 \geq 1$. These values will also satisfy Equations 4.3 and 4.4. Hence, the minimum total disk access volume is

$$D = \frac{A}{P} + 3 \sqrt{\frac{8 \times A \times B \times C}{M \times P^3}}$$

(4.20)

But it is obvious that the number of tiles does not match with that of the previous analysis to minimize communication volume. So we cannot optimize both the communication volume and disk access volume at the same time. We have computed the Overhead cost for both the cases and we choose the one which has the smaller Overhead cost. In this case we choose the number of tiles that optimizes the communication volume as this gives the least Overhead cost. The values of communication and disk access volume are as follows with these tile sizes:

$$V = \frac{2}{P} \sqrt{\frac{A \times B \times C}{M}}$$

(4.21)

$$D = \frac{A}{P} + \frac{B}{P} + \frac{2 \times C \times A}{M \times P^3}$$

(4.22)

There are three special cases if values of $I t_1$, $J t_1$, or $K t_1$ are less than 1.

- **Case 1:** $I < \sqrt{M \times P}$, $J \geq \sqrt{M \times P}$, $K \geq \sqrt{M \times P}$, In this case, the expected least overhead is $V = \frac{A \times C}{\sqrt{M \times P} \times P}$ and $D = \frac{A}{P} + \frac{B}{P} + \frac{2 \times C \times A}{M \times P^3}$ with $I t_1 = 1$, $I t_2 = 1$, $J t_1 = 1$, $K t_1 = 1$, and $I t_2 = 1$. 

61
\( Jt_1 = \frac{C}{M \times P}, \quad Kt_1 = \frac{A}{M \times P}. \) But with these values, Eqn. 4.3 is not satisfied. So, the least overhead above can not be really achieved. This is not a problem, though, as the expected lower bound in this case is same as the achievable lower bound of case ACB as shown in Table 4.3.

- **Case 2:** \( I \geq \sqrt{M \times P}, \quad J < \sqrt{M \times P}, \quad K \geq \sqrt{M \times P}, \) the expected least overhead is \( \mathcal{V} = \frac{A}{P} + \frac{B \times C}{\sqrt{M \times P}} \) and \( \mathcal{D} = \frac{A}{P} + 3 \times \sqrt{\frac{ABC}{M \times P}} \) with \( It_1 = 1, \quad It_2 = 1, \quad Jt_1 = \frac{C}{M \times P}, \quad Kt_1 = \frac{A}{M \times P}. \) But with these values, Eqn. 4.4 is not satisfied. So, the least overhead above can not be really achieved. We don’t need to mind it, from Table 4.3, we can see that the achievable lower bound of case BCA is \( \mathcal{V}' = \frac{A}{\sqrt{P}} + \frac{B \times C}{\sqrt{M \times P}} \) and \( \mathcal{D}' = \frac{A}{P} + \frac{A}{P} + 2 \times \frac{BC}{M \times P} \) which is lower than the expected lower bound in the current case.

- **Case 3:** \( I \geq \sqrt{M \times P}, \quad J \geq \sqrt{M \times P}, \quad K < \sqrt{M \times P}, \)

In this case, the least overhead that can be achieved is \( \mathcal{V} = 2 \times \sqrt{\frac{A \times B \times C}{M \times P}} \) and \( \mathcal{D} = \frac{A}{P} + \frac{B}{P} \times \frac{A}{M \times P} + \frac{2 \times C}{P} \) with \( It_1 = \frac{A}{M \times P}, \quad It_2 = \frac{\sqrt{M \times P}}{K}, \quad Jt_1 = \frac{I}{\sqrt{M \times P}}, \quad Kt_1 = 1. \) With these values, all the constraints are also satisfied.

We did the analysis for the other five disk placement orders as above. The results of the analysis are shown in Table 4.3.

**4.4.2 Replication**

For this case, let us consider the tensor contraction code with disk I/O placement order \( ABC, \) an inside communication pattern, and the replication type of parallelism as shown in Figure 4.1(b).
In this case, because the replication occurs in memory, and replicated data will be skipped after computation, so array $A$ is not replicated on disk. Arrays $A$, $B$ and $C$ are partitioned equally among the $P$ processors. We have $A.localsize = A/P$, $B.localsize = B/P$, $C.localsize = C/P$. The communication corresponds to an in-core broadcast of array $A$. Therefore, the total communication volume $\mathcal{V} = A$, and it is independent of the tile sizes. The total disk access volume $\mathcal{D} = \frac{A}{P} + \frac{B}{P} \times It_1 + 2 \times \frac{C}{P} \times Kt_1$.

The constraints involving tiles are the same as those shown in the replication part of outside communication. We do the analysis similar to the ones for the earlier cases. The minimum achievable value of $\mathcal{D}$ can be computed as $\frac{A}{P} + \frac{1}{P} \sqrt{\frac{8 \times A \times B \times C}{M}}$. This minimum value of $\mathcal{D}$ can be achieved with $It_1 = I \times \sqrt{\frac{A}{M}}$ and $Kt_1 = \frac{K}{\sqrt{2 \times M}}$, $Jt_1 \geq \frac{J}{P} \times \sqrt{\frac{A}{M}}$ and $It_2 \geq 1$. These values satisfy all the constraints. The analysis for the special cases can be done in the earlier sections.

The result of the analysis for the other five disk placement orders are shown in Table 4.4 and details can be obtained from [18]. Note that the values shown in this table are the effective communication and disk access volume $EffVol = \mathcal{D} + R \times \mathcal{V}$, where $R = \frac{B_d}{B_c}$, where $B_d$ is the disk bandwidth and $B_c$ is the communication (network) bandwidth.

### 4.4.3 Accumulation

In this section, we deal with the accumulation type of parallelism. Consider the tensor contraction code with the accumulation type of parallelism as shown in Figure 4.1(c). In this case, arrays $A$ and $B$ are partitioned equally among the $P$ processors whereas $C$ is replicated on all processors. So we have $A.localsize = A/P$, $B.localsize = B/P$, $C.localsize = C$. The communication involves in-core All-Reduce operation of array $C$. Therefore, the total communication volume $\mathcal{V} = C \times Kt_1 \times \log P$. The total disk access
volume $D = \frac{A}{P} + \frac{B}{P} \times It_1 + 2 \times C \times Kt_1$. In this case, we can optimize the total overhead cost, which is $\frac{EffVol}{B_d}$, where $EffVol$ is the effective communication and disk access volume given by (note that $R$ is defined at the end of Section 4.4.2)

$$EffVol = \frac{A}{P} + \frac{B}{P} \times It_1 + C \times Kt_1 \times (2 + R \times \log P).$$ (4.23)

Our goal is to minimize $EffVol$ under the constraints involving tile sizes that are shown in the accumulation section of the previous section. We proceed as before and compute the minimum achievable value of $EffVol$, which is found to be $\frac{A}{P} + 2 \times \sqrt{\frac{ABC(2+R \times \log P)}{M \times P^2}}$.

This minimum value is achieved with $It_1 = I \times \sqrt{\frac{(2+R \times \log P)}{M}}$, $Kt_1 = \frac{K}{P \times \sqrt{(2+R \times \log P) \times M}}$, $Jt_1 \geq J \times \sqrt{\frac{(2+R \times \log P)}{M}}$, and $It_2 \geq 1$.

The special cases are handled as before. The analysis for the other five disk placement orders are also done as above. The results are shown in Table 4.4 and details can be obtained from [18]. Again, note that the values in the table give the minimum value of $EffVol$.

### 4.5 Experiments

Our performance models for the various approaches to parallel out-of-core tensor contractions were evaluated on an Itanium-2 cluster at the Ohio Supercomputer Center. The configuration of the cluster is shown in Table 3.5. All the programs were compiled with the Intel Itanium Fortran Compiler for Linux. We considered three example computations.

(1) **Square Matrix Multiplication:**

$$C(I, J) + = A(I, K) \times B(J, K)$$ (4.24)

In order to limit the execution time we ran “scaled down” experiments by setting the available physical memory limit to 64Mbytes. All the array dimensions were set to 4000. The
parallel programs were run on 4 processors. We implemented parallel programs for the six methods discussed earlier. Table 4.5 compares the predicted costs for I/O and communication with the measured costs for the different approaches. It can be seen that there is a good match between predicted and actual times, and that the difference in performance of the various methods is quite significant.

(2) 4-index transform subexpression: The following contraction is from 4-index transform expression explained in Section 1.2.

\[ T1[a, b, c, d]^+ = A[a, b, c, p] \times B[p, d] \quad (4.25) \]

The size of all dimensions was set to 800. The parallel program was run on 4 processors. Between the different algorithms, we can find the best solution to be outside replication. The predicted overheads for the different parallel algorithms are shown in Table 4.6.

(3) CCSD: We used a sub-expression from the CCSD (Coupled Cluster Singles and Doubles) model [3, 38, 41] for determine electronic structures.

\[ T1[i, j]^+ = A[i, a, b, c] \times B[a, b, c, j] \quad (4.26) \]
The size of all dimensions was set to 800. The parallel program was run on 4 processors. The best solution on the current machine can be seen to be outside accumulation. The predicted values of different parallel algorithms are shown in Table 4.6.

<table>
<thead>
<tr>
<th></th>
<th>4index</th>
<th></th>
<th>ccsd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Disk I/O (GB)</td>
<td>Comm. (GB)</td>
<td>Total Time (sec.)</td>
<td>Disk I/O (GB)</td>
</tr>
<tr>
<td>Out/Ro.</td>
<td>1024</td>
<td>204.8</td>
<td>829440</td>
<td>1024</td>
</tr>
<tr>
<td>Out/Re.</td>
<td>307.2</td>
<td>0.00064</td>
<td>245760</td>
<td>512</td>
</tr>
<tr>
<td>Out/Acc.</td>
<td>921.600</td>
<td>819.2</td>
<td>778240</td>
<td>204.8</td>
</tr>
<tr>
<td>In/Ro.</td>
<td>307.2</td>
<td>205.8</td>
<td>256051</td>
<td>204.8</td>
</tr>
<tr>
<td>In/Re.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>921.6</td>
</tr>
<tr>
<td>Out/Acc.</td>
<td>512</td>
<td>1146.88</td>
<td>466944</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.6: Predicted performance results on 4 processors for ccscd and 4index-transform

The effective choice of parallel algorithms results in a noticeable improvement in the communication cost for most cases. The ratio of disk bandwidth and interprocessor network bandwidth determines which factor dominates the total execution time. In previous experiments, because the network is almost twenty times faster than the disk, the disk cost dominated. In such a situation, the inside rotation algorithm is the best. However, using our model, we are able to predict the best choice for a given machine and problem characteristic. Table 4.7 shows such an example for a given matrix multiplication and disk bandwidth, where $I = 160000$, $J = 160000$, $K = 160000$, and $B_d = 10MB/s$. If we use a 100M Ethernet as the interconnection network and run the program on 4 processors, then the best parallel algorithm is outside replication. If we use Myrinet and run the program on 4 processors, the best solution becomes inside rotation.
<table>
<thead>
<tr>
<th></th>
<th>I=J=K=160000 , 4 Processors</th>
<th>I = J = K =640000, 16 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$B_c = 10MB/s$</td>
<td>$B_c = 200MB/s$</td>
</tr>
<tr>
<td>Out/Ro.</td>
<td>281920</td>
<td>262464</td>
</tr>
<tr>
<td>Out/Re.</td>
<td>259683</td>
<td>232168</td>
</tr>
<tr>
<td>Out/Acc.</td>
<td>318784</td>
<td>264307</td>
</tr>
<tr>
<td>In/Ro.</td>
<td>310240</td>
<td>120240</td>
</tr>
<tr>
<td>In/Re.</td>
<td>268248</td>
<td>123502</td>
</tr>
<tr>
<td>In/Acc.</td>
<td>298304</td>
<td>243827</td>
</tr>
</tbody>
</table>

Table 4.7: When $B_d=10MB/s$, predicted best disk/communication overheads (in sec.)

### 4.6 Conclusion

This chapter addressed the problem of developing performance models for a parallel, out-of-core single contraction. The cost of disk I/O and interprocessor communication was modeled for various computational alternatives. The models were experimentally evaluated and the predictions were shown to match measured results. It was also seen that the optimal choice was dependent on the characteristics of both the tensor structure as well as machine parameters. The work presented in this chapter can be used in conjunction with high-performance libraries for parallel out-of-core matrix computations for selecting the best version of code at runtime.
CHAPTER 5

EFFICIENT SEARCH FOR INTEGRATED FUSION AND TILING TRANSFORMATIONS

In the previous chapter, we describe the optimization framework for a single tensor contraction. Our main purpose is to find the optimal program structure, which is determined by a number of program parameters (such as loop order, tile size, placements of I/O statements and communication statement), to minimize the total execution time. The main challenges lie in the huge amount of possible combinations and dynamical inter-actions between these factors. In our previous approach, we exploit the domain-specific features of tensor contractions to find out the dependence between different program parameters, and thereby to facilitate the exploration of the entire search space of optimizations. As the next step, we will try to extend our research work to more complicated and realistic cases involving a sequence of tensor contractions, where the producer-consumer relationship between contractions brings more complexity to the program structure.

A new loop transformation strategy, fusion, is introduced to merge common loop nests between different contractions, then reduce the memory requirement for intermediate results. In addition, other optimization strategies, such as tiling, are also applied across multiple contractions to increase data locality. Thus, the current framework becomes too simple to model and evaluate these behaviors. We develop a new integrated framework,
which takes the directly implemented program as input and performs a variety of transformation strategies on the program structure. These transformations are applied to dependent program parameters and interact with each other. Then, the effect of different transformations must be measured simultaneously. Compared to the old optimization framework, the new one includes more variations and causes a much larger searching space. It requires the underlying framework to provide more efficient pruning approaches and generic program structure representations.

In this chapter, we first discuss the exploration of the space of loop fusion and tiling transformations in order to minimize the disk I/O cost. An unified framework is developed to integrate these two transformations and pruning algorithms are presented that significantly reduce the number of loop structures to be evaluated for subsequent transformations. The evaluation of the framework using representative contraction expressions from quantum chemistry shows a dramatic reduction in the size of the search space using the strategies presented.

The rest of this chapter is organized as follows. The next section provide background information of loop transformation strategies. An overview of the new synthesis optimization framework, of which the work presented in this chapter is a only part, is given in Section 5.2. Section 5.3 describes a tree partitioning algorithm. In Section 5.4, we give a loop structure enumeration algorithm and prove its completeness. The reductions in the space of loop structures to be explored is shown for representative computations in Section 5.5. Conclusions are presented in Section 5.6.
5.1 Introduction

Optimizing compilers incorporate a number of loop transformations such as permutation, tiling, fusion, etc. Considerable work has addressed loop tiling for enhancement of data locality [9, 14, 19, 31, 39, 44, 50, 51, 52]. Much work has also been done on improving locality and/or parallelism by loop fusion [16, 17, 24, 25, 43]. Fusion often creates imperfectly nested loops, which are more complex to tile effectively than perfectly nested loops. Several works have addressed the tiling of imperfectly nested loops [2, 44]. Although there has been much progress in developing unified frameworks for modeling a variety of loop transformations [1, 2, 33, 52], their use has so far been restricted to optimization of indirect performance metrics such as reuse distance, degree of parallelism, etc. The development of model-driven optimization strategies that target direct performance metrics, remains a difficult task.

We face the problem in the specific domain of tensor contractions (generalized matrix products) involving tensors too large to fit into physical memory. We use special properties of the computations in this domain to integrate the various transformations and investigate pruning strategies to reduce the search space to be explored. The final plan for our integrated optimizing system is to determine the appropriate combination of loop permutation, fusion, and tiling, along with the best fitting position and ordering of I/O placements. In this chapter, the overall framework of the optimizing system will be represented, and implementation details of the first two phases are illustrated. The integration of loop fusion and tiling transformations will be discussed with the objective of minimizing disk I/O cost. We first divide the input program into several independent loop nests, then enumerate the set of candidate fusion structures of each loop nest. A generalized tiling approach is presented
that significantly reduces the number of loop structures to be explored. It also enables subsequent optimizations of I/O placements and loop permutations. This approach enables an exploration of the entire search space using a realistic performance model, without the need to resort to heuristics and search of a limited subspace of the search space to limit search time. Note that, the data structure loop nesting tree defined in Chapter 2 will be used in the new algorithm as the intermediate representation.

5.2 Overview of Integrated Optimization Framework

5.2.1 Optimization Process

Our synthesis optimization system takes an operation tree representing a set of tensor contractions as input, and generates a tiled loop structure with explicit I/O statements to implement the computation. The loop structure of an operation tree can be defined by two factors: (1) the partitioning method to divide the operation tree into a set of fused sub-trees; and (2) the internal loop structure (fusion, tiling and disk I/O placements) of each fused sub-tree. The optimization process may be viewed in terms of the following phases:

1. Operation Tree Partitioning: In this step, we divide the original operation tree into several fused subtrees by identifying cut-points. The optimal loop structures for different subtrees are independent of each other, and can be determined separately.

2. Loop Structures Enumeration: For each fused sub-tree, we find a set of candidate fusion structures to be evaluated, as a set of loop nesting trees.

3. Intra-Tile Loop Placements: For a given LNT, we tile all loops at each node and propagate intra-tile loops to all the nodes below it.
4. Disk I/O Placements Enumeration: We then enumerate all possible placements to insert disk I/O statements into a candidate loop structure.

5. Loop Permutation: For a loop nesting tree generated by the previous step, sorting and grouping loops in each node by the order of disk I/O statements within it.

6. Tile Size Selection: After determining the loop structure and disk I/O placements, the I/O cost of the program can be formulated as a non-linear optimization problem in terms of the tile sizes. The tile sizes that minimize the disk I/O cost are calculated using a general-purpose non-linear optimization solver.

7. Code Generation: We calculate the disk access cost for each solution obtained before, choose the one with the minimal disk I/O cost, and generate executable code for it.

5.2.2 Dynamic Programming Algorithm

A dynamic programming algorithm is developed to find the optimal loop structure of an operation tree based on the assumption that unfused intermediate arrays always would be put on disk. The algorithm calculates the minimal disk I/O cost and corresponding loop structure of each sub-tree of the original tree in bottom-up fashion. At a contraction node \( t \), all sub-trees rooted at its interior nodes are evaluated before, whose minimal disk I/O cost and optimal loop structure stored in their roots. Thus, we will only evaluate these new fused sub-trees, which are rooted at \( t \) and will be referred as top sub-trees of \( t \) in later description. The optimal top sub-tree would be the one that minimizes the sum of the disk cost occurred in itself and other sub-trees rooted at its leaves. After traversing the entire operation tree, the optimal loop structure can be obtained by tracking back the optimal top sub-trees from root to leaves.
Algorithm 1 is employed to find the optimal loop structure for the operation tree rooted at a given node $t$. It will be executed at each node of the operation tree from bottom-up. For an arbitrary node $t$, let $t.FS$ denote its optimal top sub-tree, which includes three fields: $TCS$, $FFS$ and $Cost$. $TCS$ is a set of cut-points to identify the range of the top subtree; $FFS$ represents its internal loop structure; and $Cost$ is the disk I/O cost occurred in it.

In the algorithm, the function $EnumerateTopSubtree(t)$ returns the set of all possible top sub-trees of $t$. The detail process will be explained in Section 5.3. After that, each of these sub-trees is evaluated in turn to find out their the optimal loop structure and disk I/O cost. The initial cost of a sub-tree is the sum of the costs of its leaves. Then, for a fused sub-tree $ts$, the function $EnumerateLoop(ts)$ will enumerate the set of candidate fusion structures represented by loop nesting trees. The enumeration algorithm is discussed in Section 5.4. For each candidate fusion structure $ffs$, the function $multiTiling(ffs)$ will insert intra-tile loops and return a multi-level tiled loop structure. The multi-level tiling strategy will be presented in Section 5.4 too. For a tiled loop structure $mtfs$, the search space of disk I/O placements, loop permutations and tile sizes is modeled and pruned as a non-linear optimization problem, which is then solved to determine the minimal disk I/O cost. This process is encapsulated in the procedure $dataLocality(mtfs)$. The implementation details can be found in [40].

5.2.3 Example

For the operation tree in Fig. 2.1(a), we start at the lowest contraction node $T1$, which has only one top sub-tree $t11$ as showed in Fig. 5.1(a). Then we have $T1.FS = t11$. The optimal loop structure and minimal cost of $t11$ is calculated using the functions.
Algorithm 1 SearchOptimalLoopStructure(t: the root of a subtree)

// Given a subtree rooted at t, the algorithm will find the optimal loop structure with minimal disk I/O

TreSet = EnumerateFusedSubtrees(t)
for each subtree Ti in TreSet do
    TCS = Ti.CutpointSet
    LeafCost = 0
    for each cut-point ct in TCS do
        LeafCost = LeafCost + ct.FS.Cost
    end for
    // Enumerate all loop structures of fused subtree Ti
    LoopSet = EnumerateLoop(Ti)
    OptCost = ∞
    // Compute the minimal disk I/O cost of subtree Ti
    for each loop structure FFS in LoopSet do
        Cost = dataLocality(FFS)
        if Cost < OptCost then
            OptCost = Cost
            OptFFS = FFS
        end if
    end for
    Cost = OptCost + leafCost
    if Cost < t.FS.Cost or t.FS = null then
        t.FS.Cost = Cost
        t.FS.TCS = TCS
        t.FS.FFS = OptFFS
    end if
end for

EnumerateLoop(t11), multiTiling(t11), and dataLocality(t11). For simplicity, we do ignore the specifics of this function and assume that t11.Cost = 100.

The second node T2 has two top sub-trees t21 and t22 as shown in Fig. 5.1(b). Assuming the internal disk cost of t21 is 150 and of t22 is 200. But, since the leaf T1 of sub-tree t21 is the root of another sub-tree t11, the total disk cost of t21 would be 250, higher than t22. So, we have T2.FS = t22, T2.FS.TCS = and T2.FS.Cost = 200.
Fig. 5.1(c) represents three top sub-trees of node $T_3$, where we assume their internal disk cost are 200, 250, and 300 respectively. Since $t_{22}$ is rooted at a leaf of $t_{31}$ and $t_{11}$ is rooted at a leaf of $t_{32}$, we get the total cost of these sub-tree as $t_{31}.Cost = 400$, $t_{32}.Cost = 350$, and $t_{33}.Cost = 300$. $t_{33}$ has the minimal total cost, then the optimal top sub-tree of $T_3$ would be $t_{33}$ with $Cost = 300$.

The four sub-trees identified at the root of the operation $B$, are shown in Fig. 5.1(d). The internal cost of these sub-trees are assumed to be 250, 300, 350, and 500. The optimal top sub-tree of $B$ is determined to be $t_B{3}$ with $Cost = 450$, which is also the minimal disk cost of the given operation tree in Fig. 2.1(a). The optimal tree partitioning method of the operation tree can be obtained by tracking back from the cut-points set of the root. $B.FS.TCS$ has one cut-point $T_1$ and $T_1.FS.TCS$ is empty. Hence the optimal tree partitioning method will divide the operation tree into two sub-trees at node $T_1$. The optimal loop structures of these sub-trees can be found in $B.FS.FFS$ and $T_1.FS.FFS$.

### 5.3 Tree Partitioning

In this section, we discuss the procedure to enumerate the set of top sub-trees. An arbitrary operation tree with $M$ intermediate nodes has at most $O(2^M)$ possible top sub-trees, but not all of them can be fully fused. We can prune the set of potential top sub-trees by using the following two rules: (i) the fused intermediate array must be fit into memory; and (ii) the parent of two fused nodes can not be fused above.

The first rule is used to prune ineffective fusions. In general, fusing a loop between the producer and consumer of an intermediate result eliminates the corresponding dimension of the intermediate array and reduces the array size. If the array fits in memory after fusion, no disk I/O is required for it. On the other hand, if the array does not fit in the physical
Figure 5.1: How to find the optimal loop structure of an operation tree by Algorithm 1
memory even after fusion, the disk I/O cost is not reduced and thus fusion does not result in any improvement. Therefore, we force the fusion of any loops corresponding to an intermediate node to cause the resulting intermediate to reside in memory. We also assume that an intermediate array resides on disk if its producer is not fused with its consumer.

The second rule is derived from the characteristics of an operation tree. Consider an intermediate node $t$. If both its children are fused with it, then the loops corresponding to the summation indices in the given node must be the outermost loops; and it can not be fused with its parent anymore. Thus either $t$ or one of its children must be a cut-point. Based on this rule, we can restrict the number of top sub-trees to $O(M^2)$.

From the first rule, it follows that contraction nodes form a chain. The second rule implies that two contraction chains may join at a root node, i.e., cut-point.

The function to enumerate the fused sub-trees rooted at a given node is shown in Algorithm 2. It is executed at each node of the operation tree in bottom-up manner and constructs the fused sub-trees rooted at a given node from those of its children. Given a node $t$, at first, we create a new sub-tree including only $t$ and its direct children. Then we extend existing sub-trees from one of its children to include itself. These sub-trees can be further extended to include the parent of $t$; so we call them the promising sub-trees, which would be in a single chain form (each node has at most one fused child). We can also create sub-trees by merging two existing sub-trees from both its children. In this case, $t$ must be a cut-point and this sub-tree cannot be extended anymore. In the algorithm, a top sub-tree $Tr$ is identified by its CutpointSet, which includes cut-points in its leaves; note that input nodes are not cut-points. The field $t.PTreeSet$ represents the set of promising sub-trees and will be used to construct the fused sub-tree rooted at the parent of $t$. Note that we do
not know whether a fused node can fit into memory at this step. This is ensured by the choice of loop structures.

5.4 Loop Structure Enumeration

In this section, we first present an algorithm that can generate a set of loop structures of a fused subtree. Then, we prove that for any loop structure $S$ of the fused subtree, we can find a corresponding loop structure $S'$ in the generated set, so that $S'$ can be transformed to $S$ by some proper multi-level tiling strategies.

5.4.1 Enumeration Algorithm

In the previous section, we showed that a fused subtree must be in one of these two forms:

- All contractions form a chain. We call it a *contraction chain*. For instance, Fig. 2.1 is such an operation tree, in which the contraction chain is $T_1, T_2, T_3, B$.

- The contractions form two chains joining at the root node. In this case, the *contraction chain* is connected by these two chains. An example of such an operation tree is shown in Fig. 5.2, in which the contraction chain is $T_1, T_2, B, T_3, T_4$

Given an operation tree that has $n$ contraction nodes $t_1, t_2, \ldots, t_n$, let $t_i.indices$ denote all loop indices surrounding the contraction node $t_i$. First, we create a contraction chain of the operation tree. It corresponds to a sequence of perfectly nested loops. Many different choices exist in the ordering of the fusions within this sequence of perfectly nested loop nests. Each of the perfectly nested loops, corresponding to a contraction, can be considered an independent loop nesting tree. The fusion of subtrees producing and consuming an intermediate creates an imperfectly nested loop nests, in which some of the common loops
are merged. The process of construction of the loop nesting tree of the fused sub-tree can be modeled as a parenthesization problem. Consider the sequence of contraction nodes T1, T2, T3, and B in the operation tree shown in Fig. 2.1. \(((T1(T2 T3))B)\) corresponds to a parenthesization in which the contractions producing T3 and consuming T3 are fused first and the resulting loop nest is fused with the contractions producing T1 and B, in that order. Fig. 5.3 shows one possible parenthesization for the four-index transform and the corresponding loop nesting tree.

We enumerate all possible parenthesizations of the contraction chain. For each parenthesization, a maximally fused loop structure is created by a recursive construction procedure. We call it *maximally fused* since, in the construction procedure, each intermediate node will have its indices fused as much as possible with its parent. The construction procedure is shown in Algorithm 3. It takes a parenthesization P as input, and generate a corresponding LNT. A parenthesization of a contraction chain with n nodes has \(n - 1\)
pairs of parentheses. Each pair of parentheses includes two elements, left and right element. Each element is either a single contraction node, or a parenthesization of a sub-chain within a pair of parentheses.

For easy understanding, we use an example to explain how the algorithm works. Consider a parenthesization \(( (T_1(T_2 T_3))B)\) of four-index transform. Fig. 5.3 shows how the construction procedure creates the corresponding LNT step by step.

---

**Figure 5.3**: Construction of a maximally fused loop structure for a particular parenthesization of the four-index transform.
5.4.2 Completeness

In this section, we prove that the set of \textit{maximally fused} loop structures generated by the enumeration algorithm above can represent all loop structures of a fused subtree. The following definitions are provided to clarify terms used in the proof.

\textbf{Definition 5.4.1} Each leaf in a LNT includes a contraction node. The set of contraction nodes from all the leaves in a LNT is called \textit{leafContractions} of the LNT.

\textbf{Definition 5.4.2} In a LNT, each node $t$ has exactly one path to the root. Let $t.upperIndices$ denotes the union of all indices belonging to nodes on the path from $t$ to the root. If a sub-tree $slnt$ is rooted at $t$, we also define $slnt.upperIndices$ to equal to $t.upperIndices$.

\textbf{Definition 5.4.3} In a LNT, suppose two leaves $t_i$ and $t_j$ belong to one subtree $slnt$. If there is no other subtree that contains both $t_i$ and $t_j$ and is a subtree of $slnt$, then we say that $slnt$ is the \textit{minimal common subtree} of $t_i$ and $t_j$, denoted as $MCS(t_i, t_j)$.

Given an arbitrary loop nesting tree $lnt$, we can map it to a maximal fused loop nesting tree $lnt'$, which is in the set of \textit{maximally fused} loop structures generated by the enumeration algorithm above, and can be translated to $lnt$ with some proper multi-level tiling strategy. The mapping algorithm consists of two steps:

1. Take $lnt$ as input, and generate a parenthesization $P$ of the contraction chain using the generation routine provided in Algorithm 4.

2. Apply the construction procedure in Algorithm 3 on $P$ to generate a maximally fused loop structure $lnt'$. 

81
Obviously, \(Lnt'\) is the set of \textit{maximally fused} loop structures generated by the enumeration algorithm. Afterward, we show that \(Lnt'\) can be translated to \(Lnt\) by sinking indices at upper levels down.

\textbf{Lemma 1} For any pair of contraction nodes \(t_i\) and \(t_j\), let \(\text{common}(Lnt, t_i, t_j)\) be the loops shared by \(t_i\) and \(t_j\) in \(Lnt\). We have \(\text{common}(Lnt, t_i, t_j) \subseteq \text{common}(Lnt', t_i, t_j)\).

\textbf{Proof 5.4.4} Given a subtree \(sLnt\), \(sLnt.upperIndices\) represents all common loops shared by \(sLnt.leafContractions\).

\textit{There is an interesting property of maximally fused loop structures in the way they are constructed. For any subtree \(sLnt\) in the LNT of a maximally fused loop structure, \(sLnt.upperIndices\) includes all common loops among \(sLnt.leafContractions\). In other words, it includes all possibly shared loops among \(sLnt.leafContractions\). In addition, from the mapping method, we can see that if \(Lnt\) has a subtree \(sLnt\), then there exist a twin subtree \(sLnt'\) in \(Lnt'\), which satisfied the following conditions:}

\[
\text{slnt.leafContractions} = \text{slnt'.leafContractions} \\
\text{slnt.upperIndices} \subseteq \text{slnt'.upperIndices}
\]

\textit{Given any pair of leaf nodes \(t_i\) and \(t_j\), we define \(mlnt = MCS(t_i, t_j)\) in \(Lnt\), where \(mlnt.upperIndices = \text{common}(Lnt, t_i, t_j)\). \textit{Hence, we can find the corresponding subtree \(mlnt'\) in \(Lnt'\), where}}

\[
\text{mlnt.upperIndices} \subseteq \text{mlnt'.upperIndices} \subseteq \text{common}(Lnt, t_i, t_j)
\]

\textit{Thus, we have \(\text{common}(Lnt, t_i, t_j) \subseteq \text{common}(Lnt', t_i, t_j)\).

\textbf{Lemma 2} If \(\text{common}(Lnt, t_i, t_j) \subset \text{common}(Lnt', t_i, t_j)\), then we can transform \(Lnt'\) to form \(Lnt''\) by sinking indices down, so that \(\text{common}(Lnt, t_i, t_j) = \text{common}(Lnt'', t_i, t_j)\).}
Proof 5.4.5 We define $mlnt$ and $mlnt'$ as $MCS(t_i, t_j)$ in $Int$ and $Int'$ respectively. Any loop in $\text{common}(Int', t_i, t_j)$ belongs to the root or an ancestor of $mlnt'$. Assuming loop $l$ is in the difference of $\text{common}(Int, t_i, t_j)$ and $\text{common}(Int', t_i, t_j)$. We remove $l$ from the original node $r$, and insert it to all children of $r$. After that, if $l$ still belongs to the root or an ancestor of $mlnt'$, we repeat the sinking operation described above, until $l$ is not in $mlnt'.upperIndices$ any more. The same method is applied for all indices in the difference of $\text{common}(Int, t_i, t_j)$ and $\text{common}(Int', t_i, t_j)$. The new LNT is denoted as $Int''$. Then, we have $\text{common}(Int, t_i, t_j) = \text{common}(Int'', t_i, t_j)$. 

Applying the sinking operation in Lemma 4 for each pair of contraction nodes $(t_i, t_j)$, we can transform $Int'$ to $Int''$, which satisfies the condition: $\forall(t_i, t_j), \text{common}(Int, t_i, t_j) = \text{common}(Int'', t_i, t_j)$. After that, if a node $r$ has no indices in $r.indices$, we remove $r$ from $Int''$, and put all children of $r$ to its parent. Then, $Int''$ is same as $Int$.

Using multi-level tiling strategy, a maximally fused loop structure can be transformed into an arbitrarily fused loop structure by appropriate choice of tile sizes. Multi-level tiling can transform the LNT of a loop structure as follows. Each loop present in the root is split into two components, inter-tile loop and intra-tile loop. The intra-tile loop is placed on child nodes of the root. Then the loops present in each of the child nodes including the intra-tile loops from the root, are again split and intra-tile loops are placed on their respective child nodes. This process is performed recursively till the leaf nodes are encountered. The loop structure corresponding to the LNT can also be transformed accordingly. Figure 5.4 shows the way to tile loop $a$ in the LNT in Fig. 5.3 and the relationship between different tiles, where $a.range$ represents the range of loop $a$.

The sinking operation in LNT can be modeled as the multi-level tiling in the loop structure. Given a loop structure, if we tile a fused loop with a tile size equal to its loop range, it
leads to the same result as we sink the loop index from original node to all its children. Let $S$ and $S'$ be loop structures represented by $lnt$ and $lnt'$ respectively. Since we can transform $lnt'$ to $lnt$ by sinking operations, we can also transform $S'$ to $S$ by proper multi-level tiling strategies. Next, we will use an example to show the details of the transformation procedure.

An arbitrary fully fused loop structure $S$ of four-index transform is shown in Figure 6.3(a), and the corresponding maximally fused loop structure $S'$ is in Figure 6.3(b). After we apply multi-level tiling strategies, $S'$ is translated to the format shown in Figure 6.2(a). In addition, if we set ranges of inter-tile loops according to the following formulas, and remove all loops with $range = 1$, then $S'$ can be rewritten as the format shown in Figure 6.2(b), which is exactly the same as $S$. It should be noted that the indexing of the intermediate arrays has been shown in a more generic way.

\[
\begin{align*}
aT_2 = aT_3 = sT_1 = sT_2 = sT_3 = rT_2 = qT_1 = 1 & \quad aT_1 = a.range & \quad rI_1 = r.range \\
\end{align*}
\]
for $a$
for $r$
for $q,s,r,p$
\[ t_{1,q} += A_{p,q,r,s} * C_{4,a,p} \]
for $b,s,q$
\[ t_{2,b,r,s} += t_{1,s,q} * C_{3,b,q} \]
for $b,c,r,s$
\[ t_{3,b,c,s} += t_{2,b,r,s} * C_{2,c,r} \]
for $b,c,d,s$
\[ B_{a,b,c,d} += t_{3,b,c,s} * C_{1,d,s} \]

(a) Arbitrary fused loop structure: $S$

for $a,s$
for $r$
for $q$
\[ t_{1} += A_{p,q,r,s} * C_{4,a,p} \]
for $p$
\[ t_{2} += t_{1} * C_{3,b,q} \]
for $b$
\[ t_{3,b,c} += t_{2} * C_{2,c,r} \]
for $b,c$
\[ B_{a,b,c,d} += t_{3,b,c} * C_{1,d,s} \]

(b) Maximally fused loop structure: $S'$

Figure 5.5: An arbitrary loop structure and the corresponding maximally fused structure

5.4.3 Complexity

The total number of loop structures generated by the enumeration algorithm is the same as the number of parenthesizations of the contraction chain. For a contraction chain with $n$ nodes, the number of all possible parenthesizations is called the $n^{th}$ Catalan Number. It is exponential in $n$, and the upper bound is $O(4^n/n^{3/2})$. In contrast, the number of possible loop structures is potentially exponential in the total number of distinct loop indices in the $n$ intermediate nodes, a considerably larger number. The fused operation tree is not very long for most representative computations. In most practical applications, a fused subtree usually has no more than 5 contractions in a single chain. Note that the $n^{th}$ Catalan Number is not very large when $n$ is small. The first six Catalan Numbers are listed here: 1, 1, 2, 5, 14, 42,...
for $aT_1, sT_1$
for $rT_1, aT_2, sT_2$
for $qT_1, rT_2, aT_3, sT_3$
for $p, qI_1, rI_2, aI_3, sI_3$
for $b, qI_1, rI_2, aI_3, sI_3$
for $b, c, rI_1, aI_2, sI_2$
for $aI_1, b, c, d, sI_1$

$B_{a,b,c,d} = t3_{aI,b,c,sI} * C_{1d,s}$

(a) After inserting intra-tile loops

for $aT_1$
for $rT_1$
for $p, qI_1, sI_3$
for $b, qI_1, sI_3$
for $b, c, aI_2, sI_2$
for $b, c, sI_1$

$B_{a,b,c,d} = t3_{aI,b,c,sI} * C_{1d,s}$

(b) After selecting proper tile counts

Figure 5.6: Translate S’ to S by multi-level tiling strategy

5.5 Results

The enumeration algorithm discussed in Section 5.4.1 generates a set of candidates loop structures to be considered for data locality optimization. Without this algorithm, and generalized tiling, the set of loop structures to be evaluated might be too large, precluding their complete evaluation and necessitating the use of heuristics.

We evaluate the effectiveness of our approach using the following tensor contractions from representative computations from the quantum chemistry domain.

1. **4-index Transform Expression**: A sequence of tensor contractions introduced in Section 1.2.

2. **CCSD**: The second and the third computations are from the class of Coupled Cluster (CC) equations [15, 32, 49] for ab initio electronic structure modeling. The sequence
of tensor contraction expressions extracted from this computation is shown as follows:

\[
S(j, i, b, a) = \sum_{l,k} (A(l, k, b, a) \\
\times (\sum_{c}(B(d, c, l, k) \times C(i, c)) \times D(j, d)))
\]

3. **CCSDT**: This is a more accurate CC model. A sub-expression from the CCSDT theory is:

\[
S(h3, h4, p1, p2) = \sum_{p9, h6, h8} (y.ooovuv(h8, h6, h4, p9, p1, p2) \times \\
\sum_{h10} (t.vo(p9, h10) \times \sum_{p7} (t.vo(p7, h8) \times \\
\sum_{p5} (t.vo(p5, h6) \times y.ooovv(h10, h3, p7, p5)))))
\]

We evaluated the fused subtree corresponding to the entire operation tree without any cut-points. The number of all possible loop structures and the number of candidate loop structures enumerated by our approach are shown in Table 5.1. It can be seen that a very large fraction of the set of possible loop structures, up to 98%, is pruned away using the approach developed in this chapter.

<table>
<thead>
<tr>
<th>#Contractions</th>
<th>#Loop structures</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Pruned</td>
</tr>
<tr>
<td>4index</td>
<td>4</td>
<td>241</td>
</tr>
<tr>
<td>CCSD</td>
<td>3</td>
<td>69</td>
</tr>
<tr>
<td>CCSDT</td>
<td>4</td>
<td>182</td>
</tr>
</tbody>
</table>

Table 5.1: Effectiveness of pruning of loop structures.
5.6 Conclusions

In this chapter we addressed the problem of optimizing the disk access cost of tensor contraction expressions by applying loop transformations. We discussed approaches to partitioning of the operation tree into fused sub-trees and generating a small set of “maximally-fused” loop structures that “cover” all possible imperfectly nested fused loop structures. The approach was evaluated on a set of computations representative of the targeted quantum chemistry domain and a significant reduction was demonstrated in the number of loop structures to be evaluated.
Algorithm 2 EnumerateTopSubtrees(t: the root of a sub-tree) returns TreeSet

\[
t_1 = \text{the left child of } t \\
t_2 = \text{the right child of } t \\
\text{if } t_1 \text{ is an input node, } b_1 = \text{null, else } b_1 = t_1 \\
\text{if } t_2 \text{ is an input node, } b_2 = \text{null, else } b_2 = t_2 \\
TreeSet = \text{empty} \\
\]
//Create a new sub-tree
Create a new Tree Tr with Tr.CutpointSet = \{ b_1, b_2 \}
Insert Tr into TreeSet
//Extending promising sub-trees from its left child
\text{if } b_1 \text{ is not null then} \\
\quad \text{for each sub-tree } st \text{ in } t_1.PTreeSet \text{ do} \\
\quad \quad \text{Create a new Tree } Tr \text{ with } Tr.CutpointSet = st.CutpointSet + b_2 \\
\quad \quad \text{Insert } Tr \text{ into TreeSet} \\
\quad \text{end for} \\
\text{end if} \\
//Extending promising sub-trees from its right child
\text{if } b_2 \text{ is not null then} \\
\quad \text{for each sub-tree } st \text{ in } t_2.PTreeSet \text{ do} \\
\quad \quad \text{Create a new Tree } Tr \text{ with } Tr.CutpointSet = st.CutpointSet + b_1 \\
\quad \quad \text{Insert } Tr \text{ into TreeSet} \\
\quad \text{end for} \\
\text{end if} \\
t.PTreeSet = \text{TreeSet} \\
//Merging sub-trees from both children, and extending the result
\text{if both } b_1 \text{ and } b_2 \text{ are not null then} \\
\quad \text{for each pair of sub-trees } st_1 \text{ in } childSet1 \text{ and } st_2 \text{ in } childSet2 \text{ do} \\
\quad \quad \text{Create a new Tree } Tr \\
\quad \quad Tr.CutpointSet = \{ st_1.CutpointSet, st_2.CutpointSet \} \\
\quad \quad \text{Insert } Tr \text{ into TreeSet} \\
\quad \text{end for} \\
\text{end if} \\
\text{return TreeSet}
Algorithm 3 Construction\((P)\)

//Given a parenthesization, the algorithm map it to a maximally fused loop structure in LNT

\[
\begin{align*}
l &= P.left \\
r &= P.right \\
\text{if } l \text{ is a parenthesization then} & \quad \quad \quad \quad \text{lt} = \text{Construction}(left) \\
\text{else if } l \text{ is a contraction then} & \quad \quad \quad \quad \text{lt} = \text{Create a new LNT node} \\
& \quad \quad \quad \quad \text{lt.indices} = l.indices \\
& \quad \quad \quad \quad \text{lt.children} = \text{null} \\
& \quad \quad \quad \quad \text{lt.contraction} = l \{\text{lt is a leaf, which includes a contraction node in it}\} \\
\text{end if} \\
\text{if } r \text{ is a parenthesization then} & \quad \quad \quad \quad \text{rt} = \text{Construction}(right) \\
\text{else if } r \text{ is a contraction then} & \quad \quad \quad \quad \text{rt} = \text{Create a new LNT node} \\
& \quad \quad \quad \quad \text{rt.indices} = r.indices \\
& \quad \quad \quad \quad \text{rt.children} = \text{null} \\
& \quad \quad \quad \quad \text{rt.contraction} = r \{\text{rt is a leaf, which includes a contraction node in it}\} \\
\text{end if} \\
\text{comindices} &= lt.indices \setminus rt.indices \\
\text{lt.indices} &= lt.indices \setminus \text{comindices} \\
\text{rt.indices} &= rt.indices \setminus \text{comindices} \\
\text{lnt} &= \text{Create a new LNT node} \\
\text{lnt.indices} &= \text{comindices} \\
\text{lnt.children} &= \{lt, rt\} \\
\text{return lnt}
\end{align*}
\]
Algorithm 4 Parenthesize(lnt)
//Given an LNT, the algorithm map it to a corresponding parenthesization

if lnt.children ≠ null then
    P = null
    for each child c in lnt.children do
        P' = Parenthesize(c)
        if P is null then
            P = P'
        else
            P = new Parenthesization(P, P')
        end if
    end for
else
    P = c.contraction {c is a leaf and includes a contraction node}
end if
return P
CHAPTER 6

EFFICIENT SEARCH FOR LOOP TRANSFORMATIONS TO OPTIMIZE DATA MOVEMENT

The disk I/O cost of an out-of-core program are determined by the loop structure used for data movement, the tile size for each data subset, and the actual placements of disk I/O operations within the loop structure. Loop transformations techniques can be applied to adjust the loop structure properly, then improve the disk I/O performance. In the previous chapter, we present an optimization framework that can integrate loop fusion and tiling transformations together, and then use a general-purpose nonlinear optimization solver to calculate the optimal loop structure. However, how to arrange disk access operations in the selected loop structure is still a challenging problem.

The disk I/O placement is defined as the location of a disk access statement in an imperfectly nested loop structure. Given a disk access operation, its location will determine the size of its local memory buffer and the volume of required data movement between memory and disk. If an application has multiple disk resident datasets, how to orchestrate corresponding disk access operations to minimize the total disk cost while satisfying the memory constraint is not simple and straightforward. Furthermore, it becomes more complicated when loop structure is considered. The optimal disk I/O placement varies in different loop structures. When loop fusion and tiling change a loop structure, they change
the disk access operations inside those loops as well. For instance, loop fusion may increase both the number and the total volume of data movements; and tiling may increase the local buffer sizes, thus leading to runtime failures of some disk access operations due to physical memory limitation. In other words, the loop structure, tile sizes and disk I/O placements are three related implementation parameters. To derive the optimal form of an out-of-core program, we can’t ignore complex interactions between them, and find the optimal solution for each one independently. Instead, we would search along all three dimensions simultaneously. However, it is impractical to do such a brute-force search due to the huge number of possible combinations.

In our previous work, we proposed two approaches to find the proper disk I/O placements. The first one is a decoupled approach that initially searches for a fusion structure with minimal disk cost by default tile sizes and disk I/O placements; then tiles the chosen fusion structure and inserts disk access operations in it. It may result sub-optimal solution, since the default disk I/O placements may be quite different than the final solution; and the disk I/O cost of a fusion structure is significantly influenced by the corresponding disk I/O placements. The second approach is to search all possible I/O placements and tile sizes for a multi-level tiled loop structure. It can guarantee to find the optimal solution. But, even with some pruning method, it is still so expensive that can only be used in very simple cases. Taking advantage of the two approaches above, we provide another solution. That is decoupled approach that first generates a set of candidate fusion structures together with disk I/O placements. Then, for each fusion structure in the set, we search for the optimal tile sizes to minimize the disk I/O cost. Finally, the tiled loop structure with lowest disk I/O cost is chosen. The algorithm of properly tiling a imperfectly-nested loop structure is the same as in the first decoupled approach, and has been proposed in [27]. This chapter
focuses on the approach to produce the set of promising fusion structures with disk I/O placements. We provide a model to analyze the relationship between fusion structure and disk I/O placements, and assess their combined effect of disk I/O cost and memory usage. Then, a novel algorithm is derived to compute the optimal loop structure and location of a single disk access operation. For multiple disk access operations, we propose a heuristic but efficient algorithm to insert them into a maximally-fused fusion structure, and transform the fusion structure suitably to reduce the total disk I/O cost.

This chapter is organized as follows. We discuss the background and basic assumptions of our work in Section 6.1. The impact of fusion structure on memory usage and disk I/O cost is discussed in Section 6.2. In section 6.3, we elaborate on the model to describe the relationship between disk I/O placement and fusion structures, and propose the algorithm to find the optimal fusion structure and location for a single disk access operation. The algorithm for determining the optimal disk I/O placement and fusion structure for a set of disk access operations is provided in Section 6.4. Section 6.5 presents results from the application of the new algorithm to an example abstracted from NWChem [45]. Finally, our conclusions are provided in Section 6.6.

6.1 Preliminaries

Consider the following expression:

$$E(i, k) = \sum_{h,l,j} A(i, l) \times B(l, j) \times C(j, h) \times D(h, k)$$  \hspace{1cm} (6.1)

Here, $A(i, l)$, $B(l, j)$, $C(j, h)$ and $D(h, k)$ are input arrays. They are initially stored on disk. $E(i, k)$ is the output array to be placed on disk at the end of computation. To reduce the number of floating point operations, the calculation is broken down into three binary
The operation-minimization transformation will create two intermediate, $t_1(i, j)$ and $t_2(j, k)$. They only exist during computation, and have no need to be stored for future use. However, they may be too large to fit into memory, and must be written to disk after production and read from disk before consumption in the next step. Fusing the common loops involved in the contractions that produce and consume an intermediate array can eliminate the corresponding dimensions of the intermediate array, and thereby reduce the size of the intermediate to retain in memory. In this way, loop fusion can save disk I/O cost. On the other side, for intermediate arrays that can’t fit in memory after fusion, and input and output arrays that can’t be loaded into memory entirely, loop fusion may cause their disk I/O statements to be put inside loops not corresponding to their dimensions. It results in these disk arrays to be loaded redundantly multiple times. Therefore, the total disk access cost in the fused program may be higher than that in the program without loop fusion.

As disk I/O placement is directly associated with the fusion structure, it can not be measured as an independent parameter. In addition, without knowing the exactly locations of disk access operations, we can’t make a concrete estimation of the benefit of loop fusion for disk I/O cost. Thus, we develop a framework to model the relationship between loop fusion and disk I/O scheduling, and derive an algorithm to search for the optimal fusion
structure and disk I/O placement simultaneously. The algorithm will obtain a set of candidate solutions of fusion structures with explicit disk access operations. For each solution, loop tiling will be applied to reduce the number of redundant disk access operations.

6.2 Fusion Structure and Disk I/O cost

In this section, first we introduce an important notion maximally-fused fusion structure and a new data structure called loop nesting tree. Then, we analyze the impact of fusion structures on disk I/O cost, and explain the reason why we design an algorithm to transform a maximally-fused loop structure to improve disk I/O performance.

6.2.1 Maximally-Fused Fusion Structure

There are many different ways to fuse loops and they could result in different fusion structures. In the last chapter, we proved that if a fused intermediate array can’t fit into memory, loop fusion applied on this intermediate will not gain any performance improvement at all. Therefore, we require that after a loop fusion, the resulted intermediate array must fit into memory. Otherwise, such fusion will not be performed. In a loop structure, only non-fused arrays (including input and output arrays) may be disk-resident. Fused arrays are always retained in memory.

A special type of fusion structure, maximally-fused fusion structure, is introduced in the last chapter, in which intermediate arrays have their indices fused as much as possible in a particular order. Given an arbitrary fusion structure of a sequence of tensor contractions, it can be transformed from one of maximally-fused fusion structures by performing appropriate loop fission. For example, the fusion structure shown in Figures 6.3(b) is a maximally-fused fusion structure, which can be transformed to the fusion structure shown in Figures 6.3(c) by splitting the loop $i$ into two loops. In other words, given a sequence of
tensor contractions, the set of maximally-fused fusion structures are general constructional templates for all possible fusion structures. Because of such characteristic, we design a new approach to get the optimal fusion structure with minimal disk I/O cost in an acceptable period of time. In this approach, we transform each maximally-fused fusion structure to the optimal form for the set of required disk access operations. Conceptually, the optimal fusion structure would be included in the set of transformed results. Since maximally-fused fusion structures occupy only a small fraction of total fusion structures, if the transformation can be performed in reasonable time, this approach will be much quicker than exploring all possible fusion structures.

6.2.2 Loop Nesting Tree

**Definition 1** If the dimension $i$ of intermediate array $t$ is eliminated after loop fusion, we call loop $i$ as a fused loop of array $t$, and array $t$ as a fused array of loop $i$.

In our previous work, loop nesting tree (LNT) is used to represent the loop configuration by a tree structure. In this section, we will extend it to provide additional informations to describe:

- the relationship between fused loops and intermediate arrays
- the location of disk access operations in the loop structure.

It will offer convenient basis for our new algorithm in the next section.

In an LNT, each node $N$ includes the indices of a set of fully permutable loops that appear at the same level of the loop structure. The leaves represent the innermost loops, while other nodes represent fused loops shared by different contractions. Loops in the same node have the same set of fused arrays, which is stored in the field fusedarrays of each
node. For example, Figures 6.1(a) shows the LNT corresponding to the fusion structure in Figures 6.2(a).

Note here that, the fusion structure in above example doesn’t include disk access operations. After explicitly orchestrating disk access operations into a fusion structure, loops in the same level may be split by a disk access operation, and no longer interchangeable with each other. Since these loops still share the same set of fused arrays, we don’t split the corresponding node in the LNT as in the fusion structure, but insert the disk access operation into the sequence of loops in this node. To illustrate the modified fusion structure and locations of disk access operations, we use an ordered list instead of a set to represent loops in a node. Each item of the list represents a set of permutable loops $L_s$, and a set of disk access operations $D_s$ nested immediately inside $L_s$. It is possible that $L_s$ or $D_s$ is empty for a given item, but not both of them. If $L_s$ in item $E_1$ is outside of $L_s$ in item $E_2$, then $E_1$ will appear before $E_2$ in the list. Based on the loop nesting tree, we define the relative position of loops and disk access operations as following.

**Definition 2** In an LNT, if disk access operation $D$ and loop nest $i$ are in the same node, and the item including $i$ is immediately followed by the item including $D$, we say that loop $i$ is below disk access operation $D$, and $D$ is above loop $i$.

In the corresponding fusion structure, loop $i$ and disk access operation $D$ would be surrounded by the same outside loops.

Loops below a disk access operation and loops inside these below loops will not affect the disk access operation at all.

For example, the LNT in Figures 6.1(b) matches the loop structure with disk access operations in Figures 6.2(b). Here, loop $k$ is below disk access operation $C$. In order to
simplify the description, we use the name of disk array to represent the corresponding disk access operation. For instance, a disk access operation is labeled as "D" if it accesses elements of disk array D.

6.2.3 Fusion Structure and Disk I/O Cost

Each disk-resident array has a corresponding disk access operation in the loop structure. The following terms is defined to describe the relationship of loops and disk access operations inside it.

Figure 6.1: Corresponding LNT for fusion structure in Figure 6.2
for \( j \)
\[
\begin{align*}
\text{for } h, k & \quad t2(k) + = C(j, h) \times \text{buf}D(h, k) \\
\text{for } i & \quad t1+ = A(i, l) \times B(l, j) \\
\text{for } k & \quad E(i, k) + = t1 \times t2(k)
\end{align*}
\]

(b) Fusion Structure with Disk I/O

Read disk \( A(i, l) \) to \( \text{buf}A(i, l) \)
Read disk \( D(h, k) \) to \( \text{buf}D(h, k) \)

\[
\begin{align*}
\text{for } j, h & \quad \text{Read disk } C(j, h) \text{ to buf}C \\
& \quad \text{for } k \quad t2(k, j) + = \text{buf}C \times \text{buf}D(h, k) \\
& \quad \text{for } i \quad \text{Read disk } B(l, j) \text{ to buf}B \\
& \quad \quad \text{for } j \quad t1(j) + = \text{buf}A(i, l) \times \text{buf}B(j) \\
& \quad \quad \quad \text{for } k, j \quad \text{buf}E(i, k) + = t1(j) \times t2(j, k)
\end{align*}
\]

Write \( \text{buf}E(i, k) \) to disk \( E(i, k) \)

(c) After fissioning loop \( j \) below \( B \)

Figure 6.2: Fusion Structures
Definition 3  If a disk access operation $D$ is located inside loop $i$, but $i$ is not a dimension of the operated disk-resident array. We call loop $i$ as a **redundant loop** of disk access operation $D$.

Definition 4  If a disk access operation $D$ is located inside loop $i$, and $i$ is a dimension of the operated disk-resident array. We call loop $i$ as an **effective loop** of disk access operation $D$.

Each disk access operation requires a local buffer in memory to hold data transferred from/to disk. The size of local buffer for disk access operation $D$ is denoted as $D\.localsize$.

The total memory usage for a loop structure is the summation of the memory required for in-memory intermediate arrays and the memory required for local buffers used by disk access operations. Loop fusion can reduce the memory requirement of intermediate arrays. The original size of an intermediate array without fusion is denoted as $array\.size$, while the fused size is denoted as $array\.fsize$.

On the other side, loop fusion will impose constraints on loop order, and limit the applicability of loop permutation in the fused loop structure. The maximally-fused fusion structure can achieve minimal memory usage for intermediate arrays. However, fusing loops too aggressively may force some disk access operations to nest inside **redundant loops**, thus, lead to many redundant disk access operations. In this situation, the amount of disk I/O volume could be much larger than the total volume of the data on disk. The example in Figures 6.3 illustrates different effects of loop fusion on memory usage and disk I/O cost. Since the used data sets are usually very large in scientific computations, we assume here that the disk I/O cost is dominated by the volume of data movement, not the number of data movement. The disk I/O cost required by a disk access operation $D$, denoted as $D\.diskcost$, is defined as the total amount of data transferred between disk and
memory. The size of the disk array being operated is denoted as $D\cdot\text{disksize}$. Because each element in the disk array will be loaded at least once, the minimal disk I/O cost of $D$ is $D\cdot\text{disksize}$.

Consider the sequence of tensor contractions in equations 6.2, 6.3, and 6.4, we assume that $i:\text{range} = 20$, $j:\text{range} = 16K$, $k:\text{range} = 20$, $l:\text{range} = 8K$ and $h:\text{range} = 8K$. The available memory during execution is $320K$. There are three programs in Figures 6.3 to implement the contraction sequence. We start with the naive implementation in Figures 6.3(a), which has three independent loop nests $La$, $Lb$ and $Lc$ corresponding to Eqn. 6.2, Eqn. 6.3, and Eqn. 6.4 respectively. The original size of both intermediate array $t_1$ and $t_2$ is $320K$, then both of them are too large to stay in memory. There are totally seven disk-resident arrays, $A$, $B$, $C$, $D$, $E$, $t_1$ and $t_2$. As shown in Figures 6.3(a), if disk access operations are inserted into the program properly, within the available memory, each disk-resident array can be read to/write from disk only once. The total disk I/O cost will be $257M$.

If we fuse $La$ and $Lc$ by loop $i$ and $j$, and then fuse the result with $Lb$ by loop $j$, we will get the loop structure in Figures 6.3(b). In Figures 6.3(b), the intermediate array $t_1$ is reduced to a scalar and $t_2$ has only one dimension $k$. Both of them can be put into memory. However, in this fusion structure, if we don’t want any redundant data movement, the local buffers of disk access operations require more than $328K$ memory, exceeding the current memory limit. As a result, one of those disk access operations must be put inside redundant loops. Without losing generality, we assume that the disk access operation $B$ is inserted into loop $i$, $j$ and $l$, then we have $B\cdot\text{diskcost} = i\cdot\text{range} \times l\cdot\text{range} \times j\cdot\text{range} = 2560M$. The total disk cost will be more than $2688M$. Comparing with the program in Figures 6.3(a),
even through the disk cost involved by intermediate arrays is eliminated, the total disk cost is increased.

The third implementation in this example can not only keep all intermediate arrays in memory, but also avoid redundant disk movements of input and output arrays. In Figures 6.3(c), loop $j$ is fused between $L_a, L_b$ and $L_c$. The $j$ dimension in $t_1$ is eliminated, then $t_1.fsize = 20$. In the same manner, we have $t_2.fsize = 20$. Both of them can fit in memory. If disk access operations are placed in the loop structure as shown in Figures 6.3(c), each disk array will be read/write only once. The memory required for local buffer is about $320M$, less than the available physical memory. The total disk I/O cost will be $256M$, which is the minimal cost we can achieve for this computation. Compared with the fusion structure in Figures 6.3(b), here loop $i$ is not fused on $t_1$, so loop $i$ and $l$ are permutable. The disk access operation $B$ can be insert inside loop $l$ and $j$ and outside of loop $i$. Then, array $B$ will be read only once with $B.localsize = 1$. In the fusion structure of Figures 6.3(b), $B$ can only achieve the same disk I/O cost with $B.localsize = 8K$. Accordingly, we can say that loop fusion will increase the memory usage of disk access operations.

Note here that, loop $i$ is a redundant loop of the disk access operation $B$. Fusing loop $i$ above $l$ causes array $B$ to be read into memory multiple time. If the $i$ is fissioned below $B$ in Figures 6.3(b), we will get the exactly same loop structure as Figures 6.3(c). Thus, we have the following challenge: given a maximally-fused fusion structure with proper disk I/O placement, how can we fission some redundant loops thereby achieve the minimal disk I/O cost? We will address this challenge in the next section.
Read disk $A(i, l)$ to $buf\, A(i, l)$
for $j$
  [Read array $B(l, j)$ to $buf\, B(l)$
   for $i, l$
   [buf1($i$)+ = $buf\, A(i, l) \times buf\, B(l)$
   Write buf1($i$) to disk $t1(i, j)$

Read array $D(h, k)$ to $buf\, D(h, k)$
for $j$
  [Read array $C(j, h)$ to $buf\, C(h)$
   for $i, k$
   [buf2($k$)+ = $buf\, C(h) \times buf\, D(h, k)$
   Write buf2($k$) to array $t2(j, k)$

for $j$
  [Read array $t1(i, j)$ to $buf\, t1(i)$
  Read array $t2(j, k)$ to $buf\, t2(k)$
   for $j, k$
   [bufE($i, k$)+ = buf1($i$)$ \times t2(k)$
   Write bufE($i, k$) to array $E(i, k)$

Write bufE($i, k$) to disk $E(i, k)$

(a): Not fused

Read disk $A(i, l)$ to $buf\, A(i, l)$
Read disk $D(h, k)$ to $buf\, D(h, k)$
for $j$
  for $h$
    [Read disk $C(j, h)$ to $buf\, C$
     for $k$
     [t2($k$)+ = $buf\, C \times buf\, D(h, k)$

    for $l$
      [Read disk $B(l, j)$ to $buf\, B$
       for $i$
       [buf1($i$)+ = $buf\, A(i, l) \times buf\, B$

       for $i, k$
       [bufE($i, k$)+ = buf1($i$)$ \times t2(k)$

Write bufE($i, k$) to disk $E(i, k)$

(b): Fully Fused

Read disk $A(i, l)$ to $buf\, A(i, l)$
Read disk $D(h, k)$ to $buf\, D(h, k)$
for $j$
  for $h$
    [Read disk $C(j, h)$ to $buf\, C$
     for $k$
     [t2($k$)+ = $buf\, C \times buf\, D(h, k)$

    for $l$
      [Read disk $B(l, j)$ to $buf\, B$
       for $i$
       [buf1($i$)+ = $buf\, A(i, l) \times buf\, B$

       for $i, k$
       [bufE($i, k$)+ = buf1($i$)$ \times t2(k)$

Write bufE($i, k$) to disk $E(i, k)$

(c): Partially Fused

Figure 6.3: Three implementations of the contraction sequence: Eqn. 6.2, 6.3, and 6.4
6.3 Optimizing Fusion Structure for a Single Disk Access Operation

In the last section, we explained how loop fusion can reduce the memory requirement of intermediate arrays, but on the other side, increase the memory usage of disk access operations. For global optimization, we need to evaluate a fusion considering both sides. This section addresses the problem of transforming a loop structure to the optimal form for a disk access operation under a memory constraint. The algorithm and the analysis model proposed in this section are not limited to tensor contractions, but are general for a range of out-of-core computations expressible as imperfectly nested loop structures.

6.3.1 Candidate Location Sequence

For a disk access operation, its location in loop structure will determine the size of its local buffer and the volume of data movement it involved. For instance, if the disk access operation is located inside a redundant loop \( j \), its disk I/O cost will be increased by \( j \cdot \text{range} \) times. Otherwise, if it is inside an effective loop \( i \), the size of its local buffer will be decreased by \( i \cdot \text{range} \) times. If loop \( j \) is enclosing loop \( i \) in a loop structure, this disk access operation is either placed inside \( j \) and \( i \) with higher disk I/O cost, or out of them with higher memory cost.

The location of a disk access operations in a loop structure can be represented by a sequence of surrounding loop indices. As an example, in Figures 6.3(b), the location of the statement “Read disk \( B(l, j) \) to bufB” is \( \{j, i, l\} \). Given two different locations \( L_1 \) and \( L_2 \), if \( L_1 \) is a prefix of \( L_2 \), we said that location \( L_1 \) is higher than \( L_2 \). The disk I/O cost of the disk access operation \( D \) at the location \( L \) is denoted as \( \text{DiskCost}(D, L) \). Also, the size of local buffer is denoted as \( \text{BufSize}(D, L) \). If \( \text{DiskCost}(D, L_1) < \text{DiskCost}(D, L_2) \) and \( \text{BufSize}(D, L_1) < \text{BufSize}(D, L_2) \), then \( L_2 \) is inferior to \( L_1 \) and can be ignored as a
sub-optimal choice. A disk access operation can be inserted at any place in a loop structure before the target array is consumed. However, if a location is immediately surrounded by a redundant loop, it is inferior to the location above the redundant loop, and can be ignored for further consideration. Many locations are pruned by this rule; the remained are called candidate locations. The set of candidate locations of a disk access operation can be ordered from high to low. There are a set of effective loops followed by a set of redundant loops between two adjacent locations. The total redundant loops of the location $L_k$ is denoted as $T J_k$. The total effective loops of the location $L_k$ is denoted as $T I_k$. The disk I/O cost and memory cost of disk access operation $D$ at location $L_k$ can be derived by the following equations:

$$DiskCost(D, L_k) = D.disksize \times T J_k.range \quad (6.5)$$

$$BufSize(D, L_k) = D.disksize/T I_k.range \quad (6.6)$$

Obviously, a higher location requires a larger local buffer, but involves lower disk I/O cost than a lower location.

As an example to illustrate the above concepts, we use the loop structure shown in Figures 6.3(b), and focus on the candidate locations of disk access operation $B$. The location sequence, $\{Location1, Location2\}$, is shown in Figures 6.4. Here, $TI_1 = \{j\}$, $T J_1 = \emptyset, TI_2 = \{l, j\}$ and $T J_2 = \{i\}$. The disk cost and memory cost for these two locations are listed below:

$$DiskCost(B, Location1) = B.disksize$$

$$BufSize(B, Location1) = B.disksize/j.range$$

$$DiskCost(B, Location2) = B.disksize \times i.range$$

$$BufSize(B, Location2) = 1$$
for \( j \)
for \( h \)
for \( k \)
\[
t2(k) + = C(j, h) \times D(h, k)
\]

Location1: Read disk \( B(l, j) \) to \( bufB(l) \)
for \( i \)
for \( l \)
Location2: Read disk \( B(l, j) \) to \( bufB \)
\[
t1+ = A(i, l) \times B
\]
for \( k \)
\[
E(i, k) + = t1 \times t2(k)
\]

Figure 6.4: The location sequence for “Read B” statement

### 6.3.2 Weight of Loop Fission

In section 6.2.3, we proposed to use loop fission to transform a maximally-fused fusion structure to reduce the disk I/O cost and memory usage of a disk access operation. Here, we will model and analyze the effect of loop fission based on the loop nesting tree defined in section 6.2.2.

Loop fission is the opposite of loop fusion, in which a loop is split into several parts in parallel. In a loop nesting tree, if a loop is fissioned, the corresponding loop index will be removed from the original node and added into all children of the original node. The fission can be performed recursively until the fissioned loop reaching the leaf nodes. Since we are interested in the effect of fission on disk access operations, we use the location of disk access operation as coordinates to specify an ending point of a loop fission, such as loop \( i \) is fissioned below disk access operation \( D \), or below the location \( L \) of disk access operation \( D \). Considering a fission of loop \( i \), let’s assume that, before fission, loop \( i \) is located at node
after fission, it is split and sinks into nodes $N_1, N_2, ..., N_k$. The consequence of such a loop fission will be discussed in the following three cases:

1. Before fission, loop $i$ is a redundant loop of the disk access operation $D$. After fission, if loop $i$ is not out of $D$, the disk I/O cost of $D$ will be decreased $i.range$ times.

2. Before fission, loop $i$ is an effective loop of the disk access operation $D$. After fission, if loop $i$ is not out of $D$, the local buffer size of $D$ will be increased $i.range$ times.

3. Before fission, loop $i$ is a fused loop of intermediate array $t$. After fission, if $i$ is not fused in $t$, the size of $t$ will be increased $i.range$ times.

In a loop nesting tree, all disk access operations at paths from $S$ to $N_1, N_2, ..., N_k$ will be affected by the fission. In addition, all intermediate arrays in the following set will be expanded $i.range$ times larger after the fission.

$$S.fusedarray - N_1.fusedarrays - ... - N_k.fusedarrays$$ (6.7)

In summary, loop fission will increase the memory usage, but reduce the amount of redundant disk access operations. To illustrate the effect of loop fission, we perform loop fission on the loop structure in Figures 6.3(b). If loop $j$ is fissioned below the operation “reading B”, then we will get a new fusion structure shown in Figures 6.3(c), whose corresponding LNT is shown in Figures 6.2(c). We can see that, after fission the $j$-dimension in intermediate arrays $t_1$ and $t_2$ are extended, as well as the local buffer of disk access operation $B$. We denote the amount of increased memory of fissioning a set of loop $I$ below the location $L$ as $weight(I,L)$. The algorithm to calculate $weight(I, L)$ in fusion structure $F$ is presented in Algorithm 5. After loop fission, if the total memory usage doesn’t exceed the current memory limit, we say that the fission is legal.
**Algorithm 5** Weight(I,L,F)

//I: The set of fission loops;  
//L: The location in the loop structure;  
//F: The LNT representing the loop structure;  

`ExtendedArrays = ∅;`  
`ExtendedBufs = ∅;`  

**for** each loop `i` in `I` **do**  
  `s` = The node including `i` in `F`;  
  Recursively split and sink `i` in `F` until below location `L`;  
  `T` = The set of nodes including `i` after it broke and sinked;  
  `DiskIO` = The set of disk access operations on paths from `s` to each node in `T`;  

`Intermediates = ∅;`  
**for** each node `t` in `T` **do**  
  `Intermediates = Intermediates + t.fusedarrays;`  
**end for**  
`Intermediates = s.fusedarrays − Intermediates;`  
**for** each array `a` in `Intermediates` **do**  
  **if** `a ∈ ExtendedArrays` **then**  
    `a.newsize = a.newsize × i.range;`  
  **else**  
    `a.newsize = a.fsize × i.range;`  
    `ExtendedArrays = ExtendedArrays + a;`  
  **end if**  
**end for**  

**for** each disk access operation `d` in `DiskIO` **do**  
  **if** `d ∈ ExtendedBufs` **then**  
    `d.newlocalsize = d.newlocalsize × i.range;`  
  **else**  
    `d.newlocalsize = d.localsize × i.range;`  
    `ExtendedBufs = ExtendedBufs + d;`  
  **end if**  
**end for**  

`M = 0;`  
**for** each array `a` in `ExtendedArrays` **do**  
  `M = M + a.newsize − a.fsize;`  
**end for**  
**for** each disk access operation `d` in `ExtendedBufs` **do**  
  `M = M + d.newlocalsize − dlocalsize;`  
**end for**  
return `M;`
6.3.3 Optimizing Loop Structure for a Single Disk Access Operation

For a loop structure $F$, a disk access operation $D$ and the amount of available memory $M$, we present an algorithm to determine:

- the proper loop fission to transform $F$; and

- the proper location to insert $D$ into the transformed loop structure;

such that the disk I/O cost of $D$ is minimized under the memory constraint. Note that here, in the next step, we will use the algorithm for a more complicated case, in which a loop structure with multiple disk access operations is involved. Therefore, the solution just with minimal disk cost is not good enough for future needs. Besides reducing the disk I/O cost, we also want to reduce the memory usage. If there are several solutions with the minimal disk I/O cost, the one with the minimal memory requirement should be chosen.

The algorithm is described as follows. First, the candidate location sequence of disk access operation $D$ in loop structure $F$ is generated. Then, we traverse the sequence from head to tail. For each location $L_k$ in the sequence, the size of required local buffer can be calculated by equation 6.6. If it is larger than the available memory, then disk access operation $D$ can’t be placed at location $L_k$, and $L_k$ will be ignored. Otherwise, we will try to fission redundant loops of $D$ below $L_k$ to reduce the disk I/O cost. For each subset $I$ of redundant loops, we perform loop fission of $I$ below $L_k$. The amount of increased memory can be calculated using Algorithm 5. If the total amount of increased memory by fissioning $I$ and local buffer required by $D$ is beyond the available memory, the fissioning $I$ is illegal. Otherwise, we calculate the disk I/O cost after fissioning $I$. The minimal disk I/O cost of $D$ at $L_k$ will be determined after checking all possible fissions of redundant loops. The loop fission leading to the minimal disk cost is denoted as $Fission(L_k)$. The
memory requirement for achieving the minimal disk I/O cost is the summation of the local buffer and the increased memory by $Fission(L_k)$. After going over the candidate location sequence, location $L$ with the minimal disk I/O cost will be selected as the optimal location. $Fission(L_k)$ will be used to transform $F$ to the optimal fusion structure for $D$. The algorithm to find the optimal fusion structure and location is presented in Algorithm 6.

In the algorithm above, the number of possible loop fissions are very large. The search procedure can be made more efficient using the following pruning rules.

**Lemma 3** Given two loop sets $t_{j_1}$ and $t_{j_2}$ and location $L_k$ of disk access operation $D$, if $t_{j_1} \subseteq t_{j_2}$, then $weight(t_{j_1}, L_k) < weight(t_{j_2}, L_k)$.

**Proof 6.3.1** Trival, hence omitted.

**Lemma 4** Given two locations $L_1$ and $L_2$ of disk access operation $D$ and loop set $st_j$, if $L_1$ is higher than $L_2$, then $weight(st_j, L_1) < weight(st_j, L_2)$.

**Proof 6.3.2** Trival, hence omitted.

**Theorem 1** Given a location $L_k$ of disk access operation $D$, if redundant loop set $st_j$ can’t be fissioned below $L_k$, then for each loop set $st_j'$ with $st_j \subseteq st_j'$, loop fission for $st_j'$ below $L_k$ is illegal.

**Proof 6.3.3** It follows from Lemma 3.

**Theorem 2** Given two locations $L_1$ and $L_2$ of disk access operation $D$ and two redundant loop sets $t_{j_1}$ and $t_{j_2}$, we assume that $L_1$ is higher than $L_2$ and $t_{j_1} \subseteq t_{j_2}$. If $weight(t_{j_1}, L_1) > M$, loop fission for $t_{j_2}$ below $L_2$ is illegal.

**Proof 6.3.4** It follows from Lemma 3 and Lemma 4.
Algorithm 6 SingleDiskIO(D,F,M,OptMemCost,OptDiskCost)

//D: The disk access operation
//F: The LNT representing the current fusion structure
//M: The available memory space
//OptMemCost: The actual memory usage
//OptDiskCost: The minimal disk I/O cost

OptDiskCost = \infty
OptMemCost = \infty

Generate the candidate location sequence CLS of D in F

for each location L_k in CLS do
    if BufSize(D,L_k) < M then
        TJ = The set of redundant loops of D at L
        LDiskCost = DiskCost(D,L_k)
        FissionSet = {} 
        for each subset stj in TJ do
            IncreasedMem = Weight(stj,L_k,F)
            if BufSize(D,L_k) + IncreasedMem < M then
                FDiskCost = DiskCost(D,L_k)/stj.range
                if FDiskCost < LDiskCost then
                    LDiskCost = FDiskCost
                    FissionSet = stj
                    LMemCost = BufSize(D,L_k) + IncreasedMem
                end if
            end if
        end for
        if LDiskCost < OptDiskCost then
            OptDiskCost = LDiskCost
            OptMemCost = LMemCost
            OptLoc = L_k
            OptFission = FissionSet
        end if
        if LDiskCost = OptDiskCost and LMemCost < OptMemCost then
            OptDiskCost = LDiskCost
            OptMemCost = LMemCost
            OptLoc = L_k
            OptFission = FissionSet
        end if
    end if
end for

Insert D to location OptLoc in F
Transform F by fissioning loops in OptFission below OptLoc
If a loop fission can satisfy the conditions specified in the above theorems, it will be ignored as illegal fission directly.

6.4 Optimizing Fusion Structure for a Set of Disk Access Operation

6.4.1 Algorithm

The last section presents an algorithm to insert a disk access operation in a loop structure and transform it to optimal format to minimize the disk I/O cost under a memory constraint. In extending the algorithm to solve the same problem for the case including multiple disk access operations, we encounter the following two difficulties.

- **Incompatible optimal loop structures for different disk access operations**
  
  Generally, the optimal loop structure for a disk access operation $D$ would have effective loops of $D$ as the outermost loop. Thus, the optimal loop structures for different disk access operations may be incompatible with each other. The loop structure that can reduce the disk I/O cost for one disk access operation may result in a lot of redundant data movements for another disk access operation. The transformation of a loop structure to optimize the overall disk I/O performance is a complicated problem.

- **Memory allocation among various disk access operations**
  
  For different candidate locations of a disk access operation, a higher location in the loop nest requires a larger local buffer, but involves less disk I/O cost; while a lower location needs a smaller local buffer, but has more redundant data movements. In summary, the disk I/O cost for a disk access operation can be reduced at the expense of increased memory usage. Theoretically, each disk-resident array can be loaded from/to disk only once, if the local buffer for the corresponding disk access operation is big enough. If there is insufficient memory to perform all disk access operations
without redundant data movements, we need to allocate available memory carefully among the disk access operations.

Conceptually, we need to search all possible loop structures, along with disk I/O placements to obtain the optimal solution. Due to the huge search space, we don’t use a brute force search, but provide an efficient heuristic. The basic idea of the algorithm relies on the following observation: larger disk-resident arrays have greater impact on the overall disk I/O performance than smaller arrays. Hence, disk access operations over larger disk resident arrays are given higher priority than those involving smaller disk resident arrays. When we adjust a loop structure, we first consider its effect on disk operations with larger arrays.

Algorithm 7 shows the algorithm to determine the proper loop fission along with disk I/O placements for a given fusion structure \( F \) and a set of disk access operations \( DS \), to reduce the total disk I/O cost subject to limited memory space \( M \). First, we sort the set of given disk access operations by the size of disk resident arrays – from largest to smallest. The second step is to iteratively take the first unhandled disk access operation \( D_1 \) from \( DS \), insert it into the fusion structure \( F \), and transform \( F \) to the optimal format for it. Since all intermediate arrays in the fusion structure are retained in memory, the available memory space is divided into two parts. \( M_{\text{inter}} \) denotes buffers used for intermediate arrays, while \( M_{\text{disk}} \) denotes local buffers for disk access operations. The size of memory allocated to \( D_1 \) is proportional to the ratio of disk array \( D_1 \) to the total unhandled disk arrays left in \( DS \).

\[
M_1 = \frac{D_1_{\text{disksize}} \times M_{\text{disk}}}{\sum_{D_k \in DS} D_k_{\text{disksize}}} \quad (6.8)
\]

In each iteration, the loop structure \( F \) is transformed to the optimal format for \( D_1 \), subject to use of no more than \( M_1 \) memory, by Algorithm 6. The transformed fusion structure is
used as the input loop structure at the next iteration. The actual memory used after the transformation (including local buffer of $D_1$) is removed from available memory space $M_{disk}$. The details are shown in Algorithm 7.

**Algorithm 7 MultiDiskIO(DS,F,M,OptDiskCost)**

//DS: The set of disk access operations;
//F: The LNT representing the input loop structure;
//M: The available memory space;
//OptDiskCost: The total disk I/O cost for all disk access operations;

$OptDiskCost = 0$;
$DiskSize = 0$;

for each disk access operation $D_k$ in DS do

$DiskSize = DiskSize + D_k.disksize$;

end for

$M_{disk} = M$;

for each intermediate array $t_k$ in F do

$M_{disk} = M_{disk} - t_k.fsize$;

end for

Sort disk access operations in DS by size of their operated disk arrays;

for each disk access operation $D_k$ in DS do

$M_1 = (D_k.disksize \times M_{disk})/DiskSize$;
$DiskSize = DiskSize - D_k.disksize$;

call SingleDiskIO($D_k, F, M_1, ActualMemCost, DiskCost$);

$M_{disk} = M_{disk} - ActualMemCost$;
$OptDiskCost = OptDiskCost + DiskCost$;

end for

### 6.4.2 Integrated Framework

In section 5.2.1, we propose an integrated optimization framework to find the optimal loop structure, along with disk I/O placements for a set of tensor contractions. Due to the high time complexity, it may be too costly to apply the approach to long and complicated contraction sequences. The algorithm presented above uses heuristic to reduce
computational complexity. If we exploit it to replace the exhaustive search method in the framework, the optimization process can be completed in reduced time. The changed optimization framework is presented as follows:

1. **Operation Tree Partitioning:** In this step, we divide the original operation tree into several **fused subtrees** by identifying **cut-points**. The optimal loop structures for different subtrees are independent of each other, and can be determined separately.

2. **Loop Structures Enumeration:** For each fused sub-tree, we produce a set of maximally-fused fusion structures, represented by **loop nesting trees**.

3. **Loop Structure Transformation:** For each maximally-fused fusion structure, we insert necessary disk access operations and transform it to the optimal form to reduce the disk I/O cost. The transformation can be done by Algorithm 7.

4. **Tile Size Selection:** After determining the loop structure and disk I/O placements, the I/O cost of the program can be formulated as a non-linear optimization problem in terms of the tile sizes. The tile sizes that minimize the disk I/O cost are calculated using a general-purpose non-linear optimization solver.

5. **Code Generation:** We calculate the disk access cost for each obtained solution, choose the one with the minimal disk I/O cost, and generate code for it.

### 6.5 Experiments

The experiments to evaluate our algorithm were taken on an Itanium-2 cluster at the Ohio Supercomputer Center. The configuration of the cluster is shown in Table 3.5. All
the programs were compiled with the Intel Itanium Fortran Compiler for Linux. The in-memory tensor contractions were calculated by the BLAS-3 routines in the Intel math kernel library.

The experiment utilizes the 4index-transform expression in 1.1. At first, we pick up a maximally-fused fusion structure for the contraction sequence as shown in Figures 6.5, in which, all intermediate arrays are fused and retained in memory. Then, two versions of implementations are generated by this loop structure. The first one uses the original loop structure directly and has disk access operations inserted manually. For the second program, the algorithm presented in section 6.4.1 were employed to transform the original fusion structure and insert disk access operations in it. We compare the disk I/O time and total execution time of the two implementations. The result is shown in Table 6.1. We can see that our algorithm can reduce the disk I/O cost for a given loop structure and improve the total performance.
### Table 6.1: I/O time and total execution time for 4index-transform: maximally-fused loop structure versus optimized loop structure

<table>
<thead>
<tr>
<th>N</th>
<th>V</th>
<th>Disk I/O (sec)</th>
<th>Total Exec. (sec)</th>
<th>Disk I/O (sec)</th>
<th>Total Exec. (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>120</td>
<td>229</td>
<td>302</td>
<td>229</td>
<td>319</td>
</tr>
<tr>
<td>180</td>
<td>120</td>
<td>400</td>
<td>528</td>
<td>333</td>
<td>453</td>
</tr>
<tr>
<td>200</td>
<td>120</td>
<td>811</td>
<td>975</td>
<td>561</td>
<td>733</td>
</tr>
<tr>
<td>220</td>
<td>120</td>
<td>1511</td>
<td>1756</td>
<td>985</td>
<td>1210</td>
</tr>
</tbody>
</table>

In addition, to evaluate the effectiveness of our algorithm in the TCE system, we generate codes for the 4index-transform by three different methods respectively, 1) the exhaustive searching; 2) the optimization framework described in section 6.4.2 and; 3) the equi-tile-size approach. The I/O time and total execution time of these three implementations are compared in Table 6.2. From the experimental result, we observe that the new optimization approach can work almost as good as the exhaustive searching, while much better than other alternative approaches.

### Table 6.2: I/O time and total execution time for three implementations of 4index-transform

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>120</td>
<td>226</td>
<td>317</td>
<td>370</td>
<td>461</td>
<td>229</td>
<td>319</td>
</tr>
<tr>
<td>180</td>
<td>120</td>
<td>336</td>
<td>464</td>
<td>522</td>
<td>650</td>
<td>333</td>
<td>453</td>
</tr>
<tr>
<td>200</td>
<td>120</td>
<td>403</td>
<td>575</td>
<td>691</td>
<td>863</td>
<td>441</td>
<td>615</td>
</tr>
<tr>
<td>220</td>
<td>120</td>
<td>542</td>
<td>769</td>
<td>1020</td>
<td>1247</td>
<td>571</td>
<td>798</td>
</tr>
</tbody>
</table>

6.6 Conclusion

In this chapter, we provided a model to describe the relationship between loop structure and disk I/O placements, and show their effect on disk I/O cost and memory cost. Based on
the analysis model, we presented an algorithm to transform a given loop structure and insert necessary disk access operations in it properly, thereby reduce the disk I/O cost under a memory constraint. This algorithm can also be used in the a domain-specific compiler system, incorporated with other optimization strategies, to generate the optimal loop structure with minimal disk I/O cost for a set of tensor contractions. Experimental results were provided to demonstrate the effectiveness of the algorithm. Though it is a heuristic algorithm, it can achieve the near optimal solution in the experiments.
CHAPTER 7

OPTIMIZATION FOR TENSOR CONTRACTIONS WITH
SYMMETRY PROPERTIES

In our previous work, we assume that all tensors used in the class of domain-specific
computations are relatively dense. On the contrary, most of them have various symmetry
properties, such as spatial symmetry, permutation symmetry, and spin symmetry. Clearly,
these symmetry properties will affect the detailed structure of tensors, their natural block
structure, and number of unique elements. Taking advantage of these symmetry proper-
ties can significantly reduce storage and computation requirements. But, they also restrict
the effectiveness and capability of some optimization strategies. In this section, we dis-
cuss characteristics of different symmetries, and how to adopt general loop transformation
strategies used in our current optimization framework for domain-specific computations
with symmetry properties.

7.1 Spatial Symmetry

In this particular class of computations being considered, most tensors have spatial
symmetry property. Generally, tensors with spatial symmetry reflect geometric symmetry
properties of the molecule on which the calculation is being performed. Spatial symmetry
gives tensors a block structure in each index and allows the calculation to be reduced to
for $iB$, $jB$

  If block $C_{iB,jB}$ is symmetry-unique

  for $kB$

    If blocks $A_{iB,kB}$ and $B_{kB,jB}$ are symmetry-unique

    for $i$, $j$, $k$ in $iB$, $jB$, $kB$

    $C_{iB,jB} = A_{iB,kB} \times B_{kB,jB}$

Figure 7.1: Pseudo code of tensor contraction with spatial symmetry property

only the *symmetry-unique* blocks. The number of blocks and their sizes are only known at run-time. Thus, we need to add *identifying statements* into generated code to verify whether a block is symmetry-unique or not. A sample tensor contraction, $C(i, j) = \sum_k A(i, k) \times B(k, j)$, will be translated into code shown in Figure 7.1, where $iB$, $jB$ and $kB$ represent symmetry blocks in $i$, $j$ and $k$ dimensions, respectively. Such loops will be referred as *blocking loops* later.

For the fusion in dense case, merging a pair of loops between producer and consumer will totally eliminate corresponding index of the intermediate tensor. But, fusion in spatial symmetry case can not achieve the same benefit. The *identifying statements* will split the origin perfect loop nest loops to an imperfect loop nest, thereby, constraint the use of loop fusion to only *blocking loops*.

For example, consider the two contractions below:

$$T1(i, j) = \sum_k A(i, k) \times B(k, j)$$

$$D(i, l) = \sum_j T1(i, j) \times C(j, l)$$

If all tensors used in the contraction sequence are dense, then loops that produce $T1$ and consume $T1$ can be fully fused to eliminate all explicit indices in $T1$ and reduce it to a scalar. The pseudo code after fusion is shown in Figure 7.2(a). On the other side, if all these
tensors are spatial symmetry, only the blocking loops can be fused. Then, the intermediate result will be reduced to a block instead of a scalar, such as $T_{1iB,jB}$ in Figure 7.2(b). The amount of memory requirement for fused $T1$ would be the size of the largest spatial block in the original $T1$. Since shapes of different spatial blocks vary significantly, the size of the maximal block may be close to that of the original tensor.

In order to take full advantage of loop fusion, we divide a dimension into multiple smaller, but still sizable, units, called bricks. For example, a way to break a two-dimensional spatial symmetry tensor into bricks is illustrated in Figure 7.3(a), where shadowed boxes represent symmetry-unique blocks. Each brick would have the same size except those located at the boundary of blocks. Brick-structure is used instead of block-structure in spatial symmetry tensors. Thereby, we will replace blocking loops by bricking loops in resulting code, and fuse bricking loops to reduce the intermediate result to a brick, which is much smaller than a block. The brick-structure provides a generic template that allows to apply the loop transformation strategies used in dense case to spatial case. Although it will cause redundant 'identifying’ operations for each brick, the advantage is much larger than
the disadvantage. For each loop transformation used in our integrated optimization framework, brick will be treated as the smallest element in a spatial tensor. The corresponding pseudo code to apply fusion and tiling on the above contraction sequence is presented in Figure 7.4, where $iB, jB$ and $kB$ symbolize bricking loops. We need to mention that, the 'identifying' operation becomes a little more complicated in the tiled program. A tile can include both symmetry-unique bricks and non-symmetry-unique bricks. We define a tile without symmetry-unique brick as empty. The logical view of tiling a brick-structure spatial tensor is shown in Figure 7.3(b).

![Block-Brick format vs Tile-Brick format](image)

(a) Block-Brick format  
(b) Tile-Brick format

Figure 7.3: The logical view of bricking and tiling a spatial symmetry tensor
for \( iB, jB \)

If brick \( T_{iB,jB} \) is symmetry-unique

for \( kB \)

If bricks \( A_{iB,kB} \) and \( B_{kB,jB} \) are symmetry-unique

\[
T_{iB,jB} = A_{iB,kB} \times B_{kB,jB}
\]

for \( lB \)

If bricks \( D_{iB,lB} \) and \( C_{jB,lB} \) are symmetry-unique

\[
D_{iB,lB} = T_{iB,jB} \times C_{jB,lB}
\]

(a) Apply fusion in brick-structure

for \( iT, jT \)

If tile \( T_{iT,jT} \) is not empty

for \( kT \)

If tiles \( A_{iT,kT} \) and \( B_{kT,jT} \) aren't empty

for \( iB, jB \) in \( iT, jT \)

If brick \( T_{iT,jT} \) is symmetry-unique

for \( kB \) in \( kT \)

If bricks \( A_{iB,kB} \) and \( B_{kB,jB} \) are symmetry-unique

\[
T_{iT,jT} = A_{iB,kB} \times B_{kB,jB}
\]

for \( lT \)

If tiles \( D_{iT,lT} \) and \( C_{jT,lT} \) aren't empty

for \( iB, jT \) in \( iT, jT \)

If brick \( T_{iT,jT} \) is symmetry-unique

for \( lB \) in \( lT \)

If bricks \( D_{iB,lB} \) and \( C_{jB,lB} \) are symmetry-unique

\[
D_{iB,lB} = T_{iT,jT} \times C_{jB,lB}
\]

(b) Apply tiling in brick structure

Figure 7.4: Apply loop transformation strategies used in dense case to spatial case
7.2 Permutation Symmetry

In the field of electronic structure calculation, tensors may be antisymmetric on the interchange of certain indices. It defines equivalence within a sign of certain portions of the tensor.

The permutation symmetry property of a tensor can be easily described by division of subscript indices into *antisymmetry groups*, so that switching any two indices within the group results only in a sign flip of the tensor. Usually, we separate groups of antisymmetric indices by commas. For example, $T_{ab,c,d}$ is antisymmetric in the first two dimensions. Here, $ab$ is an antisymmetry group, and $T_{a,b,c,d} = -T_{b,a,c,d}$ for all $a$ and $b$.

For a permutation symmetric tensor, only a fraction of elements need to be calculated and stored. That can reduce the computation and storage requirements. On the other side, if these partial stored permutation symmetric tensors are used in tensor contractions, it may change the element access order and impose restrictions on potential loop fusions.

For a better understanding of the relationship between permutation symmetry and program transformation, let’s first define some special characteristics of permutation symmetric contractions.

- **Antisymmetrization**: Given a tensor contraction, if an antisymmetry group in the result tensor is not contained in input tensors, additional operations are required more than contraction.

  Consider the example $C_{ab} = \sum_i A_{a,i} \times B_{i,b}$, the direct result of the contraction does not have antisymmetry between dimension $a$ and $b$. To achieve antisymmetry in the result, after contraction we add together all possible permutations of the result to
itself with appropriate signs as following:

\[ T_{1_{a,b}} = \sum_i A_{a,i} \times B_{i,b} \]
\[ C_{ab} = \frac{1}{2} \times (T_{1_{a,b}} - T_{1_{b,a}}) \]

The first step is contraction, and the second step is called \textit{antisymmetrization}. In order to eliminate the intermediate array, we merge two steps together and get the new expression:

\[ C_{ab} = \sum_i (A_{a,i} \times B_{i,b} - A_{b,i} \times B_{i,a}) \]

Obviously, antisymmetrization introduces extra data dependence between input and result tensors. The calculation of each element in the result tensor involves multiple rows/columns from input tensors. The \textbf{antisymmetry} operator will be denoted as \( P(a_1...a_m|...b_1...b_n) \) and attached before the contraction operator. For instance, the above symmetrization contraction will be expressed as:

\[ C_{ab} = P(a|b) \sum_i A_{a,i} \times B_{i,b} \]  

Such expression will be identified as a \textbf{symmetrization contraction}, While, index within operator \( P(a_1...a_m|...b_1...b_n) \) will be referred as \textbf{symmetrization index}. Note here that, since \( C \) is antisymmetric in \( a \) and \( b \), only elements with \( a < b \) will be stored and calculated. It is half of the original size.

The formal mathematical definition and detailed description of antisymmetrization can be found in [42].

- **Decompression**: The opposite of symmetrization is symmetry breaking(decompression). An antisymmetry group in input tensors will not remain in the result tensor, if the contraction is performed over one or a few indices from the group.
Decompression will not result in more operations, but will cause additional dependence between input and result tensors. For example, consider the following contraction

\[ C_{a,b} = \sum_i A_{ai} \times B_{i,b} \]  

(7.2)

where \( A \) is an antisymmetric tensor, and only the portion with \( a < i \) is stored. The computation involving elements \( A_{a,i} (a > i) \) will employ its permutation component \( A_{i,a} \) as a substitute. Such contraction will be referred as decompression contraction, and the index in the breaking antisymmetry group will be referred as decompression index in the later discussion.

From analysis above, we can see that tensor contraction with permutation symmetry property may cause extra data dependence between input and result tensors than regular dense contraction. These type of contractions will be divided into three categories: 1)symmetrization contraction, 2)decompression contraction and 3)others(without extra dependence). Next, let’s analyze the effect of permutation symmetry on fusion and data locality optimization for each category.

- **Symmetrization Contraction:**

Considering the sample symmetrization contraction in Expression 7.1, there are two versions of implementation codes generated for the expression. The corresponding program fragments are shown in Figure 7.5(a), where we use \([b/a]\) to indicate the loop \( b \), whose range is bounded by another loop \( a \) as \( a \leq b \leq N_b \).

These two programs are functionally equivalent, but lead to different access patterns of input and result tensors. In the first program, each iteration contains two entries of input tensor \( A(B) \), referring to discontinuous locations. From another perspective,
for any arbitrary loop order, the same element of $A(B)$ will be used in discontinuous loops. On the other side, under certain loop orders, a fraction of the result tensor can be fully computed in continuous loops. For example, in the first code fragment of Figure 7.5(a), if loop $a$ is the outermost loop in the nest, within the iteration $a = x$, the sections of $A$, $B$ and $C$ involved in the computation are showed in Figure 7.6. After the iteration, the current row of the result tensor $C_x$ is fully computed and can be used in future calculations. The current section of $A$ and $B$ will be reused in later iterations. On the contrary, in the second program, the same fraction of $A(B)$ can be used consecutively under certain loop order. But, the computation of a result element is always performed in discontinuous loops. Thus, it is not possible to get and put each element of result tensor only once. For example, if loop $a$ is the outermost loop in the second code fragment of Figure 7.5(a), the element access pattern of $A$, $B$ and $C$ inside loop $a = x$ are showed in Figure 7.7. Note here that, after the iteration,
the current section of result tensor $C$ only obtains partial result and can’t be used in the next expression. Conversely, the current row of input tensor $A_{x,*}$ is only used in iteration $a = x$, and can be discarded afterward.

Based on the analysis above, we can establish the conditions and constraints to apply loop fusion for symmetrization contractions. If we want to fuse the result tensor by a symmetrization index, the first code generation method will be applied. Otherwise, if we want to fuse the input tensor, the second code generation method will be the proper choice. Corresponding pseudo code for these two fusion structures are presented in Figure 7.8. Apparently, we can’t fuse both input and result tensors by a symmetrization index. Other indices in a symmetrization contraction can be fused as normal.
Figure 7.7: Element access pattern in iteration $a = x$ of code2 in Figure 7.5(a)

```
for $a, [b/a]$
  for $i$
    $C = \frac{1}{2} \times (A_{a,i} \times B_{i,b} - A_{b,i} \times B_{i,a})$

(a) Fuse result $C$ with its consumer by $a$ and $[b/a]$

for $a, i$
  Producing $A$
    for $b$
      if $a < b$, $C_{ab^+} = \frac{1}{2} \times A \times B_{i,b}$
      if $a > b$, $C_{ba^-} = \frac{1}{2} \times A \times B_{i,b}$

(b) Fuse input $A$ with its producer by $a$ and $i$
```

Figure 7.8: Apply loop fusion for a symmetrization contraction

In addition to loop fusion, we also consider other aspects of data locality. Assuming all unfused tensors are resident on disk, we insert disk I/O statements to programs in Figure 7.8 and achieve new programs in Figure 7.9. In the first method, we need two extra buffers to store the additional section of $A$ and $B$. That will increase the memory requirement. In the second method, each element of $C$ will be brought
for $a, [b/a]$ for $a, i$

for $i$

- Read $A_{a,i}$ to $bufA_1$
- Read $A_{b,i}$ to $bufA_2$
- Read $B_{b,i}$ to $bufB_1$
- Read $B_{i,a}$ to $bufB_2$

$C^+ = \frac{1}{2} \times (bufA_1 \times bufB_1 - bufA_2 \times bufB_2)$

Consuming $C$

(b)Fuse $A$ with its producer and access $C/B$ on disk

if $a < b$
- Read $C_{ab}$ to $bufC$
- $bufC^+ = \frac{1}{2} \times A \times bufB$
- Write $bufC$ to $C_{ab}$

if $a > b$
- Read $C_{ba}$ to $bufC$
- $bufC^- = \frac{1}{2} \times A \times bufB$
- Write $bufC$ to $C_{ba}$

(a)Fuse $C$ with its consumer and access $A/B$ on disk

Figure 7.9: Fusion structure with disk I/O for a symmetrization contraction into memory and written back to disk $2 \times N_i$ times, That will increase the disk I/O cost.

- **Decompression Contraction:**

  For the sample decompression contraction in Expression 7.2, two programs to implement the expression are shown in Figure 7.5(b). Different element access patterns in these programs will impose different restrictions on further loop transformations.

  In the first method, the element $A_{x,y}$ will be used in two sets of discontinuous loop nests with $a = x, i = y$ and $a = y, i = x$. For any arbitrary loop order, it is not possible to get an element of $A$ and fully use it in successive loops. On the other hand, under certain loop orders, a fraction of the result tensor can be fully computed in continuous loops. For example, in the first code fragment of Figure 7.5(b), if loop $a$ is the outermost loop in the nest, the computation within iteration $a = x$ requires a partial row and a partial column of $A$, a row of $C$ and entire $B$, which are illustrated in
Figure 7.10. After the iteration, the current row of result tensor $C_{x,*}$ is fully computed and can be used in future expression. The partial column of $A$ will be reused in later iterations. On the contrary, in the second method, each iteration consists two accesses of the result tensor $C$, referring to discontinuous locations. From another perspective, the element $C_{x,y}$ will be computed in two sets of discontinuous loop nests $a = x, b = y$ and $a = y, b = x$. But, under certain loop orders, a fraction of the input tensor $A(B)$ can be fully used in continuous loops. For example, if loop $a$ is the outermost loop in the second code fragment of Figure 7.5(b), within the iteration $a = x$, the sections of $A$, $B$ and $C$ involved in the computation are showed in Figure 7.11. After the iteration, the current section of result tensor $C$ only obtains partial result. Conversely, the current row of input tensor $A_{x,*}$ is only used in iteration $a = x$, and can be discarded afterward.
Based on the analysis above, we can establish the conditions and constraints to apply loop fusion for decompression contractions. If we want to fuse the result tensor by a decompression index, the first code generation method will be used. Otherwise, if we want to fuse the input tensor, the second code generation method will be used. Corresponding pseudo code for these two fusion structures are presented in Figure 7.12. Apparently, we can’t fuse both input and result tensors by a decompression index. Other indices in a decompression contraction can be fused as normal.

In addition to loop fusion, we also consider other aspects of data locality. Assuming all unfused tensors are resident on disk, we insert disk I/O statements to programs in Figure 7.12 and achieve new programs in Figure 7.13. In the first method, each element of $A$ will be brought into memory $2 \times N_b$ times, That will **increase the disk**
for \( a, b \)
\[
\begin{array}{l}
\text{for } i \\
\quad \text{if } a < i, \quad C^+ = A_{ai} \times B_{i,b} \\
\quad \text{if } a > i, \quad C^- = A_{ia} \times B_{i,b} \\
\end{array}
\]
Consuming \( C \)

(a) Fuse result \( C \) with its consumer by \( a \) and \([b/a]\)

for \( a, [i/a] \)
\[
\begin{array}{l}
\text{Producing } A \\
\quad \text{for } b \\
\quad C_{a,b}^+ = A \times B_{i,b} \\
\quad C_{i,b}^- = A \times B_{a,b} \\
\end{array}
\]

(b) Fuse input \( A \) with its producer by \( a \) and \([i/a]\)

Figure 7.12: Apply loop fusion for a decompression contraction

for \( a, b \)
\[
\begin{array}{l}
\text{for } i \\
\quad \text{Read } B_{i,b} \text{ to } bufB \\
\quad \text{If } a < i \\
\quad \quad \text{Read } A_{ai} \text{ to } bufA \\
\quad \quad \quad C^+ = bufA \times bufB \\
\quad \text{If } a > i \\
\quad \quad \text{Read } A_{ia} \text{ to } bufA \\
\quad \quad \quad C^- = bufA \times bufB \\
\end{array}
\]
Consuming \( C \)

(a) Fuse \( C \) with its consumer and access \( A/B \) on disk

for \( a, [i/a] \)
\[
\begin{array}{l}
\text{Producing } A \\
\quad \text{for } b \\
\quad \text{Read } C_{a,b} \text{ to } bufC_1 \\
\quad \text{Read } C_{i,b} \text{ to } bufC_2 \\
\quad \text{Read } B_{i,b} \text{ to } bufB_1 \\
\quad \text{Read } B_{a,b} \text{ to } bufB_2 \\
\quad \quad \quad \text{buf}C_1^+ = A \times \text{buf}B_1 \\
\quad \quad \quad \text{buf}C_2^- = A \times \text{buf}B_2 \\
\quad \quad \text{Write } \text{buf}C_1 \text{ to } C_{a,b} \\
\quad \quad \text{Write } \text{buf}C_2 \text{ to } C_{i,b} \\
\end{array}
\]

(b) Fuse \( A \) with its producer and access \( C/B \) on disk

Figure 7.13: Fusion structure with disk I/O statements for a decompression contraction

**I/O cost.** In the second method, we need two extra buffers to store the additional section of \( C \) and \( B \). That will increase the memory requirement.

- **Others:**

Without antisymmetrization and decompression, the antisymmetry property in input tensors will retain in contraction without extra operator. The generated code of this kind of contractions will be in the same fashion of regular dense contraction except
for the reduced loop area. They will not impose extra constraints on loop fusion and other loop transformation strategies.

In summary, the presence of symmetry properties will change the detailed structure of tensors and have a significant impact on code generation and optimization algorithms. We will integrate general transformation strategies used in the current compiler optimizing system with the domain-specific properties of the computations we are targeting to obtain the most efficient possible code.
CHAPTER 8

CONCLUSION

8.1 Contributions

Sequences of tensor contractions arise in the domain of scientific applications. In particular, this is characteristic of many ab initio electronic structure models in chemistry, physics and material science, that account for significant fractions of supercomputer usage at national centers. This dissertation addresses the problem of automatic code generation for high-performance parallel programs involving a set of tensor contractions. We have concentrated mainly on the reduction of two kinds of I/O costs that are essential to the overall performance: inter-processor communication cost and disk access cost.

We start with the core operation, a single tensor contraction. First, an in-memory parallel algorithm is developed to execute a tensor contraction with efficient memory usage and minimum inter-processor communication. Since data sets involved in the considered computation are so large that they may not fit into memory even in parallel environment, out-of-core solutions are required in many cases. For a parallel out-of-core tensor contraction, many factors affect its execution, including the communication pattern, the in-memory parallel algorithm, data partition, loop permutation, disk I/O placements and tile size selection. These factors are inter-related and can not be optimized independently. We
model the relationship between different factors, and analyze their combined effects on the overall performance. Also, we provide performance models for different parallel out-of-core alternatives, and use predicted information from the performance models to drive the optimization techniques. Since a tensor contraction is essentially a generalized matrix multiplication, our techniques can also be used in conjunction with high-performance libraries for parallel out-of-core matrix computations for selecting the best version of code at runtime.

Based on the work above, we design and implement an optimization framework for a set of tensor contractions, which has been used in our domain-specific optimizing compiler, the Tensor Contraction Engine (TCE). A number of loop transformations are incorporated in this framework to adjust the program structure, thereby reducing the I/O cost under memory constraints. The main challenge lies in the complicated interactions between different transformations and the huge number of possible combinations. We provide a systematic approach to combine different loop transformations, along with pruning strategies to reduce the combined search space. In particular, we introduce a special loop structure, *maximally-fused fusion structure*, that offers a convenient basis for our optimization framework.

For further optimizations, we also utilize the domain-specific properties in the targeted computations, such as symmetry. We make a detailed investigation of the influence of symmetry properties on code generation and loop transformations, and propose how to adapt our general optimization system for tensors with these domain-specific properties, to obtain efficient code.
8.2 Future Work

Our current work on compilation and optimization focuses only on static compiler-time optimizations, and is limited to tensor contractions. As computational chemistry/physics methods are widely used in industrial and academic settings throughout the world, there arises the need for a program synthesis system to provide a high-level scripting interface for a range of scientific computations, while achieving high performance that is portable across machine architectures. To serve such a need, the work presented in this dissertation can be extended in the following areas:

- **Combining portability with high performance:** Different machine architectures and datasets may require different implementations of the same algorithm for performance consideration. The fast changing, increasingly complex and diverse computing platforms, encourage the development of portable programs.

To generate code that is portable for various situations without sacrificing performance, we need to combine static compile-time optimizations with dynamic run-time adaptations. Both compiler technologies and empirical performance models should be used to generate self-adapting high-performance code for a range of machine architectures and datasets. The generated codes can be pre-analyzed, with appropriate annotations to provide information on performance characteristics. Run-time optimizations can be applied using such information without requiring reimplementation for each different case.

- **Generalization of optimization framework:** Besides tensor contractions, there are many other computationally intensive components used in computational chemistry and physics applications, such as Cholesky factorization, Gaussian elimination, LU
decomposition etc. All these computations are also performed on very large multi-
dimensional arrays, and bear similar complicated reference patterns as tensor con-
traction. The out-of-core approaches used in our current optimizing compiler can be
further extended to serve the above computations. However, some open issues still
need to be solved. A significant challenge is the extension of the approach for com-
putations with more complex and arbitrary dependences: many loop transformations
may be illegal due to the data dependences.
BIBLIOGRAPHY


