2014

Search-based Model-driven Loop Optimizations for Tensor Contractions

Ajay Panyala
Louisiana State University and Agricultural and Mechanical College

SEARCH-BASED MODEL-DRIVEN LOOP OPTIMIZATIONS FOR TENSOR CONTRACTIONS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical Engineering and Computer Science

by
Ajay Panyala
B.Tech, JNT Univeristy, 2007
August 2014

Dedicated to my parents.

Acknowledgments

This dissertation would have never been possible without the strong support and guidance of my advisor Dr. Gerald Baumgartner and co-advisor Dr. J. Ramanujam. Gerald gave me the oppurtunity to pursue a doctoral degree regardless of the weak undergraduate background I had. Despite me being very slow with making progress in the first few years, he has always been very patient, even until the end of my doctoral study. I will be grateful to him forever. Dr. Ram has always provided useful advice and valuable insights into the research directions that needed to be pursued.

This research started with the idea of developing a domain-specific compiler for specific computations arising in quantum chemistry. Dr. Chi-Chung Lam was primarily responsible for the intial ideas and algorithms. A lot of other students had contributed to the design and initial implementation which was developed at Ohio State University. I would like to acknowledge all of their efforts which served as a foundation for my dissertation.

I would like to sincerely thank both Dr. Jianhua Chen for serving on my dissertation committee and Dr. James M. Matthews for serving as the dean's representative and for providing valuable feedback. I would also like to express my sincere thanks to Dr. David Tramell for provide systems support promptly whenever I needed anything and Ms. Maggie Edwards for all the administrative support.

Table of Contents

List of Tables

List of Figures

Abstract

Complex tensor contraction expressions arise in accurate electronic structure models in quantum chemistry, such as the coupled cluster method. The Tensor Contraction Engine (TCE) is a high-level program synthesis system that facilitates the generation of high-performance parallel programs from tensor contraction equations. We are developing a new software infrastructure for the TCE that is designed to allow experimentation with optimization algorithms for modern computing platforms, including for heterogeneous architectures employing general-purpose graphics processing units (GPGPUs). In this dissertation, we present improvements and extensions to the loop fusion optimization algorithm, which can be used with cost models, e.g., for minimizing memory usage or for minimizing data movement costs under a memory constraint. We show that our data structure and pruning improvements to the loop fusion algorithm result in significant performance improvements that enable complex cost models being use for large input equations. We also present an algorithm for optimizing the fused loop structure of handwritten code. It determines the regions in handwritten code that are safe to be optimized and then runs the loop fusion algorithm on the dependency graph of the code. Finally, we develop an optimization framework for generating GPGPU code consisting of loop fusion optimization with a novel cost model, tiling optimization, and layout optimization. Depending on the memory available on the GPGPU and the sizes of the tensors, our framework decides which processor (CPU or GPGPU) should perform an operation and where the result should be moved. We present extensive measurements for tuning the loop fusion algorithm, for validating our optimization framework, and for measuring the performance characteristics of GPGPUs. Our measurements demonstrate that our optimization framework outperforms existing general-purpose optimization approaches both on multi-core CPUs and on GPGPUs.

Chapter 1
Introduction

The Tensor Contraction Engine (TCE) [BAB+05b] targets a class of electronic structure calculations, which involve many computationally intensive components expressed as tensor contractions, essentially a collection of multi-dimensional summations of the product of several higher-dimensional arrays. Manual development of accurate quantum chemistry models in this domain is very tedious and takes an expert several months to years to develop and debug without the proper tools. The TCE aims to reduce the development time to hours/days, by having the chemist specify the computation in a high-level form, from which an efficient parallel program is automatically synthesized. The computational structures that are addressed by the TCE are present in some computational physics codes modeling electronic properties of semiconductors and metals, and in computational chemistry codes such as ACES II [SGW+], GAMESS [SBB+93], Gaussian [FF96], NWChem [VBG+10], PSI [CSV+07], and MOLPRO [WKM+10]. The synthesis of efficient parallel code from a high-level specification as a set of tensor contractions requires many optimization issues to be addressed. The approach has broader applicability and can be used in the automatic synthesis of out-of-core algorithms from abstract specifications in the form of loop computations with abstract arrays.

The objective is to minimize the execution time of such computations on a parallel computer while staying within the available memory. In addition to the performance optimization issues pertaining to inter-processor communication and data locality enhancement, there is an opportunity to apply algebraic transformations using the properties of commutativity, associativity and distributivity to reduce the total number of arithmetic operations.

A performance model-driven search-based approach to program transformation [BAB+05b, BBC+02, BKC+03, CBL+02b, KKB+03, KKB+04, KKB+06, GSL+05] has been pursued in the context of TCE. The TCE takes a high-level specification expressed as a set of tensor contraction expressions as input and optimizes it by using algebraic transformations and common subexpression elimination (CSE) to find an equivalent operation-minimal form of the computation and by searching for an evaluation order that uses the least amount of memory. If such an evaluation order is not found, a search for disk I/O placements and loop fusion configurations is performed that minimizes disk-to-memory traffic while ensuring that the

storage requirements for intermediates do not exceed the memory limit. Recent work [MKA11] extended this approach to orchestrate data across an additional level of memory hierarchy, namely GPGPU memory. In certain cases it may be cheaper to recompute parts of a computation instead of paying the penalty for disk I/O. If a computation is that large that after loop fusion some of the intermediates do not even fit on disk, as can be the case in quantum chemistry, there is no choice but to use recomputation to further reduce the storage requirements. For clusters, a communication optimizer has been developed that optimizes the inter-processor communication by considering communication cost together with finding a loop fusion configuration for minimizing storage without exceeding the available memory on each processor [Lam99, HSN+05, HLH+09]. The input arrays or intermediates that do not fit in memory are transposed if necessary using effective out-of-core transpose algorithms. Finally, the layout of intermediates in memory is optimized and the final code is generated. The TCE can generate either sequential out-of-core code or parallel code using the Global Arrays and Disk-Resident Arrays libraries that interfaces with the NWChem quantum chemistry suite.

We have developed an optimization framework that appropriately models the relation between loop fusion and memory usage. We present algorithms that find an optimal loop fusion configuration that minimizes memory usage, under both static and dynamic memory allocation models.

Reduction of arithmetic operations has been traditionally done by compilers using the technique of common subexpression elimination [FL91]. Much work has been done on improving locality and parallelism by loop fusion [KM94, MA97, SM97]. However, the TCE considers a different use of loop fusion [LCBS99], which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. Traditional compiler research does not address this use of loop fusion because this problem does not arise with manually-produced programs. We are unaware of any work on fusion of multi-dimensional loop nests into imperfectly-nested loops as a means to reduce memory usage.

A prototype of the TCE [Hir03] has been successful in introducing many new models into the NWChem suite by developing several useful domain-specific transformations. An optimizing TCE [BAB+05a] has been developed primarily at the Ohio State University that has explored many novel optimization algorithms. However, many key aspects in optimizing tensor contraction codes remain unexplored as both the Prototype TCE and the Ohio TCE versions were primarily developed before the era of multi-cores, GPGPUs, and heterogeneous computing; they do not provide sufficient support for tree transformations and for generating multi-core parallelism or GPU code. Their monolithic structure is too inflexible to adapt for multi-target

code generation. They also do not take advantage of major recent advances in polyhedral compilation models that provide exact dependency analysis and a more efficient way to perform program transformations, as can be found in the tools Pluto [BHRS08a, BBK+08, BHRS08b, BRS10] and PrimeTile [HBB+09b, HBB+09a, BHH+10]. The TCE does not have Fortran or C++ frontend support for handwritten code and does not support dependency analysis. Also, since not all quantum chemistry codes can be expressed in the form of tensor contraction expressions, it would be desirable if some of the optimization algorithms, in particular the loop fusion and tiling optimizations, could be applied to handwritten code and if the TCE optimization algorithms could support symmetric and block-sparse tensors. Additional research on data movement optimization and parallelization and on developing the cost models for the compiler to perform these optimizations for heterogeneous systems is needed. The final product should also interface with quantum chemistry suites and other simulation tools.

This dissertation aims to improve upon the existing TCE compilation framework by addressing some of the unexplored issues through effective performance models and search-based optimization algorithms and by generating efficient, platform-adaptable implementations of tensor contractions. The contributions of this thesis are the following:

- We have developed a new software infrastructure for the TCE on top of the ROSE compiler infrastructure [RSE]. We have added a frontend to ROSE for our improved TCE source language and provide support for implementing TCE optimization algorithms both in the frontend and on ROSE abstract syntax trees.

- We have made significant performance improvements to the loop fusion optimization algorithm that allow large quantum chemistry equations to be optimized with complex (2-dimensional) loop fusion cost models, and we have developed a loop fusion cost model for memory minimization with dynamic memory allocation.

- We have developed a loop fusion optimization algorithm that can be applied to simple handwritten code as an alternative to the loop fusion algorithm for expression trees representing tensor contraction equations.

- We have developed an optimization framework for generating GPGPU code for tensor contractions. The optimization framework consists of a novel cost model for the loop fusion algorithm and refinements of the tiling and layout optimization algorithms.

- We have performed extensive measurements to document the performance characteristics of the loop fusion algorithm as well as to understand the performance characteristics of tensor contraction computations on multi-core CPUs and on GPGPUs.

- We have demonstrated that the performance of code resulting from our optimization framework for dense tensor contractions is superior to that of code generated by polyhedral compiler frameworks, both on multi-core CPUs and on GPGPUs.

Chapter 2
Background

2.1 Operation Minimization

In the class of computations considered, the final result to be computed can be expressed as multi-dimensional integrals of the product of many input arrays. Due to commutativity, associativity and distributivity, there are many different ways to obtain the same final result and they could differ widely in the number of floating point operations required. The problem of finding an equivalent form that computes the result with the least number of operations is not trivial and so a software tool for doing this is desirable.

Consider, for example, the multi-dimensional integral shown in Figure 2.1(a), where the input arrays are assumed to be produced by generator functions. If implemented directly as expressed (i.e., as a single set of perfectly-nested loops), the computation would require $2 \times N_i \times N_j \times N_k \times N_l$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of the distributive law of multiplication over addition is satisfactory for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $2 \times N_j \times N_k \times N_l + 2 \times N_j \times N_k + N_i \times N_j$ operations is given in Figure 2.1(b). It expresses the sequence of steps in computing the multi-dimensional integral as a sequence of formulas. Each formula computes some intermediate result and the last formula gives the final result. A formula is either a product of two input/intermediate arrays or a integral/summation over one index, of an input/intermediate array. A sequence of formulas can also be represented as an expression tree. For instance, Figure 2.1(c) shows the expression tree corresponding to the example formula sequence. The problem of finding a formula sequence that minimizes the number of operations has been proved to be NP-hard [LSW97].

2.2 Memory Minimization Problem

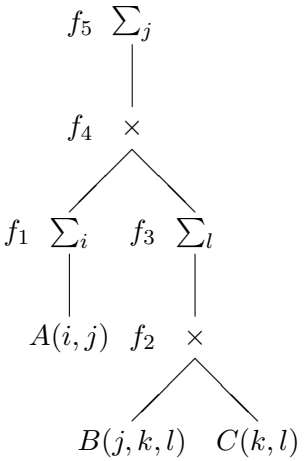
In implementing the computation represented by an operation-count-optimal formula sequence (or an expression tree), it may be necessary to perform loop fusion to reduce the sizes of the intermediate arrays. Without fusing the loops, intermediate arrays could be too large to fit into available memory. There are many different ways to fuse the loops and they could result in different memory usage.

$$W[k] \;=\; \sum_{(i,j,l)} A(i,j) \times B(j,k,l) \times C(k,l)$$

(a) A multi-dimensional integral

$$\begin{array}{lcl} f_A[i,j] & = & A(i,j) \\ f_1[j] & = & \sum_i f_A[i,j] \\ f_B[j,k,l] & = & B(j,k,l) \\ f_C[k,l] & = & C(k,l) \\ f_2[j,k,l] & = & f_B[j,k,l] \times f_C[k,l] \\ f_3[j,k] & = & \sum_l f_2[j,k,l] \\ f_4[j,k] & = & f_1[j] \times f_3[j,k] \\ W[k] = f_5[k] & = & \sum_j f_4[j,k] \end{array}$$

(b) A formula sequence for computing (a)



(c) An expression tree representation of (b)

FIGURE 2.1. An example multi-dimensional integral and two representations of a computation.

Consider again the computation $W[k] = \sum_j(\sum_i(A(i,j)) \times \sum_l(B(j,k,l) \times C(k,l)))$, with the expression tree shown in Figure 2.1(c). A naive approach to implement this expression is to use a set of perfectly nested loops for each node in the tree, as shown in Figure 2.2(a). The brackets indicate the scopes of the loops. Code for array allocation and initialization is omitted for clarity. Figure 2.2(b) shows how the sizes of the arrays may be reduced by using loop fusion. It shows the resulting loop structure after fusing all the loops between A and f_1, all the loops among B, C, f_2, and f_3, and all the loops between f_4 and f_5. After fusing all the loops between a node and its parent, each array element gets used immediately after it is produced. The dimensions of the child array are, therefore, no longer needed and can be eliminated. Here, f_A, f_B, f_C, f_2, and f_4 are reduced to scalars. Note that some loop nests (such as those for B and C) were reordered and some loops within loop nests (such as the j-, k-, and l-loops for B, f_2, and f_3) were permuted in order to facilitate loop fusions.

```
for i
⌈ for j
⌊ ⌈ fA[i,j]=A(i,j)
for j
⌈ for k
⌊ ⌈ for l
⌊ ⌈ fB[j,k,l]=B(j,k,l)
for k
⌈ for l
⌊ ⌈ fC[k,l]=C(k,l)
initialize f1
for i
⌈ for j
⌊ ⌈ f1[j]+=fA[i,j]
for j
⌈ for k
⌊ ⌈ for l
⌊ ⌈ f2[j,k,l]=fB[j,k,l]×fC[k,l]
initialize f3
for j
⌈ for k
⌊ ⌈ for l
⌊ ⌈ f3[j,k]+=f2[j,k,l]
for j
⌈ for k
⌊ ⌈ f4[j,k]=f1[j]×f3[j,k]
initialize f5
for j
⌈ for k
⌊ ⌈ f5[k]+=f4[j,k]
```

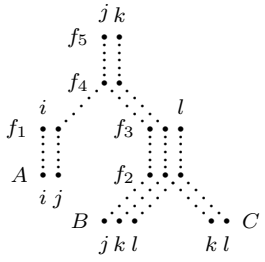
(a)

```
initialize f1
for i
⌈ for j
⌊ ⌈ fA=A(i,j)
⌊ ⌈ f1[j]+=fA
initialize f3
for k
⌈ for l
⌊ ⌈ fC=C(k,l)
⌊ ⌈ for j
⌊ ⌈ fB=B(j,k,l)
⌊ ⌈ f2=fB×fC
⌊ ⌈ f3[j,k]+=f2
initialize f5
for j
⌈ for k
⌊ ⌈ f4=f1[j]×f3[j,k]
⌊ ⌈ f5[k]+=f4
```

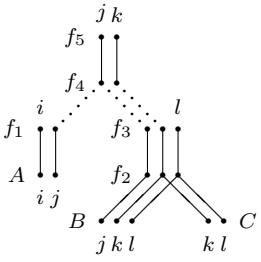
(b)

```
for k
⌈ for l
⌊ ⌈ fC[k,l]=C(k,l)
initialize f5
for j
⌈ for i
⌊ ⌈ fA[i]=A(i,j)
⌊ initialize f1
⌊ for i
⌊ ⌈ f1+=fA[i]
⌊ for k
⌊ ⌈ initialize f3
⌊ ⌈ for l
⌊ ⌈ fB=B(j,k,l)
⌊ ⌈ f2=fB×fC[k,l]
⌊ ⌈ f3+=f2
⌊ ⌈ f4=f1×f3
⌊ ⌈ f5[k]+=f4
```

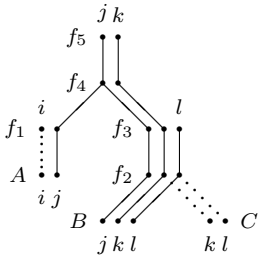
(c)



(d)



(e)



(f)

FIGURE 2.2. Three loop fusion configurations for the expression tree in Figure 2.1.

For this example, we assume the leaf node arrays (i.e., input arrays) can be generated one element at a time by a generator function so that loop fusions with their parents are allowed. This assumption holds for arrays in which the value of each element is a function of the array subscripts, as in many arrays in the physics and chemistry computations that we work on. Our algorithm also handles the case where arrays are already memory-resident, and, as will be shown in Section 2.4.6, the case where an input array has to be read in or produced in a certain order, in slices, or in its entirety.

Figure 2.2(c) shows another possible loop fusion configuration obtained by fusing all the j-loops and then all the k-loops and l-loops inside. The sizes of all arrays except f_C and f_5 are smaller. By fusing the j-, k-, and l-loops between those nodes, the j-, k-, and l-dimensions of the corresponding arrays can be eliminated. Hence, f_B, f_1, f_2, f_3, and f_4 are reduced to scalars while f_A becomes a one-dimensional array. Since j is the outermost loop, the k- and l-loops cannot be fused between C and f_2.

In general, fusing a t-loop between a node v and its parent eliminates the t-dimension of the array v and reduces the array size by a factor of N_t. In other words, the size of an array after loop fusions equals the product of the ranges of the loops that are not fused with its parent. We only consider fusions of loops among nodes that are all transitively related by (i.e., form a transitive closure over) parent-child relations. Fusing loops between unrelated nodes (such as fusing siblings without fusing their parent) has no effect on array sizes. We also restrict our attention to loop fusion configurations that do not increase the operation count. The tradeoff between memory usage and arithmetic operations is considered in Section 2.4.7.

For the class of loops considered here, the only dependence relations are those between children and parents, and array subscripts are simply loop index variables. Loop permutations, loop nests reordering, and loop fusions are, therefore, always legal as long as child nodes are evaluated before their parents. This freedom allows the loops to be permuted, reordered, and fused in a large number of ways that differ in memory usage. Finding a loop fusion configuration that uses the least memory is not trivial.

In the examples above, we have assumed that leaf node arrays are generated and that each array element should be generated only once. This assumption is appropriate if generating array elements is expensive. However, if leaf arrays are already in memory, this assumption is too restrictive. By allowing leaf arrays to be accessed repeatedly at zero cost, we might enable additional loop fusions. For example, if the array C in Figure 2.2(c) were already in memory, we could allow rereading it for every iteration of the j-loop, which would allow full fusion of C with f_2. Since the appropriate treatment of leaf arrays depends on the

application, we allow both. We will use the function call notation $A(i, j)$ for arrays that are generated by a generator function call and the notation $A[i, j]$ for arrays that are already memory resident.

Definition 2.2.1. More formally, we define an *expression tree* to be a tree structure consisting of the following types of nodes:

- Array(a,i): An array reference $a[i]$ of a memory-resident array a with index vector i.
- Fun(f, i): A function call $f(i)$ of function f with index vector i as arguments.
- Const(x): The integer or floating point constant x.
- BinOp(o, l, r): A binary operator expression $l\, o\, r$ with operator o, left subtree l, and right subtree r, where $o \in \{+, -, *\}$.
- Sum(i, t): A summation operator $\sum_i t$ with summation index set i and subtree t.

We define the *memory minimization problem* as follows:

> Given an expression tree T and the ranges of all indices.
>
> Find a loop structure for computing T that uses the least amount of temporary storage without increasing the operation count. We assume that leaf array references and constants are memory-resident and can be accessed repeatedly at zero cost.

2.3 Fusion Graphs

For facilitating the enumeration of all possible loop fusion configurations for a given expression tree, we define a representation of the expression tree's loop structure we call a *fusion graph*. The fusion graph makes the indices of nodes in the expression tree explicit and indicates the scopes of fused loops.

Definition 2.3.1. Let T be an expression tree. For any given node $v \in T$, let *subtree*(v) be the set of nodes in the subtree rooted at v, *parent*(v) be the parent node of v, and *indices*(v) be the set of loop indices for v (including the summation indices *sumindices*(v) if v is a summation node). A *fusion graph* representation of a fused loop structure for T is constructed as follows:

1. Corresponding to each node v in T a fusion graph contains a set of vertices, one for each index $i \in$ *indices*(v).

2. For each Array or Const node v in T and for each index $i \in \textit{indices}(\textit{parent}(v)) - \textit{indices}(v)$ that is fused between v and its parent (i.e., for each loop in which v is accessed redundantly), an i-index is added to the set of nodes corresponding to v.

3. For each loop of index i that is fused between a node and its parent, the i-vertices for the two nodes are connected with a *fusion edge*.

4. For each index i that is shared between a node and its parent, for which the corresponding loops are not fused, the i-vertices for the two nodes are connected with a *potential fusion edge*.

Figure 2.2 shows the fusion graphs alongside the loop fusion configurations. Fusion edges are drawn as solid lines. Potential fusion edges, i.e., fusion opportunities that were not exploited, are drawn as dotted lines. As an example, consider the loop fusion configuration in Figure 2.2(b) and the corresponding fusion graph in Figure 2.2(e). Since the j-, k-, and l-loops are fused between f_2 and f_3, there are three fusion edges between f_2 and f_3 in the fusion graph, one for each of the three loops. Since no loops are fused between f_3 and f_4, the edges between f_3 and f_4 in the fusion graph remain potential fusion edges.

Definition 2.3.2. In a fusion graph, we call each connected component of fusion edges (i.e., a maximal set of connected fusion edges) a *fusion chain*, which corresponds to a fused loop in the loop structure. The *scope of a fusion chain* c, denoted $\textit{scope}(c)$, is defined as the set of nodes it spans.

In Figure 2.2(f), there are three fusion chains, one for each of the j-, k-, and l-loops; the scope of the shortest fusion chain is $\{B, f_2, f_3\}$. The scope of any two fusion chains in a fusion graph must either be disjoint or a subset/superset of each other. Scopes of fusion chains do not partially overlap because loops do not (i.e., loops must be either separate or nested). Therefore, any fusion graph with fusion chains whose scopes are partially overlapping is illegal and does not correspond to any loop fusion configuration.

Fusion graphs help us visualize the structure of the fused loops and find further fusion opportunities. If we can find a set of potential fusion edges that, when converted to fusion edges, does not lead to partially overlapping scopes of fusion chains, then we can perform the corresponding loop fusions and reduce the sizes of some arrays. For example, the i-loops between A and f_1 in Figure 2.2(f) can be further fused and array f_A would be reduced to a scalar. If converting all potential fusion edges in a fusion graph to fusion edges does not make the fusion graph illegal, then we can completely fuse all the loops and achieve optimal memory usage. But for many fusion graphs in real-life loop configurations (including the ones in Figure 2.2),

this does not hold. Instead, potential fusion edges may be mutually prohibitive; fusing one loop could prevent the fusion of another. In Figure 2.2(e), fusing the j-loops between f_1 and f_4 would disallow the fusion of the k-loops between f_3 and f_4.

Although a fusion graph specifies what loops are fused, it does not fully determine the permutations of the loops and the ordering of the loop nests. As we will see in Section 4.4, under dynamic memory allocation, reordering loop nests could alter memory usage without changing array sizes.

So far, we have been describing the fusion between a node and its parent by the set of fused loops (or the loop indices such as $\{i, j\}$). But in order to compare loop fusion configurations for a subtree, it is desirable to include information about the relative scopes of the fused loops in the subtree.

Definition 2.3.3. The *loop scope* of index i in a subtree rooted at v, denoted *scope*(i, v), is defined in the usual sense as the set of nodes in the subtree that the fused loop spans. That is, if the i-loop is fused, *scope*(i, v) = *scope*$(c) \cap$ *subtree*(v), where c is a fusion chain for the i-loop with $v \in$ *scope*(c). If the i-loop of v is not fused, then *scope*$(i, v) = \{v\}$. As an example, for the fusion graph in Figure 2.2(e), *scope*$(j, f_3) = \{B, f_2, f_3\}$.

For describing the relative scopes of a set of fused loops, we introduce the notion of an *indexset sequence*, which is defined as an ordered list of disjoint, non-empty sets of loop indices. For example, $f = \langle\{i, k\}, \{j\}\rangle$ is an indexset sequence. For simplicity, we write each indexset in an indexset sequence as a string. Thus, f is written as $\langle ik, j\rangle$.

Definition 2.3.4. Let g and g' be indexset sequences. We denote by $|g|$ the number of indexsets in g, $g[r]$ the r-th indexset in g, and *set*(g) the union of all indexsets in g, i.e., *set*$(g) = \bigcup_{1 \le r \le |g|} g[r]$. An index i has *rank* r in indexset sequence g, i.e., *rank*$(i, g) = r$, if and only if $i \in g[r]$.

For example, if $f = \langle ik, j\rangle$, then $|f| = 2$, $f[1] = \{i, k\}$, *set*(f) = *set*$(\langle j, i, k\rangle) = \{i, j, k\}$, and *rank*$(i, f) = 1$.

An indexset sequence f represents constraints on the loop structure. For two indices i and j, if *rank*$(i, f) <$ *rank*(j, f), then the j-loop is constrained to be within the i-loop. If two indices have the same rank in f, there is no constraint on the corresponding loops and the loops, therefore, can be permuted. For example, the indexset sequence $\langle ik, j\rangle$ represents the constraint that the j-loop is within the i and k loops and that the latter two can be permuted.

We define a *nesting* of the loops at a node v as an indexset sequence. Intuitively, the loops at a node are ranked by their scopes in the subtree. Two loops have the same rank (i.e., are in the same indexset) if they have the same scope. For example, in Figure 2.2(e), the nesting at f_3 is $\langle kl, j\rangle$, the nesting at f_4 is $\langle jk\rangle$, and the nesting at B is $\langle jkl\rangle$.

Definition 2.3.5. Formally, a nesting of the loops at a node v is an indexset sequence h such that

1. *set*$(h) =$ *indices*(v) and
2. for all $i, i' \in$ *set*(h) where $r =$ *rank*(i, h) and $r' =$ *rank*(i', h),
   (a) $r = r'$ if and only if *scope*$(i, v) =$ *scope*(i', v), and
   (b) $r < r'$ if and only if *scope*$(i, v) \supset$ *scope*(i', v).

By definition, the loop nesting at a leaf node v must be $\langle$*indices*$(v)\rangle$ because all loops at v have the same scope of $\{v\}$.

Similarly, we use the notion of an indexset sequence to define a *fusion*. Intuitively, the loops fused between a node and its parent are ranked by their scopes in the subtree from largest to smallest. For example, in Figure 2.2(f), the fusion between f_2 and f_3 is $\langle jkl\rangle$ and the fusion between f_4 and f_5 is $\langle j, k\rangle$ (because the j-loop covers two more nodes, A and f_1).

Definition 2.3.6. Formally, a fusion between a node v and *parent*(v) is an indexset sequence f such that

1. *set*$(f) \subseteq$ *indices*$(v) \cap$ *indices*(*parent*(v)),
2. for all $i \in$ *set*(f), the i-loop is fused between v and *parent*(v), and
3. for all $i, i' \in$ *set*(f) where $r =$ *rank*(i, f) and $r' =$ *rank*(i', f),
   (a) $r = r'$ if and only if *scope*$(i, v) =$ *scope*(i', v), and
   (b) $r < r'$ if and only if *scope*$(i, v) \supset$ *scope*(i', v).

A legal fusion graph (corresponding to a loop fusion configuration) for an expression tree T can be built up in a bottom-up manner by merging legal fusion graphs for the subtrees of T. For characterizing legal fusions and for motivating the algorithm for constructing a fusion graph, we need to define a *prefix* relation between indexset sequences.

Definition 2.3.7. Let g and g' be indexset sequences. We say that g' is a *prefix* of g if $|g'| \leq |g|$, $g'[r] = g[r]$ for all $1 \leq r < |g'|$, and $g'[|g'|] \subseteq g[|g'|]$. We write this relation as *prefix*(g', g). E.g., $\langle\rangle$, $\langle i\rangle$, $\langle k\rangle$, $\langle ik\rangle$, and $\langle ik, j\rangle$ are prefixes of $\langle ik, j\rangle$, but $\langle i, j\rangle$ is not.

For a given node v, the nesting h at v summarizes the fusion graph for the subtree rooted at v and determines what fusions are allowed between v and its parent. A fusion f is legal for a nesting h at v, denoted *legal*(f, h, v), if *prefix*(f, h) and *set*$(f) \subseteq$ *indices*(*parent*(v)). This is because loops with larger scopes must be fused before fusing those with smaller scopes, and because only loops common to both v and its parent may be fused.

As an example, consider the fusion graph for the subtree rooted at f_2 in Figure 2.2(e). Since the nesting at f_2 is $\langle kl, j\rangle$ and *indices*$(f_3) = \{j, k, l\}$, the legal fusions between f_2 and f_3 are $\langle\rangle$, $\langle k\rangle$, $\langle l\rangle$, $\langle kl\rangle$, and $\langle kl, j\rangle$.

Definition 2.3.8. All legal fusions for a node v with nesting h are prefixes of a *maximal legal fusion*, denoted *maxFusion*(h, v). We define $f' =$ *maxFusion*(h, v) if and only if *legal*(f', h, v), and for all f, *legal*(f, h, v) implies *prefix*(f, f'). In Figure 2.2(e), the maximal legal fusion for C is $\langle kl\rangle$, and for f_2 is $\langle kl, j\rangle$.

2.4 Loop Fusion Algorithm

Given a specification of the required computation as a multi-dimensional sum of the product of input arrays, we first determine an equivalent sequence of multiplication and summation formulas that compute the result using a minimum number of arithmetic operations. Each formula computes and stores some intermediate results in an intermediate array. By computing the intermediate results once and reusing them multiple times, the number of arithmetic operations can be reduced.

The simplest way to implement an optimal sequence of multiplication and summation formulas is to compute the formulas one by one, each coded as a set of perfectly nested loops, and to store the intermediate results produced by each formula in an intermediate array. However, in practice, the input and intermediate arrays could be so large that they cannot fit into the available memory. Hence, there is a need to fuse the loops as a means of reducing memory usage. By fusing loops between the producer loop and the consumer loop of an intermediate array, intermediate results are formed and used in a pipelined fashion and they reuse the same reduced array space. The problem of finding a loop fusion configuration that minimizes memory usage without increasing the operation count is not trivial.

2.4.1 Algorithm for Static Memory Allocation

We use a dynamic programming algorithm for computing the fusion graph that uses the least amount of memory. The algorithm traverses the expression tree bottom-up and generates a set of possible loop structures for each tree node. Each solution is represented as a pair containing a fusion and the memory cost.

Let v be the root node of an expression tree. The algorithm has the following recursive structure:

1. If v is a Fun node, a single solution is generated that indicates that v can be fully fused with its parent.

2. If v is an Array or Const node, a single solution is generated that indicates that v can be accessed repeatedly at no cost and that v can be fused with any loop, not only those in *indices*(v).

3. If v is a Sum node, multiple solutions for v are created out of each solution for the subtree. For each solution for the subtree, we generate all the possible fusions with v, i.e., we enumerate all the possibilities for how the loop structure corresponding to the subtree solution can be embedded within a loop structure of the parent. The summation indices as well as any indices whose loops are nested within the summation loops are then removed from the solutions since they cannot be further fused with v's parent.

4. If v is a BinOp node, solutions for the left and right subtrees must be combined to form a solution for v. For each solution for a subtree, we again generate all the possible fusions with v. Then, for all compatible pairs of possible fusions for the two subtrees, a solution for v is generated by merging the fusion constraints from the subtree solutions.

5. In each step, inferior solutions are pruned out.

Before presenting the details of the algorithm, we need to define the mechanisms by which solutions are constructed out of solutions for subtrees.

Definition 2.4.1.1. The *concatenation* of an indexset sequence g and an indexset x, denoted $g + x$, is defined as the indexset sequence g' such that if $x \neq \emptyset$, then $|g'| = |g| + 1$, $g'[|g'|] = x$, and for all $1 \leq r < |g'|$, $g'[r] = g[r]$; otherwise, $g' = g$.

When constructing a loop nesting for a node u out of solutions for the subtrees, we must ensure that the nesting is compatible with the fusions between u and the subtrees. We compute the nesting of u by *extending* the fusions to the indices of u.

Definition 2.4.1.2. Let u be the parent of a node v and let f be a fusion between u and v. The *extended nesting* of fusion f for node u, denoted *extendNesting*(f, u), is defined as *extendNesting*$(f, u) = f + ($*indices*$(u) -$ *set*$(f))$.

If v is the only child of u, then the loop nesting at u as a result of fusion f between u and v is *extendNesting*(f, u). I.e., all the loops that are not fused between u and v must be nested within the fused loops. For example, in Figure 2.2(e), if the fusion between f_2 and f_3 were $\langle kl \rangle$, then the nesting at f_3 would be $\langle kl, j \rangle$.

If u is a binary node, then the extended nestings of the subtrees must be compatible. The constraints on the loop structure expressed by the extended nestings of the subtrees must be merged to produce the nesting of u.

Definition 2.4.1.3. Suppose u has the two children v and v', whose fusions with u are f and f', respectively. For the fusion graph for the subtree rooted at u (which will be merged from those of v and v') to be legal, $h =$ *extendNesting*(f, u) and $h' =$ *extendNesting*(f', u) must be *compatible* according to the condition: for all $i \in h[r]$ and $j \in h[s]$, if $r < s$ and $i \in h'[r']$ and $j \in h'[s']$, then $r' \leq s'$.

This requirement ensures an i-loop that has a larger scope than a j-loop in one subtree will not have a smaller scope than the j-loop in the other subtree. If the extended nestings h and h' of the subtrees are compatible, they can be merged to form a nesting of u.

Definition 2.4.1.4. Let h and h' be compatible extended nestings for the subtrees of u. The nesting $h'' =$ *mergeNesting*(h, h') of u must satisfy the following conditions. For all $i \in h''[r'']$ and $j \in h''[s'']$, if $i \in h[r]$, $i \in h'[r']$, $j \in h[s]$, and $j \in h'[s']$, then $[r'' = s'' \Rightarrow r = s$ and $r' = s']$ and $[r'' \leq s'' \Rightarrow r \leq s$ and $r' \leq s']$.

This definition ensures that h'' is compatible with both h and h'. Effectively, the loops at u are re-ranked by their combined scopes in the two subtrees to form h''. As an example, in Figure 2.2(e), if the fusion between f_1 and f_4 is $f = \langle j \rangle$ and the fusion between f_3 and f_4 is $f' = \langle k \rangle$, then $h =$ *extendNesting*$(f, f_4) = \langle j, k \rangle$ and $h' =$ *extendNesting*$(f', f_4) = \langle k, j \rangle$ would be incompatible. But if f were changed to $\langle\rangle$, then $h =$ *extendNesting*$(f, f_4) = \langle jk \rangle$ would be compatible with h', and the resulting nesting at f_4 would be $\langle k, j \rangle$.

As outlined above, the algorithm generates a set of solutions for each node. In order to effectively prune inferior solutions from such a set, we need a comparison relation on nestings and fusions. Informally, a nesting h is *more or equally constraining than* another nesting h' for the same node v if any loop fusion configuration for the rest of the tree that is compatible with h is also compatible with h'.

Definition 2.4.1.5. We define a nesting h for a node v to be more or equally constraining than a nesting h', denoted $h \sqsubseteq h'$, if and only if $h = \textit{mergeNesting}(h, h')$. Note that $\sqsubseteq$ is a partial order: if h and h' are not compatible, then $h \sqsubseteq h'$ is undefined.

If $h \sqsubseteq h'$, then for all $i \in h[r]$ and $j \in h[s]$, there exist r', s' such that $i \in h'[r']$ and $j \in h'[s']$ and $[r < s \Rightarrow r' \leq s']$ and $[r = s \Rightarrow r' = s']$. I.e., for any pair of loops i and j, if h' constrains one of them to be nested within the other, so does h. If h does not constrain them, neither does h'. Any nesting for the parent of v that can be constructed out of the nesting h for v can, therefore, also be constructed out of h'.

Comparing the nestings at f_3 between Figure 2.2(e) and (f), the nesting $\langle kl, j \rangle$ in (e) is more constraining than the nesting $\langle jkl \rangle$ in (f).

For defining the relation $\sqsubseteq$ for fusions, we need the additional requirement that the sets of indices in the two fusions are the same:

Definition 2.4.1.6. We define a fusion f to be more or equally constraining than a fusion f', denoted $f \sqsubseteq f'$, if and only if $\textit{set}(f) = \textit{set}(f')$ and $f = \textit{mergeNesting}(f, f')$.

2.4.2 Algorithm Details

The algorithm traverses the expression tree bottom up and creates a set of solutions for each node in the expression tree. Each solution $\mathrm{S}(f, c, t)$ for an expression tree t contains the maximal fusion f with the parent and the memory cost c. Since under a static memory allocation model all the intermediate arrays for evaluating an expression exist during the entire computation, the memory cost is simply the sum of the sizes of all the arrays and scalars. For presentation purposes, fusions, or indexset sequences in general, are represented as lists of lists of strings.

For each solution, we must record the subtree solution(s) from which this solution was constructed. To do so, we use the following solution tree data structure:

- Leaf(s): A leaf in the solution tree with solution s.
- Unary(s, t): A unary node in the solution tree with solution s and subtree t.
- Binary(s, l, r): A binary node in the solution tree with solution s, left subtree l, and right subtree r.
- Extended(s, h, t): An extended solution node with solution s, extended nesting h, and subtree t.

- Reduced(s, i, t): A reduced tree node with solution s, index set i, and subtree t.

The loop fusion algorithm will construct a set of such solution trees for each node in the expression tree. The structure of a solution tree mirrors the structure of the expression tree. For an Array, Fun or Const node, a single Leaf node is constructed. For Sum and BinOp nodes, the algorithm constructs Unary and Binary nodes, respectively. In addition, an Extended node is constructed when extending a solution for a subtree to the indices of the parent. Similarly, a Reduced node is constructed for removing the summation indices out of the the solutions for a Sum node. Extended and Reduced nodes correspond to edges in the expression tree. Making extended nodes explicit improves the efficiency of the algorithm, since extended nestings do not have to be recomputed, while keeping the code structure simple. Reduced nodes mostly serve to make the step of removing summation indices explicit and aid in debugging.

Before describing the algorithm in detail, we need to introduce some helper functions and operations on indexset sequences. Figure 2.3 defines auxiliary functions for expression trees and solution trees. The functions `indices` and `fusible` return the set of indices and the set of fusible indices of a node, respectively. The call `fusedSize(`v`,` f`)` returns the memory needed for storing the result of node v under the fusion f with its parent. The storage needed for an Array node is zero for the purpose of the fusion algorithm, since Array nodes are assumed to be memory resident and do not change in size based on the fusion. The functions `getFusion` and `getSoln` are simply accessor functions for the data structure; `getNesting` constructs the extended nesting from an Extended node to allow pruning extended solutions and for passing it to the cost model so it does not need to be recomputed.

```
indices(v): return { i | i is an index at node v }

fusible(v): return { i | i in indices(v) and i not a summation index of v }

fusedSize(v, f):
  if v is an Array or a Const node then return 0
  else return Product of range(i) for i in (fusible(v) - set(f))

getFusion(S(f, c, t)): return f

getSoln(s): return the solution out of solution tree node s

getNesting(s):
  if s is of the form Extended(S(f, c, t), h, s') then return h
  else return getFusion(getSoln(s))
```

FIGURE 2.3. Auxiliary functions for accessing the data structures.

Figure 2.4 shows functions that operate on indexset sequences. Indexset sequences are represented as lists of lists of strings. The constant `any`, represented as the special indexset sequence $\langle * \rangle$, is used as the

maximal fusion for Array and Const nodes. It indicates that any index of the parent can be fused between the leaf node and the parent, not only indices of the leaf node. The function `set` returns the set of indices in a fusion. The functions `prefixes` and `extendNesting` are used for constructing extended solutions. `prefixes` takes a maximal fusion f as argument and returns the set of all prefixes of f or actual fusions. `extendNesting` takes an actual fusion as argument and constructs an extended nesting.

```
any = [["*"]]

set(f): return { i | i occurs in indexset sequence f }

prefixes(f): return set of all prefixes of indexset sequence f

extendNesting(f, i):
  if f = any then return [i]
  else return f + (i - set(f))

maxPrefix(f, i):
  return maximum prefix of indexset sequence f containing only indices in i

merge(h, h'):
  if h = nil or h' = nil then return nil
  else
    y = intersect(head(h), head(h'))
    x = head(h) - y;  x' = head(h') - y
    if y = nil then return nil
    else if x = nil and x' = nil then return y :: merge(tail(h), tail(h'))
    else if x = nil then return y :: merge(tail(h), x'::tail(h'))
    else if x' = nil then return y :: merge(x::tail(h), tail(h'))
    else return y :: merge(x::tail(h), x'::tail(h'))

mergeNesting(h, h'):
  if h = any or h' = any or set(h) <> set(h') then raise Incompatible
  else
    m = merge(h, h')
    if set(m) = set(h) then return m else raise Incompatible

leq(h, h'): return set(h) = set(h') and h = merge(h, h')
```

FIGURE 2.4. Functions operating on index set sequences.

The function `maxPrefix` is used for removing summation indices from a solution for a summation node together with any indices whose loops are constrained to be within the summation loops. The helper function `merge`, which is used by `mergeNesting` and `leq`, constructs the least constraining indexset sequence that is more or equally constraining than its arguments; it returns `nil` if the arguments are not compatible. The function `mergeNesting` is used for merging extended nestings from the two subtrees of a binary node. The boolean function `leq(`h`,`h'`)` is true if h is more or equally constraining than h', i.e., $h \sqsubseteq h'$. If h and h' do not have the same sets of indices or are incomparable, or if h is not more or equally constraining than h', `leq(`h`,`h'`)` is false. This makes `leq` a total function and allows both fusions and nestings to be compared.

The main part of the fusion algorithm is shown in Figure 2.5. The calculation of the cost of individual solutions has been factored out into the cost model shown in Figure 2.6. To allow for generalizations of

the cost model, each of the cost model functions returns a set containing a single solution. The heart of the algorithm is the function `minMemFusionSet`, which traverses the expression tree bottom-up and constructs a set of solution trees in every step.

```
minMemFusion(t): return head(reduceSolnSet(minMemFusionSet(t), nil))

minMemFusionSet(t):
  if t is an Array, Fun, or Const node then
    return { Leaf(s) | s in makeLeafSoln(t)) }
  else if t is of the form Sum(i, l) then
    return makeUnarySolnSet(extendSolnSet(minMemFusionSet(l), t, true), t)
  else if t is of the form BinOp(o, l, r) then
    return makeBinarySolnSet(extendSolnSet(minMemFusionSet(l), t, false),
                             extendSolnSet(minMemFusionSet(r), t, false), t)

makeUnarySolnSet(S1, t):
  U = { Unary(s, s1) | s1 in S1 and
                       s in makeUnarySoln(getSoln(s1), getNesting(s1), t) }
  return reduceSolnSet(U, fusible(t))

makeBinarySolnSet(S1, S2, t):
  B = { Binary(s, s1, s2) | s1 in S1 and s2 in S2 and
                            s in makeBinarySoln(getSoln(s1), getNesting(s1),
                                                getSoln(s2), getNesting(s2), t)
                            where exception Incompatible was not raised }
  return pruneSolnSet(B)

extendSolnSet(S, t, u):
  s = getSoln(head(S))
  f = getFusion(s)
  i = indices(t)
  if f = any then return { Extended(s, extendNesting(f, i), head(S)) }
  else
    E = Union({ e | st in S and e = extendSoln'(st, i, u) })
    return pruneSolnSet(E)

extendSoln'(st, i, u):
  s = getSoln(st)
  f = getFusion(s)
  if (u or length(f) = 1) and (set(f) = i or subset(i, set(f))) then return { st }
  else return { Extended(s', f'', st) | f' in prefixes(f)
                                        and s' in extendSoln(s, f')
                                        and f'' = extendNesting(f', i) }

reduceSolnSet(S, i):
  R = { Reduced(r, i, s) | s in S and r in reduceSoln(getSoln(s), i) }
  return pruneSolnSet(R)

pruneSolnSet(S):
  return { s | s in S and
               not exists s' in S - {s}: inferior(getSoln(s), getNesting(s),
                                                  getSoln(s'), getNesting(s')) }
```

FIGURE 2.5. The loop fusion algorithm.

For a leaf node in the expression tree, `minMemFusionSet` calls `makeLeafSoln` to construct an individual solution and returns a set with a single solution tree Leaf node containing this solution. For an Array node or a Const node, `makeLeafSoln` constructs a single solution with maximal fusion $\langle * \rangle$ and cost zero indicating that the node can be fused with any index of the parent. For a Fun node, `makeLeafSoln` constructs a solution with a maximal fusion that indicates that all indices of the node can be fused in any

```
makeLeafSoln(t):
  if t is an Array node or a Const node then
    return { S(any, 0, t) }
  else if t is a Fun node then
    return { S([fusible(t)], 1, t) }

makeUnarySoln(S(f, c, l), h, t):
  return { S(h, c + 1, t) }

makeBinarySoln(S(f, c, l), h, S(f', c', r), h', t):
  f'' = mergeNesting(h, h')
  return { S(f'', c + c' + 1, t) }

extendSoln(S(f, c, t), f'):
  return { S(f', c - fusedSize(t, f) + fusedSize(t, f'), t) }

reduceSoln(S(f, c, t), i):
  f' = maxPrefix(f, i)
  return { S(f', c - fusedSize(t, f) + fusedSize(t, f'), t) }

inferior(S(f, c, t), h, S(f', c', t'), h'):
  return c >= c' and leq(h, h')
```

FIGURE 2.6. The cost model for static memory allocation.

order. With all indices fused, only a scalar is needed to hold the result, which is represented as cost one. (For simplicity, we count array elements instead of bytes.)

For a BinOp node and a Sum node, `minMemFusionSet` first recursively constructs the sets of solutions for the subtrees, extends these solutions to the indices of the parent, and then constructs solutions for the parent. In both cases, it is necessary to enumerate all possible ways in which the loop structures for the subtrees can be embedded within a loop structure for the parent. This is achieved by the functions `extendSolnSet` and `extendSoln'`. A boolean parameter is passed along as context to these functions to indicate whether the parent is a unary or a binary node for use in reducing the search space.

While the maximal fusion in a subtree solution results in the largest memory reduction for the root of the subtree under the fusion constraints from that subtree, it may put too many constraints on fusion choices higher up the tree or for the second subtree of a binary node. Fusing only some of the loops included in the maximal fusion would increase the memory requirements for the root of that subtree but may result in a larger memory reduction in a sibling or in an ancestor node. The function `extendSoln'` constructs partial solutions for the parent out of a single solution for the subtree by enumerating all the prefixes of the maximal fusion and then constructing extended nestings for the parent out of these prefixes. A prefix of the maximal fusion becomes an actual fusion between the child and the parent. The cost for an extended solution is calculated in `extendSoln` (in Figure 2.6); the fused size of the child node under the maximal fusion is subtracted from the cost of the subtree and the cost under the actual fusion is added in. If the actual fusion is

not identical to the maximal fusion, this will result in an increase of the cost. The extended solution and the extended nesting for the parent are then recorded in an Extended solution tree node.

If the maximal fusion of a subtree does not limit the fusion choices elsewhere in the tree, it is not necessary to enumerate all its prefixes. For a binary node, this is the case if the child has all the indices of the parent and the maximal fusion fuses all indices without constraints on the loop order. The child can be fully fused with the binary node without effecting other fusion choices. For a unary node, the enumeration of prefixes can similarly be skipped if all indices are fused. However, this is the case even if there are constraints on the loop order. In this case, fully fusing the child with the unary node will prune out some fusion choices between the unary node and its parent, but those would have resulted in higher memory usage. If the generation of extended solutions can be skipped, `extendSoln'` simply returns the singleton set containing the subtree solution.

If there is a single solution with fusion $\langle * \rangle$ for the subtree, the function `extendSoln` constructs an extended nesting that allows all the indices of the parent to be permuted in any order. Otherwise, `extendSoln` calls `extendSoln'` for every solutions of the subtree and returns the union of the resulting sets of extended solutions. Inferior solutions are then pruned out by calling `pruneSolnSet`.

For a BinOp node, `minMemFusionSet` calls `makeBinarySolnSet` on the two sets of extended solutions for the subtrees, which then calls `makeBinarySoln` on all pairs of solutions in the cross product of these two sets. In `makeBinarySoln`, the extended nestings from the two subtrees are merged to form a nesting for the BinOp node. If this merge is successful, a solution is constructed with the merged nesting as the maximal fusion. The memory requirement assuming full fusion with the parent is simply the sum of the costs of the subtrees plus one. In `makeBinarySolnSet`, Binary solution tree nodes are constructed for all pairs of subtree solutions for which the merge was successful, and inferior solutions are pruned from the resulting set.

For a summation node, `minMemFusionSet` calls `makeUnarySolnSet` on the set of extended solutions for the subtree, which then constructs unary solutions using the extended nesting as the maximal fusion. The memory cost, computed in `makeUnarySoln`, is only one more than the cost of the subtree assuming full fusion would be possible with the parent. Since the summation indices as well as any indices that are constrained to be nested within the summation indices cannot be fused with the parent, they must be removed from the maximal fusion for the summation node. This is done in `reduceSolnSet`. For each

solution, `reduceSolnSet` calls `reduceSoln`, which removes unfusable indices and adjusts the cost correspondingly. If any non-summation index is removed from the maximal fusion, the cost will increase. Reduced solutions are encapsulated in Reduced solution tree nodes (mostly for debugging purposes), and inferior solutions are pruned from the resulting set. In this pruning step, `inferior`, which is called from `pruneSolnSet`, compares fusions instead of extended nestings. This pruning step is safe since `leq` is false if two fusions do not have the same sets of indices.

Eventually, a set of solutions will be obtained for the root of the expression tree. These solutions will contain different maximal fusions and different costs. Since the root of the expression tree cannot be further fused, the top-level function `minMemFusion` calls `reduceSolnSet` to remove all indices from the maximal fusions. Since all reduced solutions then have $\langle\rangle$ as the maximal fusion, only one solution will be left after pruning. The solution tree containing this solution is then returned by `minMemFusion` as the optimal solution for the expression tree.

The solution returned by `minMemFusion` assumes that the array for the root node must be produced in its entirety. Correspondingly, the cost for this solution includes the size of the result array. Another possible scenario is that the production of the root array is fused with a write operation as the consumer. In this case, the only dimensions of the root array that need to be stored are those that are inside any summation loops. The optimal solution for this scenario can be obtained by generalizing the algorithm to allow expression tree nodes representing write operations or by simply omitting the final reduction step, selecting the solution with the minimum cost returned by `minMemFusionSet`, and inserting the write operation inside the innermost non-summation loop.

2.4.3 Code Generation

The solution tree data structure that is constructed by our algorithm is designed to efficiently represent the information needed by the algorithm, but it is not convenient for generating code. In particular, the Extended and Reduced nodes are not useful anymore once the optimal solution is found. Also, the fusion stored in a solution summarizes only the loop nesting constraints of the subtree.

To simplify code generation, we first translate the solution tree into a fusion tree representation in which for each node in the expression tree we store the nesting and the fusion with the parent. (Since the fusion is a prefix of the nesting, only the suffix of unfused loops needs to be stored for the nesting.) This fusion tree

representation is generated in a top-down traversal of the solution tree. In this traversal, any constraints on loop nestings from higher up in the solution tree are propagated down to the leafs. The resulting fusion tree does not have any Extended and Reduced nodes, and the fusions and nestings stored at each node contain the full loop nesting information. This data structure transformation is straightforward and does not warrant further discussion. We will present an example in the next section.

When generating the fusion tree, $\langle * \rangle$ fusions for Array and Const nodes are also replaced by the actual fusions. In doing so there is an opportunity for reducing the memory access cost. For the purpose of the `minMemFusionSet` traversal, we have assumed that Array nodes can be accessed repeatedly at zero cost in order to enable additional loop fusions for siblings or ancestors. In `extendSolnSet`, a $\langle * \rangle$ fusion was extended to the fusion containing all the indices of the parent with no constraints. Any indices of the parent that were later constrained to be inside the indices of the Array node can now be unfused, which results in lower memory access cost without any change in the memory requirements.

For generating code, the evaluation order must be determined. Conceptually, this can be done by topologically sorting the nodes in the fusion tree. Let v and v' be nodes of the expression tree with fusions f and f', respectively. v is evaluated before v' if

- v' is an ancestor of v, or
- neither is an ancestor of the other and *prefix*(f, f').

E.g., suppose v and v' are siblings with parent u. If $f = f'$, v and v' can be evaluated in any order. If f is a prefix of f' but $f \neq f'$, more loops are fused between v' and u than between v and u. The evaluation of v' must, therefore, follow that of v.

Once the evaluation order is obtained, it is straightforward to generate explicit loops. For each node in the expression tree, a temporary is created to hold the value of this node. Only dimensions that are not fused must be stored in the temporary. The initialization of a temporary that will hold the result of a summation node is placed just before the outermost summation loop.

The code can also be generated together with determining the evaluation order in a single bottom-up traversal of the fusion tree. For each node, the code generation algorithm constructs an expression for accessing the value of this node and a list of indexset-code pairs. The expression is used for constructing the

code of the parent. Each indexset-code pair consists of a code fragment and a set of fused indices. The latter represents the loops that still need to be generated to surround the code fragment.

Code fragments with equal fusions from different subtrees are merged. Loops that are not fused with the parent are generated as part of the code fragment for the child. Any code fragments from subtrees that are fused with the generated loops are inserted according to the topological sorting order above. The initialization code for summations is also generated as a code fragment with the same set of fused indices as the summation node itself. This way, the initialization code will get inserted into the code before the outermost summation loop. At the root of the tree all the code fragments will have been combined into the complete code for the expression tree.

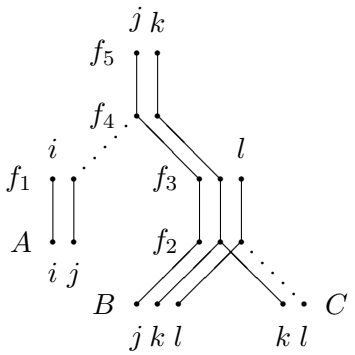
2.4.4 A Simple Example

To illustrate how the algorithm works, consider again the empty fusion graph in Figure 2.2(d) for the expression tree in Figure 2.1(c). Let $N_i = 500$, $N_j = 100$, $N_k = 40$, and $N_l = 15$. Table 2.1 shows the solution sets for all nodes. The column "solution" enumerates the solutions; the column "children" contains the solution numbers of the corresponding solutions for the subtrees. The column "max. fusion" shows the maximal fusion and "ext./red." indicates the extension or reduction operation that is performed. An extension operation is shown with the extended nesting for the parent; a reduction operation is shown with the set of indices to which solutions are reduced. The "pruning" column indicates solutions that do not need to be extended or that are inferior to other solutions. A $\surd$ mark indicates that the solution is part of an optimal solution tree for the entire expression tree. The fusion graph for the optimal solution is shown in Figure 2.7(a).

A and B can be fully fused with their parents, since they contain all the indices of the parent without fusion constraints. All other fusions for A and B with their parents would result in more constraining nestings or use more memory. However, this is not the case for the fusion between C and f_2, since the extended nesting $\langle kl, j\rangle$ resulting from the full fusion $\langle kl\rangle$ is not less constraining than those of the other three possible fusions. Merging the solution for B with each of the solutions for C results in four solutions for f_2, each of which can be fully fused with f_3. After reducing the solutions for f_3 to remove the summation index l and extending them to the indices of f_4, we are left with seven solutions, solutions 20 and 24–29, of which only solutions 20 and 26 are left after pruning. These two solutions are then merged with the two solutions that resulted from extending f_1. Since 5 and 26 have incompatible extended nestings, only three solutions are produced

TABLE 2.1. Trace of the algorithm for the example from Figure 2.1.

Node	solution	children	max. fusion	ext./red.	pruning	cost	opt
A	1		$\langle ij \rangle$		no extension	1	$\surd$
$f1$	2	1	$\langle ij \rangle$			2	$\surd$
$f1$	3	2	$\langle j \rangle$	R $\{j\}$		2	$\surd$
$f1$	4	3	$\langle\rangle$	E $\langle jk \rangle$		101	$\surd$
	5	3	$\langle j \rangle$	E $\langle j, k \rangle$		2	
B	6		$\langle jkl \rangle$		no extension	1	$\surd$
C	7		$\langle kl \rangle$			1	$\surd$
C	8	7	$\langle\rangle$	E $\langle jkl \rangle$		600	
	9	7	$\langle k \rangle$	E $\langle k, jl \rangle$		15	$\surd$
	10	7	$\langle l \rangle$	E $\langle l, jk \rangle$		40	
	11	7	$\langle kl \rangle$	E $\langle kl, j \rangle$		1	
$f2$	12	6, 8	$\langle jkl \rangle$		no extension	602	
	13	6, 9	$\langle k, jl \rangle$		no extension	17	$\surd$
	14	6, 10	$\langle l, jk \rangle$		no extension	42	
	15	6, 11	$\langle kl, j \rangle$		no extension	3	
$f3$	16	12	$\langle jkl \rangle$			603	
	17	13	$\langle k, jl \rangle$			18	$\surd$
	18	14	$\langle l, jk \rangle$			43	
	19	15	$\langle kl, j \rangle$			4	
$f3$	20	16	$\langle jk \rangle$	R $\{jk\}$	no extension	603	
	21	17	$\langle k, j \rangle$	R $\{jk\}$		18	$\surd$
	22	18	$\langle\rangle$	R $\{jk\}$		4042	
	23	19	$\langle k \rangle$	R $\{jk\}$		103	
$f3$	24	21	$\langle\rangle$	E $\langle jk \rangle$	inferior to 20	4017	
	25	21	$\langle k \rangle$	E $\langle k, j \rangle$	inferior to 26	117	
	26	21	$\langle k, j \rangle$	E $\langle k, j \rangle$		18	$\surd$
	27	22	$\langle\rangle$	E $\langle jk \rangle$	inferior to 20	4042	
	28	23	$\langle\rangle$	E $\langle jk \rangle$	inferior to 20	4003	
	29	23	$\langle k \rangle$	E $\langle k, j \rangle$	inferior to 26	103	
$f4$	30	4, 20	$\langle jk \rangle$		no extension	705	
	31	4, 26	$\langle k, j \rangle$		no extension	120	$\surd$
	32	5, 20	$\langle j, k \rangle$		no extension	606	
$f5$	33	30	$\langle jk \rangle$			706	
	34	31	$\langle k, j \rangle$			121	$\surd$
	35	32	$\langle j, k \rangle$			607	
$f5$	36	33	$\langle k \rangle$	R $\{k\}$	inferior to 37	706	
	37	34	$\langle k \rangle$	R $\{k\}$		121	$\surd$
	38	35	$\langle\rangle$	R $\{k\}$		646	
$f5$	39	37	$\langle\rangle$	R $\{\}$		160	$\surd$
	40	38	$\langle\rangle$	R $\{\}$	inferior to 39	646	




```
initialize f1
for i
┌ for j
│ ┌ fA = A(i,j)
│ └ f1[j] += fA
initialize f5
for k
┌ for l
│ [ fC[l] = C(k,l)
│ for j
│ ┌ initialize f3
│ │ for l
│ │ ┌ fB = B(j,k,l)
│ │ │ f2 = fB × fC[l]
│ │ └ f3 += f2
│ │ f4 = f1[j] × f3
└ └ f5[k] += f4
```

(a) (b)

FIGURE 2.7. An optimal solution for the example from Figure 2.1.

for f_4, which are fully fused with f_5. After reducing f_5, one solution can be pruned out, and after reducing again to the empty set of indices for storing the result, we are left with the optimal solution 39.

The optimal solution for the entire tree uses suboptimal solutions for C and f_1. Solution 11, which would have fully fused C, resulted in Solution 19, with nesting $\langle kl, j\rangle$, for f_3. Since the summation index was not the innermost index, the j dimension needed to be stored in reducing f_3, which resulted in both of the extended solutions 28 and 29 to be pruned out. Solution 5, which would have fully fused f_1, eventually resulted in Solution 40, which was pruned out in the last pruning step after reducing the solutions for f_5 to the empty set of indices.

In the top-down traversal for generating the fusion tree from the solution tree, fusion constraints from higher up in the tree are propagated down. This results in the fusions for f_2 and B both to become $\langle k, j, l\rangle$, instead of $\langle k, jl\rangle$ and $\langle jkl\rangle$, respectively.

The fusion graph representing the optimal solution and the code generated from it are shown in Figure 2.7(a) and Figure 2.7(b), respectively. The code could be improved by combining consecutive assignments. E.g., the assignments to `fB`, `f2`, and `f3` could be combined into the statement

```
f3 += B(j,k,l) + fC[l]
```

This would result in removing the scalars `fA`, `fB`, `f2`, and `f4`. Correspondingly, the cost model could be refined such that scalars that are fully fused with their consumers are not counted in the computation of the memory requirements.

2.4.5 A Realistic Example

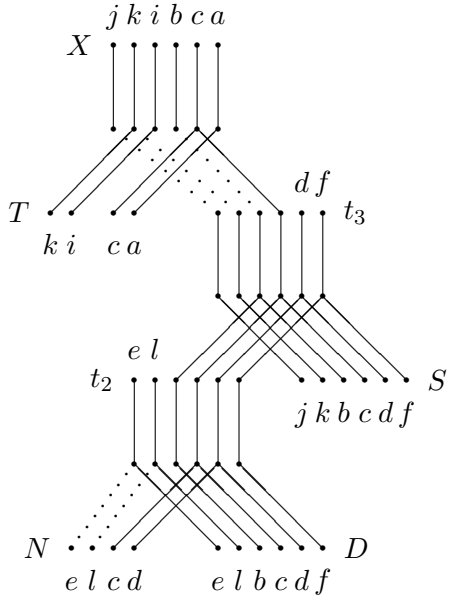
For illustrating the memory reduction that can be achieved, consider the following tensor contraction term, which reflects the type of equations found in quantum chemistry:

$$X[a,b,i,j] = \sum_{c,k} T(a,c,i,k) * \sum_{d,f} S[d,j,f,k] * \sum_{e,l} N(c,d,e,l) * D[b,e,f,l]$$

Suppose the ranges of indices a through d are 300, the ranges of e and f are 70, the ranges of i to l are 40, and array elements are double precision floating point numbers. Given these dimensions, suppose we read the input tensors T and N of sizes 1.073GiB and 1.878GiB, respectively, from disk, while we have allocated the smaller tensors S and D of sizes 256.35MiB and 448.61MiB, respectively, in memory. Similarly, the result tensor X of size 1.073GiB could be written to disk.

For the fusion algorithm to distinguish disk arrays from memory-resident arrays, we represent disk arrays as function calls. The fusion graph and code for the optimal solution with the result in memory are shown in Figure 2.8. In this solution, D is reread from memory for every iteration of the c and d loops, while S is reread for every iteration of the b and c loops. This allows a substantial reduction of the memory usage for the temporaries. The temporaries `t0` and `t2` become scalars, while `t1` and `t3` are of sizes 1.969KiB and 3.662MiB, respectively. The tensors S and D in memory, the temporaries, and the result tensor X add up to a memory usage of 1.765GiB. Without loop fusion, the memory usage of `t2`, at 14.08GiB, would have prevented an in-core computation.

With this loop structure, writing X to disk will not result in a reduction of memory usage. Since the summation loop c is the outermost loop, the production of X could not be fused with a write operation. However, by not fusing `t3` with its parent and by not fusing the summation loops c and k between T and its parent, the summation loops c and k become the innermost loops and the production of X can be fully fused with a write operation as shown in Figure 2.9. In this solution, the sizes of `t0` and `t3` are increased to 93.75KiB and 1.073GiB, respectively, while X becomes a scalar. The total memory usage is only slightly reduced to 1.761GiB.

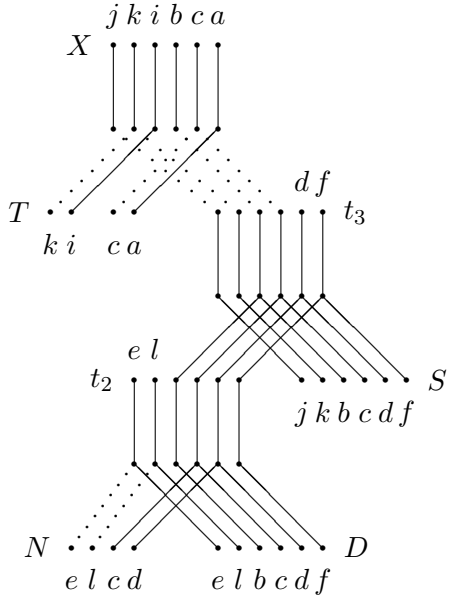


(a)

```
for a,b,i,j
[ X[a,b,i,j] = 0
for c
[ for b,j,k
  [ t3[b,j,k] = 0
  for d
  [ for e,l
    [ t1[e,l] = N(c,d,e,l)
    for b,f
    [ t2 = 0
      for e,l
      [ t2 += t1[e,l] × D[b,e,f,l]
      for j,k
      [ t3[b,j,k] += S[d,j,f,k] × t2
  for a,i,k
  [ t0 = T(a,c,i,k)
    for b,j
    [ X[a,b,i,j] += t0 × t3[b,j,k]
```

(b)

FIGURE 2.8. The optimal solution for producing $X[a, b, i, j]$ in memory.



(a)

```
for b,c,j,k
[ t3[b,c,j,k] = 0
for c,d
[ for e,l
  [ t1[e,l] = N(c,d,e,l)
  for b,f
  [ t2 = 0
    for e,l
    [ t2 += t1[e,l] × D[b,e,f,l]
    for j,k
    [ t3[b,c,j,k] += S[d,j,f,k] × t2
for a,i
[ for c,k
  [ t0[c,k] = T(a,c,i,k)
  for b,j
  [ X = 0
    for c,k
    [ X += t0[c,k] × t3[b,c,j,k]
    write X
```

(b)

FIGURE 2.9. The optimal solution for producing $X[a, b, i, j]$ on disk.

2.4.6 Alternative Cost Models

For presenting the algorithm, we have used a simple cost model for minimizing memory usage under the assumption that all temporaries are allocated statically. The algorithm, however, is general enough to be used with a variety of cost models. In this section, we demonstrate this flexibility of our algorithm.

In the problem definition, we allowed arrays in leaf nodes to be either memory resident or generated by a generator function. It is straightforward to force array elements to be produced in a certain order by restricting the maximal fusion for the leaf node. E.g., if array A in our example is stored on disk in row-major order, we can ensure that it is read consecutively by modeling it as a generator function, but using the maximal fusion $\langle i, j \rangle$ instead of $\langle ij \rangle$. Similarly, a read operation that reads an entire row into a buffer can be modeled by using the maximal fusion $\langle i \rangle$.

Instead of simply minimizing memory usage, it may be more practical to minimize another quantity, such as disk I/O or communication, under a given memory constraint. Another possibility is to use our algorithm for producing a set of suitable candidate loop structures for different tradeoffs between two or more cost components, out of which, after further optimization, the best solution is selected. We illustrate this optimization approach using a space-time tradeoff cost model that considers trading recomputation for an additional reduction in memory usage [CBL$^+$02b]. We have also used the loop fusion algorithm for minimizing disk I/O cost [BKC$^+$03, KKB$^+$03] and for minimizing communication cost [CBL$^+$02a, CGK$^+$03] under given memory constraints.

We also outline a cost model for memory minimization that uses the more realistic assumption that intermediates are allocated and deallocated dynamically as needed.

2.4.7 Space-Time Tradeoffs

So far, we have been restricting our attention to reducing memory usage without any increase in the number of arithmetic operations. However, with this restriction, the optimal loop fusion configuration for some equations may still require more than the available memory.

If not enough memory is available for a computation to fit in memory, an out-of-core solution can be produced by tiling arrays and moving entire tiles between disk and memory. However, in certain cases it may be cheaper to recompute parts of a computation instead of paying the penalty for disk I/O. If a computation

is that large that after loop fusion some of the intermediates do not even fit on disk, as can be the case in quantum chemistry, there is no choice but to use recomputation to further reduce the storage requirements.

Creating additional loops around some assignment statements may enable more loop fusions and thus reduce array sizes. To do this, we add additional vertices to the fusion graph and connect each new vertex of a node v and the corresponding vertex at the parent of v with an additional potential fusion edge. If the additional potential fusion edge is converted into a fusion edge in a fusion graph, the operation count for node v is multiplied by N_i, where i is the loop index for the fusion edge. For example, in Figure 2.7(b), if we create a k-loop around the computation of A and f_1 and fuse it with the k-loop around the other nodes, we can eliminate the j dimension of f_1 and make it a scalar. However, this increases the operation counts for A and f_1 by a factor of N_k because they are recomputed $N_k - 1$ times.

If a node ends up not fully fused with its parent in the memory-minimal solution, it is primarily because it is lacking an index from its parent that is fused with another subtree. E.g., in the optimal solution in Figure 2.7, f_1 cannot be fused with f_4, because the k-dimension of f_4 is fused with f_3. Similarly, C cannot be fully fused with f_2, because the j-dimension of f_2 is fused with B. Our strategy for enumerating possible space-time tradeoffs is, therefore, to consider as recomputation loop for a node v any loop that is needed for an ancestor of v but not for v itself.

The cost model for space-time tradeoffs is shown in Figure 2.10. A solution $\mathrm{S}(f, c, r, t)$ for an expression tree t contains the maximal fusion f with the parent, the memory cost c assuming static memory allocation of intermediates, and the recomputation cost r. The computation of the memory cost is the same as in the static memory allocation cost model in Figure 2.6. The recomputation cost of an interior node, which is computed by function `recompCost`, is the size of the node times the product of the ranges of recomputation indices minus one.

Instead of a single solution, `makeLeafSoln` creates a set of solutions for a leaf node, one solution for each combination of recomputation indices. The recomputation cost of memory-resident arrays and constants is zero. For a function call v, the recomputation cost is computed as `recompCost(`v`)` times `functionCost(`v`)`, where the latter is an estimate of the cost of a single function call.

A solution s is inferior to another solution s' if, in addition to the existing criteria, the recomputation cost for s is greater than or equal to that for s'. Furthermore, any solution with a cumulative memory usage higher than a user-specified limit can also be pruned.

```
recompIndices(v): return { i | i in indices(v') for some ancestor v' of v
                               and i not in indices(v) }

recompCost(v, f):
  if v is an Array node or a Const node then return 0
  else return Product of range(i) for i in indices(v) *
              ((Product of range(ri) for ri in set(f) - indices(v)) - 1)

functionCost(v): return computation cost for Fun call v

makeLeafSoln(t):
  if t is an Array node or a Const node then
    return { S(f, 0, 0, t) |  ri in powerset(recompIndices(t)) and
                              i = union(fusible(t), ri) and
                              f = if i is empty then [] else [i] }
  else if t is a Fun node then
    return { S(f, c, rc, t) | ri in powerset(recompIndices(t)) and
                              i = union(fusible(t), ri) and
                              f = if i is empty then [] else [i] and
                              c = fusedSize(t, f) and
                              rc = recompCost(t, f) * functionCost(t) }

makeUnarySoln(S(f, c, rc, l), h, t):
  return { S(h, c + 1, rc + recompCost(t, f), t) }

makeBinarySoln(S(f, c, rc, l), h, S(f', c', rc', r), h', t):
  f'' = mergeNesting(h, h')
  return { S(f'', c + c' + 1, rc + rc' + recompCost(t, f''), t) }

extendSoln(S(f, c, rc, t), f'):
  return { S(f', c - fusedSize(t, f) + fusedSize(t, f'), rc, t) }

reduceSoln(S(f, c, rc, t), i):
  f' = maxPrefix(f, union(i, recompIndices(t)))
  return { S(f', c - fusedSize(t, f) + fusedSize(t, f'), rc, t) }

inferior(S(f, c, rc, t), h, S(f', c', rc', t'), h'):
  return c > MemoryLimit or (c >= c' and rc >= rc' and leq(h, h'))
```

FIGURE 2.10. A space-time tradeoff cost model for static memory allocation.

Since the fused indices of a child are used to form the nesting of the parent, simply selecting all the possible recomputations for the leaf nodes has the effect of propagating those recomputation choices to the root. At the end, the algorithm computes a set of space-time tradeoff solutions for the entire expression tree. E.g., for our running example the loop fusion algorithm with a space-time tradeoff cost model finds three solutions: the memory minimal solution without recomputation shown in Figure 2.7 at a memory cost of 160, a solution of memory cost 61 in which A and f_1 have k as a recomputation loop, and a solution of memory cost 47 in which C is recomputed in the j loop. In the latter solution, all nodes except for the root have been fused to scalars.

Instead of simply choosing the solution with the lowest recomputation cost that still fits in memory, we can reduce the recomputation cost further using tiling. Recomputation loops are split into tiling loops and intra-tile loops. While the tiling loops maintain the fused loop structure that was found by the loop fusion algorithm, the intra-tile loops are moved inside all of the tiling loops such that only tiling loops remain as recomputation loops. By expanding fused dimensions to a larger tile size, the memory requirements increase while the recomputation cost decreases. Each of the candidate loop structures found by our algorithm is tiled in this manner. After determining the optimal tile sizes that minimize the recomputation cost while remaining below the memory limit, the best solution overall is chosen [CBL$^+$02b].

As discussed above, a similar optimization approach can be used for minimizing disk I/O or communication cost under a memory constraint. In the case of communication minimization [CBL$^+$02a, CGK$^+$03], after the fusion algorithm found a set of candidate loop structures with different communication and memory tradeoffs, the communication-minimal solution that fits in memory is chosen. In the case of data locality optimization [BKC$^+$03, KKB$^+$03], the loop fusion algorithm uses only an approximate cost for modeling the disk I/O in order to produce candidate loop structures. After tiling, a tile size search on the abstract syntax trees for these loop structures selects the optimal solution using a more precise cost model.

The space-time tradeoff cost model described here does not enumerate all possible solutions. Suppose that each element of array C in Figure 2.7 is orders of magnitude more expensive to compute than an element of array A. In that case, recomputing C $N_j - 1$ times may be too expensive for reducing all nodes to scalars. An alternative solution would be to make l and k the outermost loops by making k a recomputation loop for A and f_1 and l a recomputation loop for A, f_1, f_4, and f_5. Since this solution requires using the summation

loop l for node f_3 as a recomputation loop in ancestors and the sibling of f_3, this solution is not found using the cost model in Figure 2.10.

It is possible to extend the space-time tradeoff cost model to consider summation loops as recomputation loops for ancestors and siblings as shown in Figure 2.11. Instead of producing a single solution, `reduceSoln` would then produce one solution for each combination of fused summation indices. However, this would increase the search space dramatically for very little practical benefit.

```
recompIndices(v): return { i | (i in indices(v') for some ancestor v' of v
                                 or i is a summation index anywhere in the tree)
                               and i not in indices(v) }

reduceSoln(S(f, c, rc, t), i):
  if i = {} then
    return { S(<>, c - fusedSize(t,f) + fusedSize(t, <>), rc, t) }
  else
    return { S(f', c - fusedSize(t, f) + fusedSize(t, f'), rc, t) |
                     si in powerset(indices(t) - i) and
                     f' = maxPrefix(f, union(union(i, si), recompIndices(t))) }
```

FIGURE 2.11. Modifications for the cost model to allow summation loops as recomputation loops.

Chapter 3
Related Work

3.1 Loop Fusion Optimization

Aspects of some of the important problems addressed in the synthesis system such as operation minimization, memory reduction, and locality optimization have also received some attention in research on compiler optimizations. Reduction of arithmetic operations has been traditionally done by compilers using the technique of common subexpression elimination. Much work has been done on improving locality and parallelism by loop fusion [KM94], [MA97], [SM96]. The TCE considers a different use of loop fusion, which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. The contraction of arrays into scalars through loop fusion is studied in [GOST93] but is motivated by data locality enhancement and not memory reduction. Loop fusion in the context of delayed evaluation of array expressions in APL programs is discussed in [GW78], but their work is also not aimed at minimizing array sizes; in addition, they consider loop fusion without considering any loop reordering. Some work has explored the use of loop fusion for memory reduction for sequential execution. Strout et al. [SCFS98] present a technique for determining the minimum amount of memory required for executing a perfectly nested loop with a set of constant-distance dependence vectors. Fraboulet et al. [FHM99] use loop alignment to reduce memory requirement between adjacent loops by formulating the one-dimensional version of the problem as a network flow problem. Song [Son00] and Song et al. [SWL03, SXWL01] present a different network flow formulation of the memory reduction problem and they include a simple model of cache misses as well. However, they do not consider the issue of trading off memory for recomputation.

Darte [Dar99] showed that optimal fusion is NP-complete in general. A typed fusion heuristic which can achieve maximal fusion for loops of a particular type was proposed by Kennedy and McKinley [KM93]. Zhao et al [ZYK$^+$05] parametrize the output of loop fusion transformation algorithms so that an external empirical search engine can take advantage of a compiler's knowledge to find the optimal operating point within the space of legal fusion configurations and generate optimized code. Hu et al. [HVP$^+$06] explore loop fusion and loop shifting for improving data locality to reduce the number of costly accesses to off-chip memories, where the focus is on Scratch-Pad Memory (SPM) based memory hierarchy. For each loop nest,

a data reuse analysis phase builds data reuse trees, which represent different ways to copy data across the memory hierarchy. A Memory Hierarchy Layer Assignment (MHLA) estimation phase determines which data needs to be stored into the SPM, such that the number of (energy consuming) off-chip memory accesses is minimized. Energy impact of loop fusion is also studied in [ZMS+04, MCG04] where the general goal is to get maximum energy savings for the same or improved performance benefits for DVS processors and embedded systems. Ashby and Boyle[AO06] propose a technique called *iterative collective loop fusion* which employs a heuristically guided search to find legal fusion partitions (which have lesser partition size and more scope for array contraction) that improve temporal locality and ultimately performance irrespective of the architecture. The runtimes of these legal fusion partitions are empirically tested and the best one is chosen. Grelck et al. [GHS06] explore fusion in a functional array processing language designed for numerical applications with a goal to reduce loop overhead and memory footprint size by changing the order of references into existing arrays in a way that improves data locality. They do not use search or any other mechanism to find the most profitable fusion structure. Blainey et al. [BBA05] focus on eliminating conditions that prevent loop fusion. The algorithm scans the code to find pairs of normalized loops that can be fused and greedily fuses them. If any code between the loops has no data dependencies with one of the loops, it is moved either before the first loop or after the second loop to allow fusion between the loops.

Qasem and Kennedy study loop fusion using model-driven empirical search [QK05, QK06, QK08]. The fusion algorithm tries to fuse loops in such a way that performance does not degrade as a result of increased pressure on system resources due to fusion. Their model uses uses machine specific information (e.g. cache line size, latency) in combination with reuse distances in determining if fusion is profitable for a pair of loops. Empirical tuning is used to tune the set of fusion parameters which cannot be measured accurately through static analysis. In a nut shell, fusion is only applied if it is profitable in terms of performance. The model presented does not analyze the complex interaction of tiling and loop fusion. Pike and Hilfinger [PH02] apply tiling and fusion to a set of consecutive perfectly nested loops (each containing one statement) of the same nesting depth. Their work does not apply to the class of loops with complex nestings that are considered here. There has been some work in the area of design automation to estimate storage needed for a single perfectly nested loop or a sequence of such loops [CGS98, KCA03], [RHKN01, RHKN06] and references therein. These techniques do not consider tiling and they incur additional runtime memory management overhead.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [CM95], [Li93], [MCT96], [WL91], [VBJC03], [Xue05], [PMHC03].

Nevertheless, a performance-model-driven approach like that in the TCE to the integrated use of loop fusion and loop tiling for enhancing locality in imperfectly nested loops has not been addressed in these works. Wolf et al. [WMC96] consider the integrated treatment of fusion and tiling only from the point of view of enhancing locality and do not consider the impact of the amount of required memory; the memory requirement is a key issue for the problems considered in context of the TCE.

Loop tiling for enhancing data locality has been studied extensively [CM95], [KAP97, KPCM99], [RS92], [RT99], [SL99], [WL91, WMC96], and analytic models of the impact of tiling on locality in perfectly nested loops have been developed [GMM98], [LRW91], [MHCF98]. Frameworks for handling imperfectly nested loops have been presented in [AMP01], [LLL01], [SL99]. Ahmed et al. [AMP01] have developed a framework that embeds an arbitrary collection of loops into an equivalent perfectly nested loop that can be tiled; this allows a cleaner treatment of imperfectly nested loops. Lim et al. [LLL01] develop a framework based on affine partitioning and blocking to reduce synchronization and improve data locality. Specific issues of locality enhancement, I/O placement and optimization, and automatic tile size selection have not been addressed in the works that can handle imperfectly nested loops [AMP01], [LLL01], [SL99].

The approach undertaken in the TCE bears similarities to some research in other domains, such as the SPIRAL project, which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [JJPX01, MJJ$^+$00, PMJ$^+$05, XJJP01]. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language called SPL by a systematic search through the space of possible implementations. Other efforts in automatically generating efficient implementations of programs include FFTW [FJ98, FJ05], the telescoping languages project [KBC$^+$01, KBC$^+$05], ATLAS [WD98], [DDE$^+$05] for deriving efficient implementation of BLAS routines, and the PHIPAC [BACD97] project. All these efforts use search-based approaches for performance tuning of codes. A comparison of model-based and search-based approaches for matrix-matrix multiplication is reported in [YLR$^+$03]. In addition, motivated by the difficulty of detecting and optimizing matrix operations hidden in array subscript expressions within loop nests, several projects have worked on efficient code generation from high-level languages such as MATLAB and Maple [CE05], [DRP96], [BSC$^+$00], [MP99b], [MP99a].

While the TCE effort shares some common goals with several of the projects mentioned above, there are also significant differences. Some of the optimizations in the TCE, such as the algebraic optimizations, memory minimization, and space-time tradeoffs, do not appear to have been previously explored, to the best of our knowledge. The TCE also takes advantage of certain domain-specific properties of the computations; for example, since all expressions considered in this framework are tensor contractions, the loops of the resulting code are fully permutable, and there are no dependencies preventing fusion. This observation is crucial for the optimization algorithms of several components (memory minimization, space-time transformation, data locality). Also, some of the multidimensional arrays involved in the computation have certain domain-specific symmetry properties that can be exploited in order to lower the number of arithmetic operations, and, thus, total execution time. While optimization of performance is a significant goal, more important in the TCE's context is the potential for dramatically reducing the developmental effort required of a quantum chemist to develop a new ab initio computational model.

3.2 Loop Fusion Optimization for Handwritten Code

In the previous TCE research, a performance-model driven search-based approach to program transformation [BAB+05b, BBC+02, BKC+03, CBL+02b, KKB+03, KKB+04, KKB+06, GSL+05] has been pursued. We have extended this approach to a class of handwritten array computations. The data locality optimization problem arising in this synthesis context, focusing primarily on minimizing memory-to-cache data movement [CWB+01, CWL+01] has also been addressed. In [CWL+01], an integrated approach to fusion and tiling transformations for the class of loops arising in the context of our program synthesis system was developed. However, that algorithm was only applicable when the sum-of-products expression satisfied certain constraints on the relationship between the array indices in the expression. The algorithm developed in [CWB+01] removed the restrictions assumed in [CWL+01]. However, it decoupled the fusion step from the tiling step. The fusion step was first performed, attempting to minimize the total space required by the intermediates. Tiling was then performed on the fused loop structure resulting from the fusion step. While this approach provided very good improvement in practice when compared to unfused or untiled loop structures, an examination of the code structures produced for some examples from the quantum chemistry domain revealed that better solutions were possible. Initially, the approach of letting the fusion algorithm produce a range of candidate code structures from a two-dimensional search space [LCBS99]

was developed, from which the tiling algorithm selected the best overall code structure, in the context of space-time tradeoffs [CBL+02b]. This optimization approach was then applied to generating out-of-core code [BKC+03, KKB+03]. The approach presented in this dissertation generalizes the fusion algorithm from expression trees representing formulas to an abstract syntax tree representing handwritten code satisfying certain constraints. Coupled with an appropriate cost model and a tiling optimization algorithm, the current approach can be used to translate such handwritten code into out-of-core code.

The Pluto [BHRS08a, BBK+08, BHRS08b, BRS10] and PrimeTile [HBB+09b, HBB+09a, BHH+10] projects at LSU and Ohio State University were follow-on efforts at the end of the TCE project, enabling the much broader application of key transformations like loop tiling and fusion that were key to performance optimization for tensor expressions in the TCE project. These efforts have produced publicly-available tools, namely Pluto [BHRS08a] and PrimeTile [HBB+09b]. Tiling is a key transformation in optimizing for parallelism and data locality. For locality, tiling groups points in an iteration space into smaller blocks (tiles), enabling reuse in multiple directions. For coarse-grained parallelism, tiling partitions the iteration space into groups of points that may be concurrently executed and reduces frequency and volume of inter-processor communication. A tile is atomically executed; no communication happens within the tile execution. A significant advantage of the polyhedral compiler model used by these tools is the integrated treatment of arbitrary compositions of transformations on (affine) imperfectly nested loops with multiple statements (which is precisely the structure of tensor contractions handled by TCE), and reasoning about their cumulative effects through algebraic cost functions. Pluto uses a novel model-driven approach to determine tiling hyperplanes to optimize for both data reuse and parallelism. Although automatic parallelization has been the subject of study for several decades, to our knowledge Pluto is the first robust and effective automatic tiling and parallelization system for affine imperfectly nested loops; the technology has been integrated into the ROSE [RSE] compiler infrastructure (developed at Ohio State University and used by the PACE project), Reservoir Labs' R-Stream compiler [RSM] and IBM's XL [IBM] compilers. While Pluto is a powerful transformer, it has the following limitations (i) for tensor contraction expressions, it does not have adequate models to guide the exploration and generation of loop fusion structures; (ii) Pluto is a "black box" tool and is quite inflexible by not allowing users to drive transformations through a script-based interface; and (iii) Pluto lacks support for model-driven and memory-access-oriented transformations that are key for achieving high levels of performance.

3.3 Loop Fusion Optimization for GPGPUs

For the tensor contraction expressions arising in quantum chemistry, we found that automatic parallelization frameworks [VJC+13, BRS10] for general purpose computation on GPUs that generate CUDA code from input sequential C code do not achieve the same levels of performance compared to our approach, primarily because they do not leverage optimized libraries like BLAS. The code generated by these frameworks is too complex for further manual/automated transformation to insert optimized library calls. They also do not employ loop fusion optimization as a means for reducing memory usage.

Ma et al. [MKV+13] demonstrate an approach to automatically generate GPU code for the most expensive tensor contractions in CCSD(T). These contractions are independent and are analogous to a special case discussed in Chapter 6 of this dissertation. The authors map a chain of such binary tensor additions effectively to GPUs. However, they do not consider using fusion as a means of reducing storage requirements on the GPU and also do not consider the cases where the contractions have a producer-consumer relationship. DePrince and Hammond [DH11] have demonstrated GPU optimizations for the iterative solution of CCSD(T). For a total of 34 contractions, they classify the contractions scaling as N^4 or less as work to be performed on the CPU and N^5 or more on the GPU (with the sixth-power contractions executed first), where N refers to the system size. The tensors that are too large to fit on the GPU are then tiled and the tiles are executed sequentially, which is the case for only four contractions in their experiments. While this is a manual optimization for a specific equation, a generalization of this approach may be useful as a future extension of our optimization framework.

Chapter 4
Improvements to the Loop Fusion Algorithm

This chapter discusses the performance improvements we made to the loop fusion algorithm and also present a dynamic memory allocation cost model for the fusion algorithm. For optimizing the performance of the fusion algorithm, there are two main implementation choices: the choice of set data structures and the pruning strategy. As we discuss subsequently, they can depend on the cost model that is used as well as on the structure of the input equations. With these improvements, we outperform the existing implementation of the algorithm by a factor of 3.5 for the static memory allocation cost model and upto a factor of 55 for the space-time tradeoffs cost model. The complete measurements are presented in Section 8.1.

4.1 Data Structures

Operations on index sets are very frequent, especially when extending solutions, constructing binary solutions, and in pruning. Since the number of indices in an equation is bounded and rarely larger than 32, the best choice of data structure is a bit set in which each index is mapped to a unique bit position. Also, instead of implementing `leq` for index set sequences as presented, it is more efficient to implement a four-valued comparison function, `compare(`h`,` h'`)`, that indicates whether h and h' are equal, incomparable, or whether one is more constraining than the other. With similar four-valued functions corresponding to `inferior` and `leqFrags`, this would allow reducing the number of comparisons when pruning a solution set with n solutions from n^2 to $n(n-1)/2$.

While a linked list is an efficient data structure for implementing sets of solution trees for memory minimization, this is not the case for a two-dimensional cost model, such as space-time tradeoffs. With two or more orthogonal cost components, such as memory and recomputation cost, a solution set will contain multiple solutions with the same fusion. When computing the solutions for a binary node, this will result in repeated calls to `mergeNesting` with the same pair of nestings from the left and right subtrees. A more efficient data structure for this scenario is a hashed set data structure that uses the fusion as a hash index to select the subset of solutions that contain the same fusion [Bib04]. By moving the call to `mergeNesting` from the cost model into `makeBinarySolnSet` the number of calls to `mergeNesting` can be drastically

reduced. Another advantage of a hashed solution set data structure is that solutions with equal fusion can be pruned when inserting solutions into the set.

For a one-dimensional cost model, a hashed solution set data structure would also allow pruning solutions with equal fusion when inserting them into the set. While this results in smaller sets, it comes at the expense of computing the hash codes and of more memory accesses for maintaining the data structure.

4.2 Pruning

We first perform an optimization in `extendSoln'` where we omit the extension step in certain cases. If the maximal fusion of a subtree does not limit the fusion choices elsewhere in the tree, it is not necessary to enumerate all its prefixes. For a binary node, this is the case if the child has all the indices of the parent and the maximal fusion fuses all indices without constraints on the loop order. The child can be fully fused with the binary node without effecting other fusion choices. For a unary node, the enumeration of prefixes can similarly be skipped if all indices are fused. However, this is the case even if there are constraints on the loop order. In this case, fully fusing the child with the unary node will prune out some fusion choices between the unary node and its parent, but those would have resulted in higher memory usage. We present a correctness proof for this optimization in Section 4.3. For clarity of the presentation, the code changes corresponding to this optimization in function `extendSoln'` have already been made in the algorithm presented in Chapter 2 (see Figure. 2.5).

The function `pruneSolnSet` is called after extending solutions, after forming binary solutions, and after reducing solutions. Since pruning is an expensive operation — it is quadratic in the size of the solution set — it may be beneficial to prune less frequently. However, which of these three pruning operations are the most effective and are worth their cost can depend on the cost model as well as on the structure of the input expression. For determining the optimal pruning strategy for a given cost model and application domain, it is, therefore, necessary to profile different combinations of pruning operations.

With a two-dimensional cost model, such as space-time tradeoffs, the size of solution sets can become especially large along a long chain of binary addition nodes. For a given fusion, there can be many solutions with different memory and recomputation cost combinations. The size of the solution sets can be reduced further using heuristic pruning [Bib04]. The solutions for a given fusion are sorted in increasing memory cost and decreasing recomputation cost. A heuristic pruning rule that has proved successful is to divide the

range of solutions into a fixed number of equal-size intervals (e.g., along the memory dimension) and to pick at most one solution per interval.

4.3 Correctness and Complexity of the Loop Fusion Algorithm

The `minMemFusion` algorithm exhaustively enumerates loop structures for an expression tree and finds a solution with minimal memory requirements under the assumption that all temporaries are allocated statically. By enumerating all possible prefixes of each maximal fusion for a subtree and by forming the full cross product of solutions from the two subtrees of a binary node, we ensure that the search is exhaustive.

We employ two pruning techniques for pruning out solutions that could not lead to a memory-minimal solution for the entire expression tree. In `pruneSolnSet` we eliminate suboptimal solutions from the set of solutions for a node, and in `extendSoln'` we omit the extension step in certain cases.

The function `pruneSolnSet` is called after extending solutions, after forming binary solutions, and after reducing solutions. The pruning in `pruneSolnSet`, by using the function `inferior` for comparing solutions, removes solutions that have a more or equally constraining fusion and a greater or equal cost than another solution in the set. By only eliminating solutions with a more or equally constraining fusion, we ensure that no fusion possibilities are prevented in a sibling or in an ancestor. By comparing the cost only for solutions with compatible fusions, we ensure that the optimal solution for the subtree is not removed from the set.

In `extendSoln'`, the extension step is skipped

- if the parent is a binary node, the child has the same indices as the parent, and the maximal fusion for the subtree fuses all indices without constraints on the loop order, or
- if the parent is a unary node and the maximal fusion for the subtree fuses all indices regardless of the constraints on the loop order.

In both cases, if there are no constraints on the loop order, it is always optimal to fuse all indices. Since the child and the parent have the same sets of indices, fusing only some of the loops only increases the memory requirements for the child node without any change in the fusion possibilities in a sibling or an ancestor.

Suppose u is a Sum node with child v and the maximal fusion f of v contains all indices of v. Without loss of generality, assume that f has the form $\langle i, \ldots, j, \ldots, k \rangle$, where j is the left-most summation index of u and i and k are non-summation indices. If all indices are fused, after the reduction operation for u removes

the summation loops and all loops nested within the summation loop, the maximal fusion of u still fuses i but not k. If an actual fusion for v still fuses j but not k, the memory requirement for v increases without any change in the fusion possibilities in an ancestor. If an actual fusion for v still fuses i, but neither j nor k, there are additional fusion possibilities between u and its parent, in particular, k can now be fused. But any dimension of u that can now be fused must be stored for v, and in addition the j dimension of v must be stored. There are no additional fusion possibilities for a sibling or ancestor of u. Eliminating the extension step between v and u, therefore, only prunes out sub-optimal solutions.

Since the maximal fusions of BinOp nodes and of all leaf nodes always fuse all indices, the only time the extension step for the child of a summation node is necessary is if that child is a summation node as well and some indices are not fused because they were constrained to be nested within a summation index of the child.

The dynamic programming algorithm in `minMemFusionSet` visits each node in the expression tree exactly once. The number of solutions for a node, however, can grow exponentially with the number of indices, since the set of prefixes of a fusion with n indices that is generated in `extendSolnSet` can contain 2^n elements. Since every Array or Fun leaf node can introduce a unique index, the number of indices can be linear in the size of the tree. The worst-case complexity is, therefore, exponential in the size of the tree. In practice, however, the number of indices is usually small and pruning proves very effective, which results in fairly small sets of solutions and very good overall performance.

4.4 Dynamic Memory Allocation

For simplicity, we had been assuming that all intermediates are allocated statically, but this can unnecessarily increase the memory usage. In this section, we present a cost model that assumes that memory for intermediate arrays and the result array can be allocated and deallocated dynamically as needed. We allow an array to be allocated/deallocated multiple times by putting the allocate/deallocate statements inside loops, but each array must be allocated/deallocated in its entirety. We do not consider sharing of space between arrays.

Given a loop fusion configuration, the positions of the allocate/deallocate statements are determined as follows. Let v be an array, s_p be the statement that produces v, s_c be the statement that consumes v, and the i-loop be the innermost fused loop that contains both s_p and s_c. The latest point an "allocate v" statement can be placed is inside the i-loop and before s_p and any loops that surround s_p and are inside the i-loop. The

earliest point a "deallocate v" statement can be placed is inside the i-loop and after s_c and any loops that surround s_c and are inside the i-loop.

The memory usage of a loop fusion configuration is no longer the sum of the array sizes, but instead the maximum total size of the arrays that are allocated at any time.

Another difference between dynamic allocation and static allocation is that, with dynamic allocation, the evaluation order of the nodes having equal fusion with their parents can affect memory usage even though the sizes of individual arrays remain unchanged.

Let u and v be two nodes that have the same fusion with their respective parents, and let t_u and t_v be the intermediate arrays that hold the results of the computations of the subtrees rooted at u and v, respectively. Since the fusion with their parents is the same, the code for u and v can be generated consecutively within the same outer loops. Suppose that the maximum memory usage during the computation of u is 1000, the size of t_u is 700, the maximum memory usage for v is 500, and the size of t_v is 100. If u is evaluated first, the maximum memory usage is $700 + 500 = 1200$ since t_u must remain in memory while v is evaluated. On the other hand, if v is evaluated first, the maximum memory usage is only $100 + 1000 = 1100$. Therefore, in addition to finding the fusions between children and parents, we also need to find an evaluation order of the nodes for minimizing memory usage.

The simpler problem of finding the memory-optimal evaluation order of the nodes in an expression tree in the absence of loop fusion is the register allocation problem for expression trees. If the expression tree is a binary tree in which all nodes require the same amount of memory, the Sethi-Ullman algorithm [SU70] can be used, which finds an evaluation order in which the subtree that uses the most memory is evaluated first. Appel and Supowit generalized the register allocation problem to higher degree expression trees of arbitrarily-sized nodes [AS87]. However, they restricted their attention to solutions that evaluate subtrees contiguously, which could result in non-optimal memory usage. Lam et al. present an algorithm that finds the optimal evaluation order without requiring subtrees to be evaluated contiguously [LRB$^+$11].

The Appel-Supowit algorithm simply traverses the expression tree bottom up and computes two quantities for each node. The *hi* of a node is the largest amount of memory needed during the evaluation of the subtree rooted at this node. The *lo* of a node is the amount of memory needed for the temporary array that holds the result. If a node has multiple subtrees, the subtrees are evaluated in non-increasing *hi* $-$ *lo* order. The

algorithm by Lam et al. uses the same $hi - lo$ ordering, but for fragments of tree traversals that do not necessarily evaluate entire subtrees. For a proof of the optimality of this ordering, see Lemma 2 in [LRB+11].

While the Appel-Supowit algorithm is not guaranteed to find the optimal evaluation order, for typical tensor contraction expressions in quantum chemistry, however, it finds the same solution as Lam et al.'s algorithm. Either algorithm could be used as the basis of a dynamic memory allocation cost model for the loop fusion algorithm. Because of the constraints placed on the evaluation order by loop fusion, it is unlikely that in practice a non-contiguous evaluation order of subtrees is needed. Therefore, we use the simpler Appel-Supowit algorithm as the basis of our cost model. A dynamic memory allocation cost model based on Lam et al.'s algorithm can be found in [LCBS99, Lam99].

Suppose v and v' are siblings with parent u but do not have the same fusion with u, and suppose loop i is fused between v and u but not between v' and u. The set of fused indices between v' and u is, therefore, a subset of the fused indices between v and u. Since children must be evaluated before their parent, and since v' must be evaluated outside the fused i loop, it must be evaluated first. Reducing the memory usage by evaluating nodes in non-increasing $hi - lo$ order, therefore, is only possible for nodes that have the same fusion with their parents. Furthermore, if the entire subtree rooted at v is fused with v, v' must remain allocated during the evaluation of the entire subtree rooted at v.

The costs of subtrees are not simply added up as with the static memory allocation cost model. Instead, the *lo* of the subtree that is evaluated first is added to both the *lo* and the *hi* of the next subtree. Then the maximum of the two *hi* values is taken as the cost of the two subtrees combined. This cost calculation depends on the order of evaluation of subtrees, which in turn depends on the fused loop structure. The dynamic memory allocation cost model, therefore, must maintain a list of tree fragments for computing the evaluation order and then compute the overall cost from the cost of the fragments. The fragments in the cost model correspond to the code fragments in the code generation algorithm in Section 2.4.3, except that the cost model only needs to compute the memory usage for each fragment instead of the code.

We use the following data structure for representing a solution in our dynamic memory allocation cost model for an expression tree t:

$$\mathrm{S}(f, (hi, lo, hiC, loC), frags, t)$$

where f is the maximal fusion between the root node of t and its parent, *hi* is the maximum memory usage during the evaluation of the root fragment of t under fusion f, and *lo* = `fusedSize(`t`,` f`)` is the memory needed for storing t under fusion f. To allow recomputing *hi* and *lo* when the fusion changes in `extendSoln` or `reduceSoln`, we also store *hiC* and *loC*, where *hiC* is the maximum memory usage during the evaluation of the subtrees of t for those parts of the subtrees that are fused into the root fragment and *loC* is the memory needed for storing the results of those subtrees. Finally, *frags* is a list of $(\mathit{fset}, \mathit{fhi}, \mathit{flo}, t')$ quadruples representing those fragments that have not yet been fused with the root fragment.

For each fragment, *fset* is the set of fused indices and *fhi* and *flo* are the maximum memory usage for computing the fragment and the memory needed for storing the result, respectively. The tree node t' is the parent node of a fragment. The cost model also generates fragments for representing the initialization of a temporary variable; for those fragments t' is null. The fragments in *frags* are sorted in the order in which they will be fused with the root fragment, which is in non-increasing number of fused indices and for fragments with equal fused indices in non-decreasing *hi* − *lo* order. The set of fused indices of any fragment in the list is equal to or a superset of all following fragments. The set of fused indices of any fragment is a proper subset of $\mathit{set}(f)$. The fragments are in the reverse of the order in which the corresponding code fragments will appear in the final code.

Figure 4.1 shows functions operating on lists of fragments. The function `mergeFrags` merges two sorted lists of fragments in non-increasing number of fused indices and for equal fused indices in non-decreasing *hi* − *lo* order. The function `combineFrags` merges any neighboring fragments with the same fused indices into a single fragment. It is called in `extendSoln` after the actual fusion for the root node of a subtree has been chosen and ensures that any parts of the subtree with equal fusion are evaluated contiguously.

The functions `leqFrags` and `Low` are used for comparing lists of fragments when pruning solution sets. The function `Low` adds up the *lo* fields of all fragments. The call `leqFrags(`*frags*`,` *frags*′`)` returns true if and only if for any set of fused indices (i.e., at any point in the lists of fragments), both the *hi* and the `Low` at this point in *frags* are larger than or equal to those in *frags*′. Suppose *frags* and *frags*′ are the lists of fragments of two otherwise identical solutions for a subtree. If `leqFrags(`*frags*`,` *frags*′`)` returns true, then inserting any fragment from another subtree in *frags* can never result in a better final solution than inserting the same fragment at the corresponding point in *frags*′.

```
mergeFrags(frags, frags'):
  if frags = nil then return frags'
  else if frags' = nil then return frags
  else
    (fi,hi,lo,t) = head(frags)
    (fi',hi',lo',t') = head(frags')
    if subset(fi', fi) or (fi = fi' and hi' - lo' >= hi - lo) then
      return (fi,hi,lo,t) :: mergeFrags(tail(frags), frags')
    else
      return (fi',hi',lo',t') :: mergeFrags(frags, tail(frags'))

combineFrags(frags):
  if frags = nil or tail(frags) = nil then return frags
  (fi,hi,lo,t) = head(frags)
  (fi',hi',lo',t') = head(tail(frags))
  if fi = fi' then
    return combineFrags((fi, max(hi',lo'+hi), lo+lo', t) :: tail(tail(frags)))
  else
    return (fi,hi,lo,t) :: combineFrags(tail(frags))

leqFrags(frags, frags'):
  if frags' = nil then return true
  else if frags = nil then return false
  else
    (fi,hi,lo,t) = head(frags)
    (fi',hi',lo',t') = head(frags')
    if fi = fi' and (hi < hi' or Low(tail(frags)) < Low(tail(frags'))) then
      return false
    else if fi = fi' then
      return leqFrags(tail(frags), tail(frags'))
    else if subset(fi', fi) then
      return leqFrags(tail(frags), frags')
    else
      return leqFrags(frags, tail(frags'))

Low(frags):
  if frags = nil then return 0
  else
    (fi,hi,lo,t) = head(frags)
    return lo + Low(tail(frags))
```

FIGURE 4.1. Operations on fragments for the dynamic memory allocation cost model.

Figure 4.2 shows the dynamic memory allocation cost model. Similar as in the static memory allocation cost model, `makeLeafSoln` creates a solution with $hi = lo = 0$ for `Array` and `Const` nodes and a solution with $hi = lo = 1$ for `Fun` nodes. In either case, the list of fragments is empty.

If the subtree of a summation node is fully fused with its parent, `makeUnarySoln` creates a solution in which $lo = 1$ (assuming full fusion of the summation node with its parent) and in which the *hi* is the maximum of the *hi* of the subtree and the *lo* of the subtree plus the *lo* of the summation node. If the subtree is not fully fused with the summation node, the root fragment of the subtree is added to the list of fragments and a new root fragment consisting only of the summation node is created with cost $hi = lo = 1$.

Similar as for unary nodes, `makeBinarySoln` tests whether the subtrees are fused with the parent. If both subtrees are fused, the cost is computed for an evaluation in non-increasing $hi - lo$ order. If a subtree is not fused, the root fragment of that subtree is added to the list of fragments. Only the cost of fused subtrees is incorporated in the cost of the root fragment. Finally, the fragments from the two subtrees are merged.

The functions `extendSoln` and `reduceSoln` result in a new fusion f' for the root fragment in the same way as in the static memory allocation cost model. The *lo* of the root fragment is recomputed using f'. Since f' may fuse only a subset of the indices fused by the original maximal fusion f, all fragments whose set of fused indices is equal to or a superset of $set(f)$ now need to be fused or concatenated with the root fragment. Furthermore, the initialization of the root node t must be moved before any loops that will be fused with fragments but not with the parent. To calculate the cost for initializations correctly, we insert an initialization fragment into the list of fragments. The function `useFrags` then recursively traverses the list of fragments and updates the cost of the root fragment to reflect the fusion of fragments with more fused indices than f' and the concatenation of fragments with the same set of fused indices. The cost computation in `useFrags` depends on whether the fragment is the initialization fragment, the fragment for a subtree of t, or the fragment for a descendent of t. Finally, if `useFrags` was called by `extendSoln`, any remaining fragments with equal sets of fused indices can now be combined since they will not be fused with t anymore.

Finally, the function `inferior` compares two solutions for the purpose of pruning. A solution s is inferior to s' if s has equal or larger *hi* and *lo* than s', if the `Low` of its fragment list is larger or equal than that of s', if the nesting of s is more or equally constraining that that of s', and if `leqFrags` for the two fragment lists returns true.

```
makeLeafSoln(t):
  if t is an Array node or a Const node then
    return { S(any, (0,0,0,0), [], t) }
  else if t is a Fun node then
    return { S([fusible(t)], (1,1,0,0), [], t) }

makeUnarySoln(S(f, (hi,lo,hiC,loC), frags, l), h, t):
  if set(f) = indices(t) then
    hi' = max(hi, lo + 1)
    return { S(h, (hi',1,hi,lo), frags, t) }
  else                                  // subtree is not fused with parent
    return { S(h, (1,1,0,0), (set(f),hi,lo,t) :: frags, t) }

makeBinarySoln(S(f, (hi,lo,hiC,loC), frags, l), h,
               S(f', (hi',lo',hiC',loC'), frags', r), h', t):
  f'' = mergeNesting(h, h')
  if set(f) = indices(t) and set(f') = indices(t) then
    if hi - lo >= hi' - lo' then        // both subtrees are fused with parent
      hiC'' = max(hi, lo + hi')         // evaluate left then right subtree
    else
      hiC'' = max(hi', lo' + hi)        // evaluate right then left subtree
    hi'' = max(hiC'', loC + loC' + 1)
    return { S(f'', (hi'',1,hiC'',loC+loC'), mergeFrags(frags, frags'), t) }
  else if set(f) = indices(t) then      // only left subtree is fused
    hi'' = max(hi, lo + 1)
    return { S(f'', (hi'',1,hi,lo),
               mergeFrags(frags, (set(f'),hi',lo',t)::frags'), t) }
  else if set(f') = indices(t) then     // only right subtree is fused
    hi'' = max(hi', lo' + 1)
    return { S(f'', (hi'',1,hi',lo'),
               mergeFrags((set(f),hi,lo,t)::frags, frags'), t) }
  else                                  // neither subtree is fused with parent
    return { S(f'', (1,1,0,0),
               mergeFrags((set(f),hi,lo,t)::frags, (set(f'),hi',lo',t)::frags'), t) }

useFrags(S(f, (hi,lo,hiC,loC), frags, t), e):
  if frags = nil then return S(f, (hi,lo,hiC,loC), [], t)
  (fi,hi',lo',t') = head(frags)
  if t' = null then             // head(frags) is initialization of t
    return useFrags(S(f, (lo'+hi,lo',hiC,loC), tail(frags), t), e)
  else if fi = set(f) or subset(set(f), fi) then        // use head(frags)
    loC' = loC + lo'
    hiC' = max(hi', lo' + hiC)
    if t' = t and not(fi = set(f)) then // head(frags) is a child of t
      hi'' = max(hiC', loC' + lo)
      return useFrags(S(f, (hi'',lo,hiC',loC'), tail(frags), t), e)
    else                                // head(frags) is a decendent of t
      hi'' = max(hi', lo' + hi)
      return useFrags(S(f, (hi'',lo,hiC',loC'), tail(frags), t), e)
  else return S(f, (hi,lo,hiC,loC), if e then combineFrags(frags) else frags, t)

extendSoln(S(f, (hi,lo,hiC,loC), frags, t), f'):
  lo' = fusedSize(t, f')
  if set(f') = indices(t) then return S(f', (hi,lo,hiC,loC), frags, t)
  else return { useFrags(S(f', (hiC,0,hiC,loC),
                           mergeFrags([(set(f'),lo',lo',null)], frags), t), true) }

reduceSoln(S(f, (hi,lo,hiC,loC), frags, t), i):
  f' = maxPrefix(f, i)
  lo' = fusedSize(t, f')
  return { useFrags(S(f', (hiC,0,hiC,loC),
                      mergeFrags([(set(f'),lo',lo',null)], frags), t), false) }

inferior(S(f, (hi,lo,hiC,loC), frags, t), h,
         S(f', (hi',lo',hiC',loC'), frags', t'), h'):
  return hi >= hi' and lo >= lo' and Low(frags) >= Low(frags') and
         leq(h, h') and leqFrags(frags, frags')
```

FIGURE 4.2. The cost model for dynamic memory allocation.

Code generation is similar as for the static memory allocation cost model. When constructing the fusion tree representation from the solution tree, both the *hi* and the *lo* for each node are recorded. The code generation algorithm then recomputes the $hi - lo$ ordering of any fragments with equal fusion and generates the code accordingly. In addition to generating the initialization fragments for summation nodes, the code generation algorithm now also must generate memory allocation statements for all non-scalar temporaries. For generating deallocation statements, a list of deallocation fragments is maintained. When generating a loop, any allocation, initialization, and computation fragments with the same set of fused indices are added before the loop, any deallocation fragments with the same set of fused indices are added after the loop.

If we would base our cost model on the memory allocation algorithm by Lam et al. [LRB+11] and allow non-contiguous evaluation of subtrees, code generation would become more complex. Instead of simply ordering fragments in non-increasing $hi - lo$ order, the code generation algorithm would need to recompute fragment boundaries. Alternatively, the cost model could record the evaluation order as a list of tree nodes for each fragment, which then would be used to guide code generation.

Table 4.1 shows the steps of the fusion algorithm with the dynamic memory allocation cost model for computing the optimal solution for the example from Figure 2.1. Solutions that do not lead to the optimal solution are omitted. The resulting loop structure is the same as with the static memory allocation cost model, as shown in Figure 2.7. Because of the constraints placed on the evaluation order by loop fusion, dynamic memory allocation did not result in a noticeable reduction in memory usage. The maximum memory usage of 158 that is computed is only slightly less than the memory usage of 160 for the static memory allocation cost model. The maximum memory usage occurs during the evaluation of `f2`. At this point, `fA` has already been deallocated and `f4` has not yet been allocated. Since in code generation no allocation statements would be generated for scalars and since consecutive assignments to scalars could be combined into a single statement, the actual memory usage for this example is identical for static and dynamic allocation.

TABLE 4.1. Algorithm trace for the example from Fig. 2.1 with a dynamic memory allocation cost model.

Node	max. fusion	ext./red.	cost of root	fragments
A	$\langle ij \rangle$		(1,1,0,0)	[]
$f1$	$\langle ij \rangle$		(2,1,1,1)	[]
	$\langle j \rangle$	R $\{j\}$	(2,1,1,1)	[]
	$\langle \rangle$	E $\langle jk \rangle$	(101,100,1,1)	[]
B	$\langle jkl \rangle$		(1,1,0,0)	[]
C	$\langle kl \rangle$		(1,1,0,0)	[]
	$\langle k \rangle$	E $\langle k, jl \rangle$	(15,15,0,0)	[]
$f2$	$\langle k, jl \rangle$		(2,1,1,1)	[(k,15,15,f2)]
$f3$	$\langle k, jl \rangle$		(2,1,2,1)	[(k,15,15,f2)]
	$\langle k, j \rangle$	R $\{jk\}$	(3,1,2,1)	[(k,15,15,f2)]
	$\langle k, j \rangle$	E $\langle k, j \rangle$	(3,1,2,1)	[(k,15,15,f2)]
$f4$	$\langle k, j \rangle$		(3,1,3,1)	[(k,15,15,f2),(,101,100,f4)]
$f5$	$\langle k, j \rangle$		(3,1,3,1)	[(k,15,15,f2),(,101,100,f4)]
	$\langle k \rangle$	R $\{k\}$	(19,1,18,16)	[(,101,100,f4)]
	$\langle \rangle$	R $\{\}$	(158,40,118,116)	[]

A disadvantage of the dynamic memory allocation cost model is that it is more expensive. Maintaining its solution data structure is more time-consuming than maintaining a single memory count, and the more complex cost model results in fewer pruning opportunities. While with the static memory allocation cost model a total of 40 solutions are generated for our example, as shown in Table 2.1, with this cost model there are a total of 51 solutions.

Chapter 5
Loop Fusion Optimization for Handwritten Code

The search-based loop fusion optimization has been successful in the Tensor Contraction Engine for minimizing storage or, together with a search-based tiling optimization, for automatically generating efficient out-of-core code from tensor contraction expressions. This chapter presents a generalization of the loop fusion algorithm to handwritten code as a first step towards generating out-of-core code from handwritten in-core code. Handwritten tensor contraction code in a general purpose language could be interspersed with non-tensor contraction code (e.g., fast Fourier transform code). Expressing such a code mixture in a domain-specific language (DSL) could be very complex, but is easier to handle as handwritten code. Polyhedral compilation currently is state-of-the-art for analyzing handwritten general purpose loop code with complex dependencies and producing optimized CPU and accelerator code. We present a discussion comparing our approach with polyhedral compilation in Section 5.3. We demonstrate in our measurements in Section 8.2 that the current polyhedral compilers are unable to deliver competitive performance for tensor contractions.

The loop fusion algorithm presented in Chapter 2 optimizes dense array computations by fusing producer and consumer loops to minimize the storage requirements for intermediate arrays. It undoes any fusion provided by the programmer and, using a cost model for memory minimization, searches all loop structures for the memory-minimal code. Then the fused loops are expanded to tile size and these tiles are used as units of data transfers between different levels in the memory hierarchy. We demonstrate the algorithm using an example. We have implemented it as a source-to-source translation using the ROSE compiler framework. We also demonstrate the effectiveness of this approach for multi-core CPUs and GPGPUs in Section 8.2.

The combination of fusion and tiling optimizations has proven successful in the TCE for minimizing disk to memory traffic for dense tensor computations. While other optimizations are specific to tensor contraction expressions, these two optimizations can be useful for optimizing handwritten dense array computations. The fusion and tiling optimizations can be used for different purposes. For example, the fusion optimization by itself can be used for minimizing memory requirements for intermediate results [LCBS99, Lam99]. Together with the tiling optimization and with different cost models, these optimizations can be used for minimizing

disk to memory traffic [BKC+03, KKB+03] or for space-time tradeoffs [CBL+02b]. Since the problem of making the fusion algorithm work on an abstract syntax tree is independent of the cost model, we limit the discussion to the use of the fusion algorithm as a stand-alone optimization for memory minimization.

We now focus on the non-trivial problem of finding a loop fusion configuration for well behaved handwritten code that minimizes memory usage without increasing the operation count. The loop fusion algorithm is a bottom-up dynamic programming algorithm that operates on expression trees. An expression tree has the advantage that the consumer of an intermediate array is the parent node of the producer of that array. Furthermore, all loops are implicit, since the interior nodes in an expression tree represent array operations. In order to handle handwritten code, we generalize the loop fusion algorithm to abstract syntax trees with explicit loops. For this, it is necessary to reconstruct the producer-consumer relationship and recognize any language constructs or code structures that cannot be optimized with the fusion algorithm. The goal is to use search with memory cost and disk I/O cost for handwritten tensor contraction code. The programmer can use arrays of arbitrary size and need not worry about disk I/O.

For generalizing the fusion algorithm to an abstract syntax tree, we face the following challenges:

- The producer and consumer of an intermediate array may be in different loop nests. It is, therefore, necessary to reconstruct the producer-consumer relationship using reaching definitions analysis.

- A producer of an intermediate might have multiple consumers. Conversely, a consumer might have multiple reaching definitions.

- Handwritten code can be unfused, partially fused, or suboptimally fused. When the producer and consumer of an intermediate are fused, the intermediate has fewer dimensions than would be necessary if the producer and consumer were not fused. In case this is not an optimal fusion, we need to reconstruct the indices that would be necessary if there were no fusion.

- When working with handwritten code, any language constructs that cannot be optimized, such as while loops, if statements, scalar code, or function calls need to be ignored, while for loops involving array computation must be optimized.

- Any function calls within an expression might have side effects. The loops around these function calls, therefore, cannot be reordered, which constrains the fusion possibilities.

- Complex array subscript expressions or loop bounds create dependencies among array elements that can prevent certain loop interchanges or loop fusions.

- Finally, sparse arrays, variable loop ranges, etc., can make the computation of the memory requirements needed for the cost model impossible at compile time.

Additionally, for arbitrary code in the presence of function calls with side effects in array subscript expressions, it can be impossible to calculate the memory requirements or to determine the validity of fusions at compile time. However, it is possible to identify the parts of the code that can benefit from fusion optimization, optimize those, and leave the rest untouched.

As a first step towards the goal of turning in-core code into out-of-core code for a wide range of handwritten codes, we make the following simplifying assumptions:

- Array subscript expressions are simple loop variables.

- Arrays are dense and loop bounds are constant.

While these are drastic simplifications, the resulting code is still general enough to cover tensor contractions and some other linear algebra codes.

5.1 Algorithm

Our fusion algorithm searches through possible loop fusion configurations between the producers and the consumers of intermediates. When operating on an expression tree, the producer and the consumer of an intermediate are conveniently in a parent-child relationship. In handwritten code, they may be in different loop nests. Furthermore, for handwritten code it is necessary to identify the code regions on which a loop fusion optimization can safely be performed.

Our memory minimization algorithm for handwritten code runs the loop fusion algorithm on the dependency graph after a series of traversals of the abstract syntax tree that prepare the abstract syntax tree for the fusion algorithm and identify where loop fusion optimizations can safely be performed. After completion of the loop fusion algorithm, we generate a new abstract syntax tree that represents the memory-minimal fused code.

Our algorithm consists of the following steps:

Canonicalization moves code containing side effects out of expressions.

Region identification identifies code regions in which loop fusion optimizations are safe.

Subscript inference computes the indices needed for intermediates in unfused code.

Reaching definitions analysis identifies producers and consumers of intermediate arrays.

Loop fusion optimization identifies the memory-minimal loop fusion configuration.

Code generation constructs an abstract syntax tree for the optimal loop fusion configuration.

5.1.1 Canonicalization

Since the loop fusion algorithm can reorder loops, any code that may depend on the loop order must be flagged as such or moved out of larger expressions. For individual subexpressions, it is sufficient to store the required fusion in the corresponding parse tree node. Suppose we have the source code

```
for i
  for j
    x[i,j] = ... + f(i,j) + ...;
```

and that the function f has side effects. If the function call does not have a side effect on other subexpressions of this expression, the function call node could simply be marked with the fusion $\langle i, j \rangle$, which indicates that for the loops surrounding the function call itself i must be the outer loop and j must be the inner loop. The loop fusion algorithm would still be able to reorder the loops surrounding the rest of the expression. E.g., it might decide to fuse the assignment to x with its consumer by moving f into a temporary:

```
for i
  for j
    t[i,j] = f(i,j);
for j
  for i
    x = ... + t[i,j] + ...;
    y[i,j] = ... * x;
```

If two or more subexpressions might have side effects on one another, they are moved out of the expression and assigned to temporaries. The entire sequence of assignments to temporaries is then marked with the appropriate fusion.

5.1.2 Region Identification

This traversal identifies constraints on fusing loops by marking code regions on which to run the loop fusion algorithm. If fusion across two code fragments is valid, both are assigned the same region mark. If one of the code fragments is inside a loop or an *if-then-else* statement but not the other, they cannot be in the same region. Similarly, two code fragments cannot be in the same region if they are separated by a language construct such as a *while* loop, a *function call*, or an *if-then-else* statement. Fusion across any two code regions containing these language constructs would reorder the computation and, therefore, change the semantics of the code.

5.1.3 Subscript Inference

The indices at a node are used by the fusion algorithm to determine the constraints for fusing loops. For any node *v* inside a loop, *dimensions* refers to the loop indices around it. If *v* is an array reference, *dimensions* refers to loop indices that are used to index the array dimensions. The procedure for computing indices is shown in Figure 5.1.

```
computeIndices(T):
  for each node v in some top down traversal of T do
    if v is an Assignment node then
      indices(v) = union(dimensions(child1(v)), dimensions(child2(v)))

      if dimensionCount(child1(v)) < dimensionCount(child2(v)) then
        summationIndices(v) = difference(dimensions(child2(v)), dimensions(child1(v)))

    if v is a Binary operator node then
      indices(v) = union(dimensions(child1(v)), dimensions(child2(v)))

    if v is a Leaf node then
      if v is a Constant then
        indices(v) = < * >
      if v is any variable other than a loop variable then
        if parent(v) is a Summation operator node and
           v is the left child of parent(v) then
             indices(v) = < * >

      else if !exists(reachingDefinition(v)) then
             indices(v) = dimensions(v)
      else   indices(v) = union(indices(reachingDefinition(v)),
                                loopIndicesFusedBetween(v, reachingDefinition(v)))
```

FIGURE 5.1. Procedure to compute indices for fusion.

5.1.4 Reaching Definitions Analysis

In order to generalize the loop fusion algorithm to handwritten code, it is necessary to reconstruct the producer-consumer relationship and recognize any language constructs or code structures that cannot be optimized with the fusion algorithm. We perform the reaching definition analysis to reconstruct the expression

tree as a dependency graph containing the producer-consumer relationships. This analysis is performed at the level of abstract syntax trees as opposed to the traditional analysis on intermediate code. We know that for any variable used in the code, its reaching definition would be the most recent definition of that variable. We only record a reaching definition if both the variable used and its reaching definition have the same mark assigned during the region identification traversal. For example, if the definition of the variable is inside a *while* loop and its use is outside the loop, then we do not create a reaching definition edge. In case of such loops, we may have a reaching definition to a statement in which a variable is both used and defined. We do not record the reaching definition for a statement inside a loop in which a variable is both used and defined. This is because during fusion, we identify the root of the dependency (reaching definitions) graph as a starting point for running the fusion algorithm. A node is identified as the root if it has no incoming reaching definitions. Since array reference indexes are simple loop indices and not affine, there exist no dependencies from one loop iteration to the next. Multiple root nodes can exist in the dependency graph we construct for a given tensor contraction code. A usage node can have multiple reaching definitions and a definition node can have multiple incoming reaching definitions. In this case, we leave the code regions as is and do not attempt to consider more complex fusion possibilities.

5.1.5 Loop Fusion for Handwritten Code

The fusion algorithm is run on the dependency graph with sufficient information at each node collected as the result of all the previous traversals, that is an expression tree consisting of reaching definitions edges and the expressions and assignments from the abstract syntax tree. We use a dynamic programming algorithm for computing the memory optimal loop fusion configuration. The algorithm traverses the dependency graph bottom-up and generates a set of possible loop structures for each tree node. Each solution is represented as a pair containing a fusion and memory cost. We restrict ourselves to a static memory allocation model and hence compute the memory usage by simply summing up the sizes of all arrays in the memory-optimal tree achieved as the result of fusion.

- *FunctionCall Node:*

  For the fusion algorithm to distinguish disk arrays from memory-resident arrays, we represent disk arrays as function calls. In such a case, a single solution is generated which indicates that this node can

be fully fused with its parent in any order, provided they are both present in code regions having the same mark.

- *A leaf node:*

  A single solution is generated to indicate that the node can be accessed repeatedly at zero cost, and also if the leaf node has the special index set with the single *any* entry. This means that not only the indices of the leaf node but also any index of the parent can be fused between the leaf node and the parent. For leaf nodes which do not have the special index set representing their indices, this node can only be fused with an index at the parent found in the *indices* set of the leaf node.

- *Binary Operator Node:*

  Let v be a binary operator node. The solutions for the left and right subtrees of v must be combined to form a solution for v. This is achieved using `makeBinarySolnSet` shown in Figure 2.5. Also for binary operations, the parent may have more dimensions than the child and hence, it is necessary to enumerate all possible ways in which the loop structures for the subtrees can be fused with the parent, such that only the compatible solutions for the two subtrees can be combined to form a solution for v. It is possible that a binary operator is being used for the purpose of summation or product of a variable. To serve this purpose, there exists a loop index (indices) which is already in the solution set of v. This loop index is not required by the parent of v. Hence, we need to perform the *Reduction* step, which is the process of removing the extraneous index from the solution set of v.

- *Assignment Node:*

  For an assignment node v, we combine the solutions for the two subtrees using `makeBinarySolnSet`. However, if v has more indices from the right subtree compared to the left subtree, it means that there is a binary operator node in the subtree of v used for the purpose of summation or repetitive multiplication. In such a case, we also perform the *Reduction* step that was delayed at the binary operator node, at v. This removes the extraneous indices from the solution set of v and adjusts the memory cost accordingly.

Finally, we obtain a set of solutions for the root of the dependency graph for a tensor contraction expression. Multiple root nodes can exist in a dependency graph. The solutions obtained have different maximal fusions and different costs. At the root node, we remove all indices from the maximal fusions for all solutions thereby increasing the memory costs for the various solutions. These sets of solutions are further pruned to find the

```
1    initialize f1
2    for i
3      for j
4        f1[j] = f1[j] + A(i,j)
5    initialize f3
6    for k
7      for l
8        fC = C(k,l)
9        for j
10         f3[j][k] = f3[j][k] + B[j][k][l] * fC
11   initialize f5
12   for j
13     for k
14       f5[k] = f5[k] + f1[j] * f3[j][k]
```

FIGURE 5.2. Partially fused input code

optimal solution. To generate code from the optimal solution, we use the same mechanism described in Section 2.4.3.

5.2 Example

To illustrate how the algorithm works, consider the partially fused input code in Fig. 5.2. Let $N_i = 500$, $N_j = 100$, $N_k = 40$, and $N_l = 15$ be the loop index ranges. The code after canonicalization is shown in Fig. 5.3. The function call `A(i,j)` is moved outside the expression and evaluated for side-effects. If the array `f1` is not in the scope of `A(i,j)`, it cannot be modified. In such a case, `f1` and `A(i,j)` are said to commute and hence `f1` is not moved outside the expression. Otherwise, `f1` needs to be moved out and assigned to a temporary.

In this example, the region identification traversal results in all code regions having the same mark because we do not have code where either the definition or the usage is inside a conflicting language construct.

The outcome of reaching definitions analysis for lines 7 through 11 is shown in Fig. 5.4. The use of `f3` in the left hand side of assignment 2 does not have a reaching definition edge to assignment 2, where `f3` is also defined. This is because we identify the root of the dependency (reaching definitions) graph as the assignment node that has no incoming reaching definition. There exist no dependencies between the array reference `f3` of one loop iteration to the next.

To demonstrate the computation of indices, consider the function call `fC` in line 11. The indices at `fC` are the indices at its reaching definition in line 9 and the indices of the loops already fused between the definition and use of `fC`, which are {`k,l`}. This computation allows to unfuse the existing fusion, and finds out that

```
1    initialize f1
2    for i
3      for j
4        fA = A(i,j)
5        f1[j] = f1[j] + fA
6    initialize f3
7    for k
8      for l
9        fC = C(k,l)
10       for j
11         f3[j][k] = f3[j][k] + B[j][k][l] * fC
12   initialize f5
13   for j
14     for k
15       f5[k] = f5[k] + f1[j] * f3[j][k]
```

FIGURE 5.3. Canonicalized code.

`fC` refers to a two dimensional array which was accessed repeatedly in the {`l`} loop. This information is used by the fusion algorithm to explore all possible fusions between `fC` and its parent node.

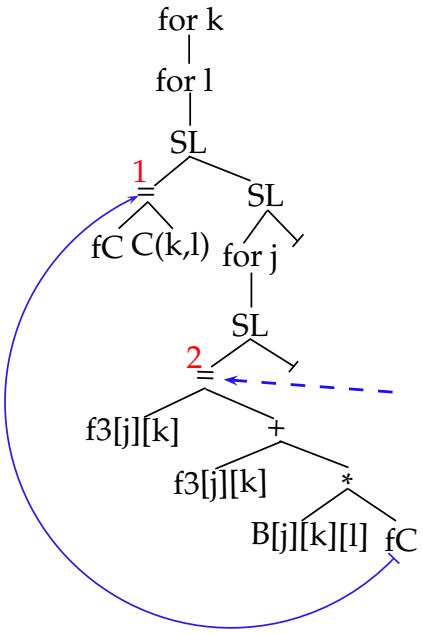


FIGURE 5.4. Absyn Tree

To illustrate the fusion algorithm on our example, consider the subtree rooted at the parent of `f3`. The fusion at `B` is the full fusion $\langle jkl \rangle$ because other fusion configurations are more constraining and result in a higher memory cost. At `fC`, the full fusion $\langle kl \rangle$ leads to other fusion possibilities at the parent due to the {l} loop. The fusion $\langle kl, j \rangle$ is optimal for the subtree rooted at the parent of `B` and `fC`, but later turns out to be non-optimal for the entire tree. If `B` was referred using a function call like `C`, the loop order $\langle k, l, j \rangle$

should be preserved at `fB`, otherwise rearranging loops around a function call might result in side effects. The fusion possibilities would be very limited as a result of preserving the loop order. The optimally fused code in Fig. 5.6 is constructed from the optimal fusion graph in Fig. 5.5.

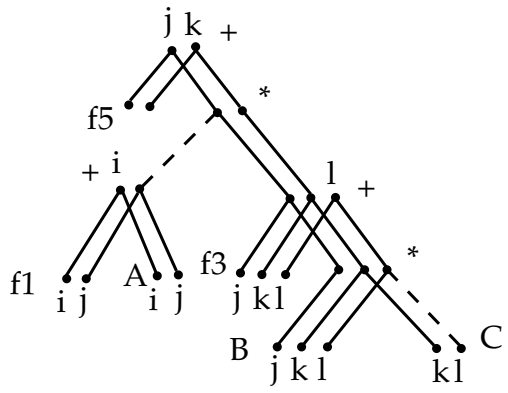


FIGURE 5.5. Optimal Fusion Graph

```
1    initialize f1
2    for i
3      for j
4        fA = A(i,j)
5        f1[j] = f1[j] + fA
6    initialize f5
7    for k
8      for l
9        fC[l] = C(k,l)
10     for j
11       initialize f3
12       for l
13         f2 = B[j][k][l] * fC[l]
14         f3 = f3 + f2
15       f4 = f1[j] * f3
16       f5[k] = f5[k] + f4
```

FIGURE 5.6. Optimally Fused code.

5.3 Comparison with the Polyhedral Model

Polyhedral compilation currently is state-of-the-art for analyzing handwritten general purpose loop code with complex dependencies and producing optimized CPU and accelerator code. Pluto [BHRS08a] is an automatic parallelizer and locality optimizer based on the polyhedral model, which is a powerful mathematical framework that provides precise dependency analysis and the mechanism to implement complex transformations of

codes with nested loops and also the ability to reason about execution instances of statements. Pluto automatically generates OpenMP [DM98] C code for multi-cores and optimizes for coarse-grained parallelism and data locality. However, for tensor contraction computations, Pluto does not give an acceptable performance. It does not have support for performing array contraction, which is the key for reducing memory requirements for very large tensor contraction sequences requiring significant memory.

Our measurements with Pluto by providing the TCE equations as input and using the various fusion optimizations and tiling in Pluto show that the performance of fused code produced by the TCE is identical to the unfused code when both the unfused and fused codes are provided as input to Pluto. However, when Pluto performs tiling, it does not try to increase the fused dimensions to tile sizes because it does not have the fusion information from the TCE fused code that is provided as input to Pluto. With larger equations, just applying loop fusion and tiling to improve locality does not necessarily result in the best performance, because the arrays are still not contracted (which leads to no reduction in memory sizes) and we need to resort to disk I/O, which is expensive. Also, Pluto does not automatically generate out-of-core code when the computation does not fit in memory. Using optimized BLAS routines directly for these code structures would not necessarily give the best possible performance, mainly due to the disk access penalty and bad layout of the multidimensional intermediate arrays as a result of loop fusion. Performing these optimizations (loop fusion and tiling) first and then using BLAS routines when generating code results in much better performance. We present the detailed measurements in Section 8.2.

Chapter 6
Loop Fusion Optimization for GPGPUs

The loop fusion algorithm presented in Chapter 2 is a bottom-up dynamic programming algorithm that operates on expression trees. An expression tree has the advantage that the consumer of an intermediate array is the parent node of the producer of that array. Furthermore, all loops are implicit, since the interior nodes in an expression tree represent array operations.

The disadvantage (later discussed in detail in Section 8.2) of the loop fusion algorithm, however, is that it can result in poor temporal and spatial locality. By contrast, if the tensors are small enough, each individual contraction could be implemented as a combination of index permutation and matrix multiplication library calls, where the index permutations transform the layout of the multi-dimensional tensors such that their storage can be interpreted as two-dimensional matrices. The use of such library calls results in significantly improved temporal and spatial locality. In order to improve locality, tensor dimensions that were eliminated through fusion, must be expanded again to some tile size, such that the index permutation and matrix multiplication calls can be performed on individual tiles.

Fusion plays an important role when trying to orchestrate the data between disk, CPU memory and GPU memory. Due to the reduction in size of intermediates, the volume of data that needs to be orchestrated across various memory levels is reduced. A cost model that searches for loop fusion configurations that minimize data movement cost across disk, CPU memory, and GPU memory is needed. Data layout optimizations could be performed using the CPU cores, and the remaining computation can be performed on the GPU using CUDA BLAS calls.

Similar as the data transfer between memory and disk in out-of-core computations, the transfer between GPU memory and CPU memory is very time-consuming and needs to be avoided where possible. For larger tensors, a contraction on the GPU is so much faster than on the CPU, that even for individual contractions it is beneficial to move the data back and forth. When adding two tensors, however, it is cheaper to add them on the CPU than to transfer them to the GPU, perform the addition there and transfer the result back to the CPU. The advantage of loop fusion plus tiling for tensor contractions compared to tiling alone is that the amount of transfer between GPU and CPU memory can be reduced because of the size reduction of intermediates.

However, because of the competitive cost of addition on the CPU, there is no benefit to using fusion to allow addition to be performed on the GPU.

Based on these constraints, we need to generate the following general code structures for increasing sizes of the tensors:

1. For very small tensors, perform the entire computation on the CPU.

2. Once the tensors are large enough to benefit from the parallelism on the GPU, perform the entire computation on the GPU.

3. If the computation does not fit on the GPU anymore, perform contractions on the GPU without fusion and perform additions on the CPU.

4. If individual terms of contractions are too large for the GPU, use fusion plus tiling to reduce the storage requirements on the GPU. This may result in additions to fit on the GPU again, too.

5. As the tensor sizes increase, use fusion plus tiling to keep contractions on the GPU and perform additions on the CPU.

6. For very large tensors, intermediate results of contractions may have to be transferred to CPU memory temporarily.

6.1 Algorithms

Similar to loop fusion for handwritten code, we use a compilation framework consists of three optimizations for generating GPU code: loop fusion, tiling, and layout optimizations. The loop fusion algorithm is a dynamic programming algorithm operating on expression trees. If it is used with a memory-minimization cost model, it produces a *fusion tree* representation of a single, memory-minimal fused loop structure. With a cost model containing more than one cost component, such as memory usage and data transfer cost, it produces a set of solutions with different trade-offs. For each of these solutions, the tiling algorithm, using a fine-grained cost model, selects the tile sizes that minimize the data transfer cost while staying below the memory limit. The best solution overall is then selected and further optimized using the layout optimization algorithm. We have previously demonstrated the combination of fusion and tiling optimizations for space-time trade-offs [CBL+02b] and for producing out-of-core code for individual contraction terms

[BKC+03, KKB+03]. Our layout optimization algorithm [LGK+12] has been developed for untiled code, but it could also be generalized for tiled code.

6.1.1 Fusion Algorithm for GPGPUs

The fusion algorithm visits each node in an expression tree in a bottom-up traversal. For each node, it computes a set of solutions, each of which consists of compatible solutions for any subtrees, the maximal fusion with the parent and one or more additional cost components. A solution that is inferior to another solution in all cost components is pruned out. For GPU code generation, we propose the following cost vector for each tree node:

(fusion, mem-cost, transfer-cost, proc, target)

where

- **fusion** is the maximal fusion between the node and its parent,
- **mem-cost** is the memory usage for computing the subtree assuming static allocation of all tensors without tiling,
- **transfer-cost** is the volume of data to be transferred without tiling,
- **proc** is the processor performing the operation (CPU or GPU), and
- **target** is final location of the generated intermediate (CPU memory or GPU memory)

The *mem-cost* is the same as in the original loop fusion algorithm in Chapter 2. Here, it represents the memory usage for computing the subtree on the GPU (instead of the CPU). Similar to the disk-memory data transfer cost used in data locality optimization [BKC+03, KKB+03] for minimizing disk I/O, *transfer-cost* is used to measure the volume of data to be transferred between CPU and GPU memories without tiling. Both the *mem-cost* and *transfer-cost* are only cost approximations since they are based on the assumptions that all tensors are allocated statically and that loops are not tiled. However, these approximations are good enough for deciding which solutions to prune out. The resulting set of solutions will consist of a variety of different fused loop structures, that can be further optimized using the tiling algorithm with a more precise cost model.

The *proc* component indicates whether a binary tensor addition is performed by the CPU or the GPU. If a child of the node is a binary addition as well, then both the child and the parent operations are performed

by the same processor. For the purpose of the loop fusion algorithm, contractions are always performed on the GPU. The optimization of performing very small computations on the CPU can be done by the tiling algorithm.

The *target* component indicates whether the result of a contraction should be transferred to CPU memory. In that case, we always transfer the entire tensor to CPU memory. We do not consider partial transfers using fusion, since that would result in an increase of the transfer cost. The result of a binary addition is always assumed to stay where the operation was performed. The optimization of performing the last binary addition in a chain where the result is needed can be done by the tiling algorithm.

To ensure that all possible code structures can be generated, the loop fusion algorithm needs to create different solutions for performing an operation on the GPU and the CPU and similarly for where the result should be transferred. The tiling algorithm will then select the optimal code using a more precise cost model. For other cost models, the fusion algorithm enumerates all possible fusion choices. For the purpose of GPU code generation, however, we do not fuse a node with its parent if the parent is a binary addition operation, since this would result in a larger number of transfer operations and since it is usually preferable to perform the binary addition on the CPU. Similarly, for chains of binary addition, we do not consider all combinations of where the operations are performed.

6.1.2 Tiling Algorithm

The solutions generated by the fusion algorithm are first translated into an abstract syntax tree representation that makes loops explicit. Similar to the approach taken to reduce recomputation cost when using the space-time tradeoffs cost model (in Section 2.4.7), each fused loop is then split into two loops, a tiling loop and an intra-tile loop, and all intra-tile loops are moved inside any tiling loops.

The tiling algorithm then searches for the tile sizes that minimize the data transfer cost while keeping the computation below the memory limit. Because of the size of the search space, we employ a heuristic hill-climbing search [KKB+03] for enumerating candidate tile sizes. For each set of candidate tile sizes, the memory and transfer costs are calculated using a fine-grained cost model based on detailed measurements of the constituent operations.

As the tile size increases, the memory usage increases while the transfer cost decreases. If a computation fits within GPU memory, the tiling algorithm will determine that the tile sizes for all tensor dimensions can

equal the dimension size, thus undoing any fusion (i.e., the fused tiling loops will have a single iteration). For this case, the tiling algorithm can then re-evaluate the computation with the cost model for CPU computation to determine whether the entire computation should remain on the CPU.

6.1.3 Layout Optimization

Our layout optimization algorithm [LGK+12] is a bottom-up dynamic programming algorithm that enumerates the possible layouts for intermediates and selects the most efficient index permutation and DGEMM library calls for implementing a contraction. Where possible, it selects a layout that allows to eliminate the index permutation of the intermediate. Like the tiling algorithm, it is based on detailed measurements of the constituent operations. Since for each contraction there are only four possible DGEMM invocations to consider, and since in most cases the number of index permutations to consider is sufficiently constrained, the layout optimization algorithm runs in linear time.

While our layout optimization algorithm was designed for unfused and untiled code, it would be straightforward to generalize it to fused tiled code. It would also be possible to run the layout optimization algorithm for every tile size choice considered in the tiling algorithm. This would make the tiling algorithm much more precise, since it could take layout into consideration, but we expect it to increase the running time of the tiling algorithm by approximately an order of magnitude.

Chapter 7
The New TCE Infrastructure

A prototype of the TCE [Hir03] has been successful in introducing many new models into the NWChem suite by developing several useful domain-specific transformations. An optimizing TCE has been developed primarily at the Ohio State University that has explored many novel optimization algorithms. However, the code bases for both the Prototype TCE and Ohio TCE are too inflexible to perform this research. We have identified the following key aspects required for a flexible yet rich research infrastructure going forward:

- Both the Prototype TCE and the Ohio TCE versions were developed as a "black box" transformer–they took in tens to hundreds of lines of input specification of tensor expressions and output tens to hundreds of thousands of lines of parallel C/Fortran code. While very beneficial, chemists desired the ability to manipulate more readable intermediate representations.

- Both the TCE versions were primarily developed before the era of multi-cores, GPGPUs, and heterogeneous computing; they do not provide sufficient support for tree transformations and for generating multi-core parallelism or GPU code. Their monolithic structure is too inflexible to adapt for multi-target code generation.

- The Ohio TCE included several domain-specific transformations that were extremely useful. Nevertheless, it does not take advantage of major recent advances in polyhedral compilation models, such as Pluto [BHRS08a] and PrimeTile [HBB+09b].

- Both the TCE versions do not have front ends for Fortran or C++ for handwritten code.

The long-term goal of this research is for a future version of the TCE to transform both high-level tensor contraction expressions and handwritten tensor contraction code into optimized parallel programs for any hardware platform and for the generated code to interface with quantum chemistry suites and other simulation tools. This will require additional research on data movement optimization and parallelization and on developing the cost models for the compiler to perform these optimizations for a variety of platforms.

This chapter describes the current progress of building a new software infrastructure for the TCE as a foundation for continuing this research.

7.1 Overview

The Ohio TCE was written in Java and uses a simple abstract syntax tree (absyn) data structure for the domain-specific tensor expression language. We are building the new TCE on top of the ROSE compiler framework, which uses SAGE trees for representing absyns. SAGE is the intermediate representation (AST) used by the ROSE compiler. We built a new TCE front end for processing input as a high level specification of a tensor expression and implement optimizations on the absyn as SAGE tree transformations. We chose ROSE as the foundation of the TCE because of its well-designed architecture, the active development and support, and the added recent support for polyhedral optimizations in case we need them.

7.2 The TCE Front End

The TCE front end consists of a parser and simple semantic analysis for a domain-specific tensor expression language. We extended the TCE language to accommodate more features such as specifying domain information (e.g., symmetry information), architectural features, etc. To enable research on cost models for symmetric and block-sparse tensors, we would need support for representing these symmetries and for for generating appropriate code. We have added language support in our tensor expression language for declaring symmetry properties and corresponding declarations.

The parsed expressions are optimized using algebraic transformations and common subexpression elimination to minimize the operation count. The resulting expression trees can then be translated into abstract syntax trees with explicit loop nodes. These parts of the Ohio TCE are stable and were straightforward to port. The abstract syntax trees and any additional information that is passed to later optimizers must be designed to be uniform for tensor expressions and appropriate C++ and Fortran subsets.

In addition to generating error information and other information for use in later optimizations, semantic analysis fixes parts of the TCE internal AST based on a set of criteria that will lead to greater options in the optimizers. These enhancements will ensure correctness of the given code and proved information that can be used for selecting which optimizations can run on which equations or for calculating information that's needed for some of the optimizers. The optimizers could also compute this information later, but it is efficient to calculate it ahead of time and only once. A type system for computing the symmetry properties of intermediates from their subtrees has also been built.

The traversals implemented by semantic analysis are as follows:

Index checking Recognize and validate the indices of a tensor.

Type node creation Generate type information.

Populating the symbol table Generate symbol table information.

Basic type checking and validating legal tensor contractions

7.3 Porting existing optimizers

We have successfully ported the loop fusion algorithm from the Ohio TCE. The loop fusion optimization operates on expression trees. We are currently working on a loop fusion implementation on dependency graphs for handwritten code as discussed in Chapter 5. Our implementation for handling handwritten code (C/C++ only) is built on top of the ROSE compiler framework. Once this implementation is refined, it would be straightforward to merge it into the new infrastructure as a new interface since the input would then be handwritten code as opposed to equations representing tensor contraction expressions. For efficiency and flexibility, it is preferable the fusion algorithm is available for both data structures. Of particular interest for our optimization approach are the TCE cost models for data locality optimization (for generating out-of-core code) and for communication minimization (for generating distributed code).

We have also interfaced with a standalone implementation of operation minimization [HSN+05, HLG+06, LZR+12] written in Python from Ohio State University. This is the latest implementation of operation minimization with support for handling symmetries.

7.4 Translating to ROSE Sage Trees

We developed a new translator that allows converting a TCE abstract syntax tree (AST) to an equivalent ROSE *sage* tree. We use the Minitermite [BP10] library that is built on top of the ROSE compiler framework. Minitermite generates a term-based serialization of an AST constructed by ROSE. The serialization can be imagined as a plain-text term version of the AST. Minitermite parses the term representation and re-creates a ROSE AST and subsequently invokes the ROSE unparser to generate source code. Our translator tries to directly create an equivalent term representation of a ROSE AST from the corresponding TCE expression tree. For each node in the top-down traversal of the TCE expression tree, the translator generates term sequences that would correspondingly map to nodes in the equivalent ROSE AST. Minitermite then takes the term representation generated by the TCE and unparses it to produce the equivalent source (C/C++) code.

Chapter 8
Experimental Evaluation

8.1 Evaluation of the Loop Fusion Algorithm

We have implemented two versions of the loop fusion algorithm: a prototype with simple but inefficient data structures in ML and an optimized implementation in Java as part of the Tensor Contraction Engine (TCE) [BAB+05a]. The ML implementation parses an input file consisting of tensor expressions and index range declarations and generates either pseudo code or a C function for each tensor expression. It provides memory minimization cost models for both static and dynamic memory allocation as well as a space-time tradeoff cost model for static memory allocation. Our ML implementation is available from `http://www.csc.lsu.edu/~gb/TCE/`. The implementation in the TCE only has memory minimization and space-time tradeoff cost models for static memory allocation.

In this section, we present measurements of the memory reduction that can be achieved with loop fusion and performance measurements of the algorithm for different data structure choices and pruning strategies.

8.1.1 Memory Usage

We tested the loop fusion algorithm on a variety of representative equations as shown in Table 8.1. The coupled cluster singles and doubles (CCSD) equations represent an important quantum chemistry model. We used the shorter spin-orbital variant of these equations. Each term in these equations is the contraction of up to five two- and four-dimensional tensors. To illustrate the complexity of these equations, Figure 8.1 shows the factored (parenthesized) CCSD doubles equation in Einstein notation, where indices that occur twice in a term (once as upper and once as lower index) are implicitly summed over. The h indices represent occupied orbitals of range O; the p indices represent virtual orbitals of range V. For our measurements, we used the dimension sizes $O = 50$ and $V = 100$ for modeling a medium-sized molecule.

The coupled cluster singles, doubles, and triples (CCSDT) lambda 1 equation consists of 139 terms of up to six two-, four-, and six-dimensional tensors. The coupled cluster singles, doubles, triples, and quadruples (CCSDTQ) lambda 1 equation is the largest equation in the TCE testsuite with 264 terms of up to six two-, four-, six-, and eight-dimensional tensors. Again, we used $O = 50$ and $V = 100$ for simplicity. In practice, these large models would be used for smaller molecules (with smaller values for O and V) or run on large

TABLE 8.1. Comparison of memory usage.

Equation	terms	no fusion	fusion static alloc.	fusion dynamic alloc.
Example from Section 2.4.4	1	504.68KiB	1.25KiB	1.23KiB
Example from Section 2.4.5	1	16.4GiB	1.08GiB	1.08GiB
CCSD energy	3	381.47MiB	96B	40B
CCSD singles	14	572.24MiB	137.80KiB	98.07KiB
CCSD doubles	31	1.3GiB	193.88MiB	193.77MiB
CCSDT lambda 1	139	1.82TiB	57.52MiB	N/A
CCSDTQ lambda 1	264	8.88PiB	266.73MiB	N/A
CCSDT lambda 1 no op	139	21.68EiB	4.13MiB	N/A
CCSDTQ lambda 1 no op	264	105.88ZiB	8.56MiB	N/A

$$R^{p1p2}_{h1h2} = v\_vvoo^{p1p2}_{h1h2} - t\_vo^{p1}_{h7} * (v\_ovoo^{h7p2}_{h1h2} - 0.5 * t\_vo^{p2}_{h8} * (v\_oooo^{h7h8}_{h1h2} + (v\_ooov^{h7h8}_{h1p3|} - 0.5 * v\_oovv^{h7h8}_{p3p4} * t\_vo^{p4}_{h1}) * t\_vo^{p3|}_{h2} + 0.5 * t\_vvoo^{p5p6}_{h1h2} * v\_oovv^{h7h8}_{p5p6}) + t\_vo^{p3|}_{h2} * (v\_ovov^{h7p2}_{h1p3|} - 0.5 * t\_vo^{p4}_{h1} * v\_ovvv^{h7p2}_{p3p4}) - (f\_ov^{h7}_{p3|} - t\_vo^{p4}_{h4} * v\_oovv^{h4h7}_{p3p4}) * t\_vvoo^{p2p3|}_{h1h2} - t\_vvoo^{p2p7}_{h2h4} * (v\_ooov^{h4h7}_{h1p7} + t\_vo^{p3|}_{h1} * v\_oovv^{h4h7}_{p3p7}) + 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_ovvv^{h7p2}_{p3p4}) + t\_vo^{p3|}_{h2} * (v\_vvov^{p1p2}_{h1p3|} - 0.5 * t\_vo^{p4}_{h1} * v\_vvvv^{p1p2}_{p3p4}) + (f\_oo^{h6}_{h1} + (f\_ov^{h6}_{p6} + t\_vo^{p4}_{h4} * v\_oovv^{h4h6}_{p4p6}) * t\_vo^{p6}_{h1} - t\_vo^{p4}_{h4} * v\_ooov^{h4h6}_{h1p4} - 0.5 * t\_vvoo^{p4p5}_{h1h5} * v\_oovv^{h5h6}_{p4p5}) * t\_vvoo^{p1p2}_{h2h6} + (f\_vv^{p2}_{p3|} - t\_vo^{p4}_{h4} * v\_ovvv^{h4p2}_{p3p4} - 0.5 * t\_vvoo^{p2p4}_{h4h5} * v\_oovv^{h4h5}_{p3p4}) * t\_vvoo^{p1p3|}_{h1h2} + 0.5 * t\_vvoo^{p1p2}_{h6h8} * (v\_oooo^{h6h8}_{h1h2} + t\_vo^{p6}_{h2} * (v\_ooov^{h6h8}_{h1p6} + 0.5 * t\_vo^{p4}_{h1} * v\_oovv^{h6h8}_{p4p6}) - 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_oovv^{h6h8}_{p3p4}) + t\_vvoo^{p1p3|}_{h2h3|} * (v\_ovov^{h3p2}_{h1p3|} - t\_vo^{p5}_{h1} * v\_ovvv^{h3p2}_{p3p5} - 0.5 * t\_vvoo^{p2p5}_{h1h5} * v\_oovv^{h3h5}_{p3p5}) + 0.5 * t\_vvoo^{p3p4}_{h1h2} * v\_vvvv^{p1p2}_{p3p4}$$

FIGURE 8.1. The spin-orbital CCSD doubles equation.

parallel machines. The last two equations are unfactored (i.e., non-operation-minimal) versions of the CCSDT lambda 1 and CCSDTQ lambda 1 equations, in which each term has only a single summation node with multiple summation indices.

Array elements for all examples are assumed to be double precision floating point numbers. For the first two examples, we used the same array dimensions that were used in Section 2.4.4. All result arrays are produced in memory and, except for the first two examples, all inputs are assumed to be in memory. We compared the memory usage calculated by the loop fusion algorithm, both with static and dynamic memory allocation cost models, to the memory usage of a memory-optimal evaluation order without fusion, as computed by Lam et al.'s algorithm [LRB$^+$11]. The numbers presented are those produced by the algorithms, i.e., they do not include the memory usage for input tensors in memory.

For the CCSDT and CCSDTQ equations, the loop fusion algorithm with a dynamic memory allocation cost model is impractical because of the large solution data structures that are produced. For the factorized CCSDT lambda 1 equation, we killed our ML implementation after two weeks. It likely ran into a garbage collection loop. We did not attempt the other CCSDT and CCCSDTQ equations. For the CCSD doubles

equation, the running time of the loop fusion algorithm with a dynamic memory allocation cost model was about four orders of magnitude longer than with the static memory allocation cost model.

For the memory-optimal evaluation order without loop fusion, the unfactored equations require several orders of magnitude more storage than the corresponding factored equations, since outer tensor products are formed that result in intermediate tensors with additional dimensions. As shown in Table 8.1, loop fusion results in dramatic reductions in memory usage for all equations. For the unfactored equations, however, the redundant summation indices result in additional fusion opportunities that allows the memory usage to be reduced even further.

Perhaps surprisingly, the fusion algorithm with the more accurate dynamic memory allocation cost model did not lead to much improvement over the cost computed with the static memory allocation cost model. The reason for this lies in the structure of tensor contraction equations. Since non-summation indices occur only once in each term, each subtree of a contraction will typically have at least one unique index, which in turn results in different fusions for different subtrees. The order of evaluation, and, therefore, the points in the code where intermediates can be allocated and deallocated, will be determined primarily by the fused loop structure. If both subtrees have the same set of fusible indices, such as in the case of binary addition, the order of evaluation as determined by the dynamic memory allocation cost model can result in a more significant reduction in memory usage.

For tensor contraction equations, we feel that the slight improvement in accuracy of the memory usage calculation does not justify the high cost of the dynamic memory allocation cost model. For computational domains, in which large parts of an expression tree can have the same set of fusible indices, the cost of the dynamic memory allocation cost model may be justified.

8.1.2 Experimental Setup

For measuring the performance of the loop fusion algorithm, we used a quad-core Xeon workstation with the architecture shown in Table 8.2. Since all code is sequential and to make the measurements more predictable, we disabled multi-core support, hyper-threading, Intel Turbo Boost (overclocking), and Intel Speed Step (CPU throttling). We also turned off all network interfaces, the X window system, and unnecessary background processes to minimize interrupts.

TABLE 8.2. Configuration of the Intel Xeon workstation.

Processor	Memory	L1 Cache per core	L2 Cache per core	L3 Cache	OS
Intel Xeon E5530 2.4GHz quad-core	4GiB	64KiB	256KiB	8MiB	Linux 2.6.31

For measuring the performance of the loop fusion algorithm, we used our implementation in the TCE, which is written in Java. We compiled the code using javac 1.6.0-15 and ran it on the Java server VM with the command line options $-XX : CICompilerCount = 1$ and $-Xbatch$ to prevent the compiler from running in parallel with itself and to serialize the compiler with the application. This ensures that the measurements are not distorted by background compilation.

The measurements of the loop fusion algorithm were run 100 times for cases that had a wall-clock time of less than 35 seconds, 35 times for cases that had a wall-clock time ranging from 40 seconds to three minutes, five times for cases that had a wall time ranging from three to 90 minutes, and only once for cases that had a wall time greater than 90 minutes. We then calculated average and standard deviation and discarded outliers more than four standard deviations from the average. This process of discarding outliers was repeated twice to eliminate any influence of operating system interrupts. Only for one of the equations (CCSD energy with a static cost model) did this process result in three out of 100 measurements being discarded. We then calculated the error bars for a confidence of 95%. For measurements that were run 100 times, the error bars were no more than 0.35% of the measured value. For all measurements that were run more than once, the error bars were no more than 0.77% of the measured value. In cases where the error bars of similar measurements overlap or where a speedup is not statistically significant, we explicitly discuss this in the text.

8.1.3 Effects of Data Structure Choice and Pruning Strategy on Algorithm Performance

There are large differences in the performance of the loop fusion algorithm depending on the choice of data structure for solution sets and on the pruning strategy. Which data structure and pruning strategy are superior depends on the cost model and in part on the input equations. For the following benchmarks, we use the same input equations as for the memory usage measurements, with the exception of the unfactored CCSDTQ Lambda 1 equation, which is prohibitively expensive for many of the data structure and pruning choices. The main aspect of the cost model effecting data structure and pruning choices is whether it is a one-dimensional or a two-dimensional cost model. We chose memory minimization with static memory allocation and space-time tradeoffs as representative cost models.

One pruning choice that is always beneficial is to eliminate the extension step for the subtree of a unary node if the maximal fusion for that subtree fuses all indices, as we have proved in Section 4.3. Figure 8.2 shows the speedup achieved by this optimization for both memory minimization and space-time tradeoffs. The measurements were performed with linked list solution set data structures and without further pruning optimizations. Except for the small CCSD energy equation, this optimization results in significant improvements, with the benefit being greater for larger equations and for the one-dimensional cost model.

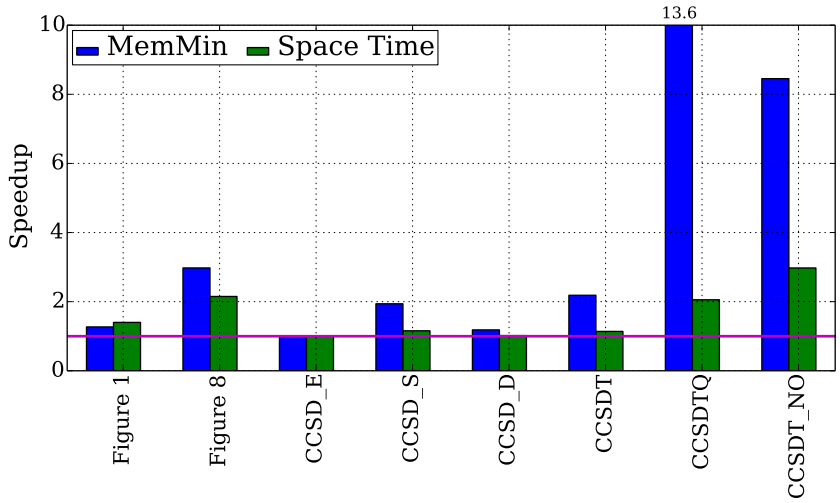


FIGURE 8.2. Speedup achieved by eliminating the extension step below unary nodes where possible.

As discussed in Section 4.1, we can implement sets of solution trees either as linked lists or as hashed sets that use the maximal fusion as the hash key. For a one-dimensional cost model, the only advantage of a hashed set is that it allows pruning of solutions with equal fusions when constructing the set. Because of the overhead of hashing, however, a linked list implementation slightly outperforms a hashed set implementation for most input equations, as shown in Figure 8.3. These measurements were performed without unnecessary extensions below a unary node but without further pruning optimizations. The detailed running times are shown in Table 8.3.

For a two-dimensional cost model such as space-time tradeoffs, a hashed set data structure has the additional advantage that the number of calls to `mergeNesting` can be reduced. We demonstrate the relative impact of pruning solutions with equal fusions and reducing the calls to `mergeNesting` by showing the performance of two solution set data structures with hashing: 1D SolnSet (linked list with hashing) and 2D SolnSet. Like the hashed set data structure for memory minimization, 1D SolnSet only allows pruning solutions with equal fusions when constructing the set. In addition, 2D SolnSet also allows the

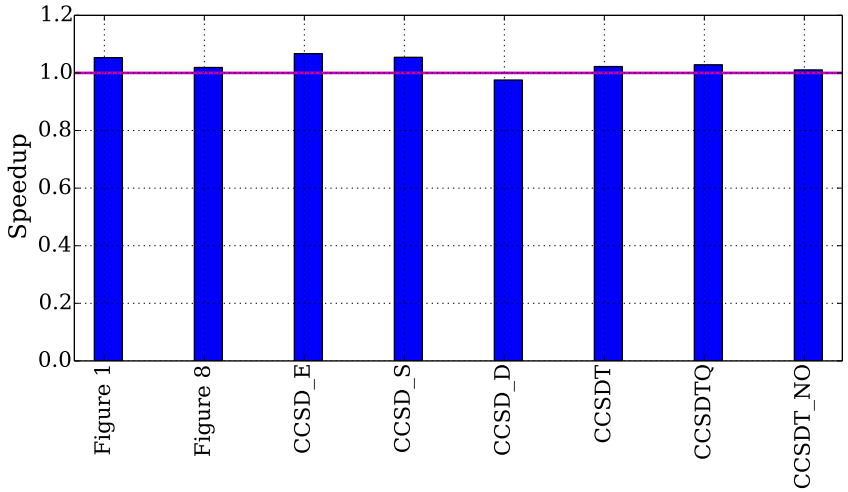


FIGURE 8.3. Speedup of linked lists relative to hashed sets for memory minimization.

TABLE 8.3. Memory minimization running times without the extension optimization.

Equation	no extend opt.	linked list	hashed set
Figure 2.1	6.23ms	4.91ms	5.17ms
Figure 2.8	109.27ms	36.72ms	37.41ms
CCSD energy	3ms	2.99ms	3.19ms
CCSD singles	40.84ms	21.07ms	22.21ms
CCSD doubles	191.77ms	162.12ms	158.11ms
CCSDT L1	4.96sec	2.27sec	2.32sec
CCSDTQ L1	33.7min	2.48min	2.55min
CCSDT L1 no op	98.22min	11.62min	11.74min

reduction of calls to `mergeNesting` when computing solutions for binary nodes. As shown in Figure 8.4, the performance of the algorithm with 1D SolnSet set data structures is slightly worse then with linked lists. For the smaller equations (upto CCSD singles), the overhead of hashing and maintaining the data structure makes 2D SolnSet the most expensive data structure. For equations about the size of CCSD doubles and greater (except for the non-operation-minimal CCSDTQ), using the 2D SolnSet data structure results in substantial performance improvements. Since the differences for small equations are on the order of milliseconds, while for large equations they are on the order of minutes to hours, as shown in Table 8.4, 2D SolnSet is the preferred data structure overall for a two-dimensional cost model. However, it is also straightforward to use linked lists for small equations and 2D SolnSet for equations larger than the CCSD singles equation. Again, these measurements were performed without further pruning optimizations.

Since the size of solution sets can grow exponentially in the number of indices, it is important to prune suboptimal solutions. As we discussed in Section 4.2, however, pruning is quadratic in the size of solution sets and is, therefore, only beneficial if it results in a significant reduction of the size of the solution set.

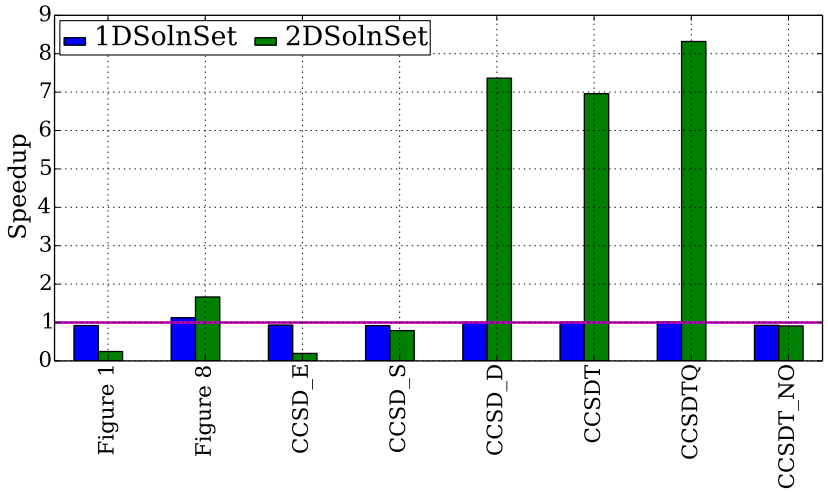


FIGURE 8.4. Speedup of hashed sets relative to linked lists for space-time tradeoffs.

TABLE 8.4. Space-time tradeoff running times without the extension optimization.

Equation	no extend opt.	linked list	1DSolnSet	2DSolnSet
Figure 2.1	8.51ms	6.08ms	6.6ms	24.9ms
Figure 2.8	6.73sec	3.13sec	2.78sec	1.88sec
CCSD energy	4.71ms	4.71ms	5.04ms	24.11ms
CCSD singles	165.06ms	142.71ms	155.25ms	180.73ms
CCSD doubles	15.67sec	15.39sec	15.29sec	2.09sec
CCSDT L1	30.83sec	27.07sec	26.8sec	3.89sec
CCSDTQ L1	186.92min	91min	89.37min	10.94min
CCSDT L1 no op	420.44min	141.33min	152.4min	155.25min

We described the loop fusion algorithm with three calls to `pruneSolnSet`, after extending solutions (`extendSolnSet`), after constructing binary solutions (`makeBinarySolnSet`), and after reducing solutions (`reduceSolnSet`). Extending solutions can result in an exponential increase of the size of the solution set, while constructing binary solutions can result in a quadratic increase. Reducing solutions does not increase the size of the solution set, but since it removes indices from the maximal fusions, many solutions may become inferior to other solutions. Which calls to `pruneSolnSet` have the largest impact depends both on the structure of the input equation and on the cost model.

The pruning choices are denoted by P_XY, which specifies pruning in X and Y. For example, the default case where pruning is performed in all three calls to `pruneSolnSet` is denoted by P_EBR, where B represents the pruning operation performed in `makeBinarySolnSet`, E represents the pruning operation performed in `extendSolnSet`, and R represents the pruning operation performed in `reduceSolnSet`. The pruning choices P_BR, P_ER, P_EB, P_E and P_R are measured.

At a BinOp node, the algorithm first recursively constructs solution sets for the subtrees. These solutions are then extended (by calling `extendSolnSet`) to the indices of the parent, and then the extended solutions are merged (by calling `makeBinarySolnSet`) to form the solutions for the BinOp node. The resulting set is then pruned for inferior solutions. The extended solutions at the subtrees are already pruned in `extendSolnSet` before the merge operation at the BinOp node is performed. Hence, there are very few solutions left after the merge that could possibly be pruned. Therefore, we skip the expensive pruning operation in `makeBinarySolnSet` since it would only eliminate a few solutions. Also, the resulting solution set only grows quadratically during the merge and thereby skipping pruning would not result in a memory bottleneck as the equations get larger. In the `reduceSolnSet` step, by removing some indices in a fusion, other fusions are now comparable and can be removed. Not pruning such solutions leads to the later `extendSolnSet` step extend these additional inferior solutions and then prune them accordingly. Hence, the time saved by skipping pruning in `reduceSolnSet` is spent in the `extendSolnSet` step that follows later, thereby leading to insignificant or no performance improvements as shown in Table 8.5. The number of solutions at a node can grow exponentially with the number of indices when extending solutions. Skipping pruning in `extendSolnSet` would leave a significant number of inferior solutions to be processed in `makeBinarySolnSet`. These inferior solutions are pruned after the merge in `makeBinarySolnSet`, but they are compared with every other solution in the sibling node's solution set during the merge which incurs a greater overall operational cost when the algorithm is run on large equations. This behavior is also indicated in our measurements. Skipping pruning in both `extendSolnSet` and `makeBinarySolnSet` can lead to insufficient system memory very quickly due to the exponential increase in the number of solutions that need to be processed at each level higher up in the tree. In this case, pruning is performed only in `reduceSolnSet`. Our measurements show that in this case, the algorithm runs out of memory for large equations.

The measurements for the one-dimensional static memory allocation cost model are shown in Table 8.5. The pruning choice P_ER (pruning skipped in `makeBinarySolnSet`) results in significant improvements (up to 3.4X) for very large equations like CCSDTQ as shown in Fig 8.5. The actual numbers are shown in Table 8.5. It also shows an insignificant speedup for all other equations. P_E shows performance comparable to P_ER. For the pruning choice P_R, due to the lack of pruning in `extendSolnSet` and `makeBinarySolnSet`, the loop fusion algorithm runs out of memory for equations with more than 20

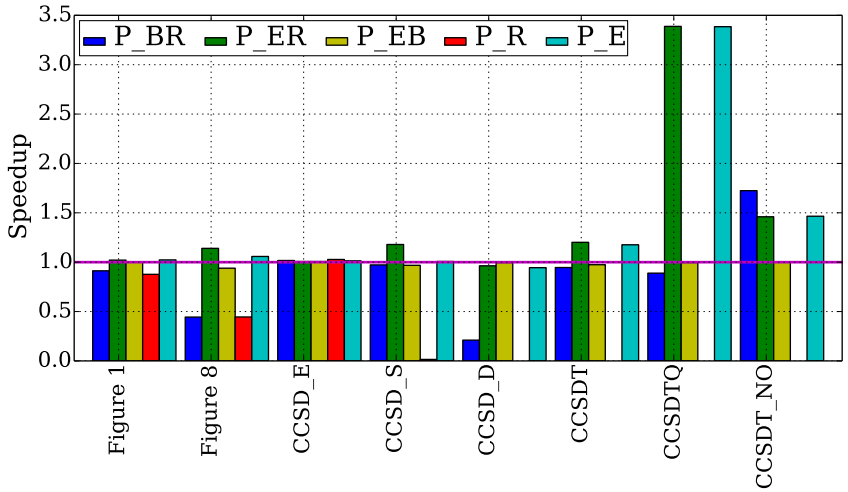


FIGURE 8.5. MemMin — different pruning calls without hashing (relative to linked list).

terms, which is indicated by the gap shown in between the bars in Figure 8.5. Pruning choice P_B has a similar behavior as P_R and is hence not presented. The pruning choice P_EB does not result in any significant performance improvement. The pruning choice P_BR decreases the performance of the loop fusion algorithm, except for the non-operational minimal version of CCSDT lambda 1 as seen in Figure 8.5. This is because more constraints on fusion choices result in more solutions after the extension step. The non-operational minimal equation has very few constraints on fusion choices. Hence, skipping pruning in `extendSolnSet` improves the performance since the resulting solution set is not significantly large enough to benefit from pruning.

The pruning choices measured use the linked list data structure for implementing sets of solution trees. As discussed earlier in this section, the performance of a 1D SolnSet data structure for a one-dimensional cost model is worse than with linked lists due to the overhead of hashing every single solution inserted into the solution tree. The combination of 1D SolnSet and the pruning choices mentioned also has a similar performance trend, i.e., it is worse than the pruning choices implemented using linked lists. Hence, we choose the pruning strategy P_ER which uses the linked list data structure as the default implementation choice for a one-dimensional cost model. We selectively choose P_BR only for large non-operational minimal equations.

The reasoning behind the performance of the different implementation versions for the space-time tradeoffs cost model is the same as explained for the MemMin cost model. The graphs in Figure 8.6 show that for the smaller equations up until CCSD singles, the implementation P-ER or P-E would be a good choice. The actual numbers are shown in Table 8.6.

TABLE 8.5. MemMin — the different pruning numbers.

Equation	P-BR	P-ER	P-EB	P-R	P-E
Figure 2.1	5.38ms	4.81ms	4.89ms	5.6ms	4.8ms
Figure 2.8	82.72ms	32.21ms	39.1ms	82.5ms	34.71ms
CCSD energy	2.94ms	2.97ms	2.97ms	2.91ms	2.95ms
CCSD singles	21.67ms	17.87ms	21.76ms	1.39sec	20.92ms
CCSD doubles	0.77sec	168.31ms	162.84ms	OOM	171.68ms
CCSDT L1	2.4sec	1.89sec	2.33sec	OOM	1.93sec
CCSDTQ L1	2.79min	43.92sec	2.49min	OOM	43.96sec
CCSDT L1 no op	6.74min	7.96min	11.64min	OOM	7.93min

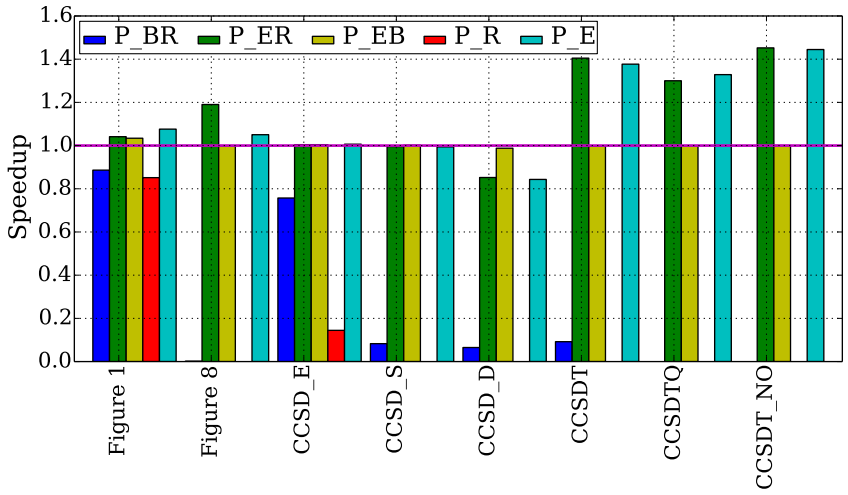


FIGURE 8.6. Space-time tradeoffs — different pruning calls without hashing (relative to linked list).

TABLE 8.6. Space-time tradeoffs — the different pruning numbers without any hashing.

Equation	P-BR	P-ER	P-EB	P-R	P-E
Figure 2.1	6.86ms	5.84ms	5.88ms	7.14ms	5.65ms
Figure 2.8	28.25min	2.63sec	3.13sec	OOM	2.98sec
CCSD energy	6.22ms	4.69ms	4.69ms	32.58ms	4.68ms
CCSD singles	1.72sec	143.33ms	142.47ms	OOM	143.57ms
CCSD doubles	3.94min	18.06sec	15.58sec	OOM	18.25sec
CCSDT L1	4.9min	19.27sec	27.12sec	OOM	19.66sec
CCSDTQ L1	OOM	70min	91min	OOM	68.5min
CCSDT L1 no op	OOM	97.32min	141.26min	OOM	97.83min

For the space-time cost model, the 2D solution set implementation is needed in order to achieve far more significant performance improvements for larger equations. We observe from Figure 8.7 that the best choice for large equations (CCSD doubles onwards) is the implementation strategy 2DSolnSet-P-E. The actual numbers that prove this fact can be found in Table 8.7.

The behavior of the loop fusion algorithm depends on the structure of the equation. For the non-operational minimal equations, we choose the 1D SolnSet implementation with either P-ER or P-E as the pruning strategy. Non-operational minimal code results in less constraints amongst the fused loop structures. There are redundant edges between a child and its parent, which results in more fusion possibilities. The measurements for non-operational minimal equations are just a result of one run. We did not measure the performance of non-operational minimal versions of all equations considered in the measurements since the operational minimal form of the equations would mostly be used in practice.

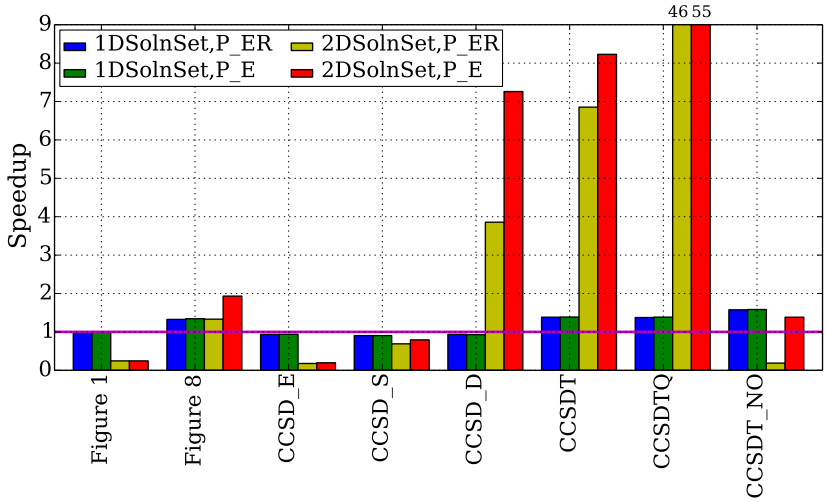


FIGURE 8.7. Space-time tradeoffs — pruning calls with hashing, 2D solution sets (relative to linked list).

TABLE 8.7. Space-time tradeoffs — the different pruning numbers with hashing.

Equation	1DSolnSet-P-ER	1DSolnSet-P-E	2DSolnSet-P-ER	2DSolnSet-P-E
Figure 2.1	6.34ms	6.14ms	24.8ms	24.77ms
Figure 2.8	2.36sec	2.33sec	2.35sec	1.62sec
CCSD energy	5.03ms	5.02ms	26.45ms	24.05ms
CCSD singles	157.84ms	157.82ms	207.01ms	180.05ms
CCSD doubles	16.43sec	16.58sec	3.99sec	2.12sec
CCSDT L1	19.57sec	19.51sec	3.95sec	3.29sec
CCSDTQ L1	66.2min	65.7min	1.96min	1.65min
CCSDT L1 no op	89.72min	89.25min	752.52min	102.19min

To summarize, we choose the implementation P-ER for both memmin and space-time tradeoffs cost models. For larger equations with space-time tradeoffs cost model, we choose 2DSolnSet-P-E. For non-operational minimal equations, we choose 1DSolnSet-P-ER for space-time tradeoffs cost model.

8.2 Evaluation of the Performance of the Generated Code

A quad-core Xeon workstation whose configuration is shown in Figure 8.8 was used for all measurements. The machine was running Linux 3.5.0 x86-64 in single user mode. To make the measurements more predictable, we disabled hyper-threading, Intel Turbo Boost (overclocking), and Intel Speed Step (CPU throttling). All network interfaces, the X window system, and unnecessary background processes are turned off to minimize interrupts. The wall clock times are reported. For the CPU measurements, the sequential version of the Intel Math Kernel library (MKL) [MKL] was used with the Intel C/C++ compiler [ICC]. MKL provides high performance dense linear algebra libraries and includes the full BLAS [DCHD90, LHKK79] API. It uses empirical timings to tune the BLAS library to arbitrary machines thereby providing highly optimized and parallelized thread-safe BLAS routines that are tuned for maximum performance on Intel processors. For multi-core measurements, the threaded version of Intel MKL library is used. For array index permutation operations, the state-of-art index permutation library routines (CPU only) [LKS06] is used. For GPU measurements, the NVIDIA CUBLAS library [CBL] is used with NVIDIA CUDA compiler [NVC]. The Accelereyes ArrayFire library [ACC] was used for performing index permutation on the GPU. The polyhedral source-to-source compilers used are Pluto [BHRS08a] (for generating optimized sequential and multi-core code) and PPCG [VJC+13] (for generating optimized GPU code).

8.2.1 TCE-Generated Sequential Fortran Code

For measuring the performance of the code generated by the loop fusion algorithm, we compiled the TCE-generated Fortran code with both the Intel Fortran and GNU Fortran compilers. We demonstrate the benefits of tiling the fused code where the inner-most loop nests are replaced with calls to the Intel MKL Library. For demonstrating the effect of transforming the generated code with Pluto, we used our stand-alone implementation of the loop fusion algorithm in ML to generate C code, which was then transformed by Pluto. We manually verified that the code structure and the layout of intermediates are identical between the generated Fortran and C codes, i.e., that the order of loops for perfectly nested loop nests are the same,

- Processor: Quad-core Intel Xeon CPU E5530 @ 2.4GHz
- TLB: 512 entry
- Cache Configuration:
  - L1 - Instruction Cache = 32 KB, per core
  - L1 - Data Cache = 32 KB, per core
  - L2 - 256 KB Mid-Level Cache per core
  - L3 - 8 MB shared among 4 cores
- CPU Memory: 12GB, Virtual Memory: 10GB
- GPU: Tesla C2075. Advertised Memory: 6GB, Usable Memory: 5.2GB
- Compilers: icc 13.0 (-O2), nvcc 5.0 (-O2)
- BLAS Libraries: Intel MKL 11.0 (CPU), CUBLAS 5 (GPU)
- Polyhedral tools: Pluto, PPCG-CUDA
- GPU Libraries: ArrayFire, SciGPUGEMM
- Pluto Options: –smartfuse –tile –isldep –lastwriter (Sequential)
  –smartfuse –tile –isldep –lastwriter –parallel (OpenMP)

FIGURE 8.8. T5500 Configuration

and that the order of array subscripts in C code is the reverse of the order of the corresponding subscripts in Fortran code.

The measurements of the generated code were averaged over 100 times after warming up the cache for cases that had a wall-clock time of less than two minutes, 5 times for cases that had a wall-clock time ranging from two to sixty minutes, and only once for cases that had a wall-clock time greater than an hour. Since the differences between competing measurements were fairly large, there was no need for computing error bars.

We now demonstrate the performance of the code generated by the loop fusion algorithm. The dimension sizes *O=48, V=96* were used for the measurements. The measurements were averaged over 100 times for cases that had a wall-clock time of less than two minutes, 5 times for cases that had a wall-clock time ranging from two to sixty minutes, and only once for cases that had a wall-clock time greater than an hour. For discussion purposes, the Fortran code generated by the loop fusion algorithm is referred to as *fused* code and the code generated by disabling the loop fusion algorithm as *unfused* code. We also demonstrate the benefits of tiling the fused code by tiling the fused loops and replacing the inner-most loop nests with BLAS calls. This is achieved by using the tiling algorithm implemented in the TCE where the fused array dimensions are expanded back to tile size such that the computation still fits in memory. The performance of the following TCE-generated Fortran codes is measured for the equations CCSD Singles and CCSD Doubles.

- Unfused code generated by disabling the loop fusion algorithm marked with *UF* in the tables in this section.

- Unfused code where loop nests performing a tensor contraction are replaced with BLAS (dgemm) calls marked with *UF-dgemm*.

- Fused code generated by the loop fusion algorithm marked with *F*.

- Fused code where fused loops are tiled and the inner-most loop nests are replaced with BLAS (dgemm) calls.

The tiled and dgemm code versions are not available for CCSD energy and the equations shown in Figure 2.1 and Figure 2.8. The measured numbers for the remaining code versions (unfused and fused) for these equations are shown in Table 8.8. The data cache (level 1, 2) and TLB misses explain the difference between fused and unfused code performance. The numbers for the CCSD singles and CCSD doubles codes

TABLE 8.8. Performance of the generated code for O=48, V=96.

Equation	Runtime	DL1 misses	DL2 misses	DTLB misses
Figure 2.1 (UF)	534μs	23,075	7,986	5,584
Figure 2.1 (F)	159.8μs	6,379	3,703	1,649
Figure 2.8 (UF)	10.8min	8,437,301,746	4,807,759,206	31,786,447,050
Figure 2.8 (F)	78min	256,901,811,413	75,923,237,384	292,979,081,477
CCSD energy (UF)	41.1ms	10,210	5,010	208,395
CCSD energy (F)	35ns	5,956	3,179	329

TABLE 8.9. Performance of the generated code for CCSD singles, doubles using O=48, V=96.

Equation	Runtime	DL1 misses	DL2 misses	DTLB misses
CCSD singles (UF-dgemm)	4.7sec (ifort)	293,198,347	127,965,122	99,907,776
CCSD singles (UF-ifort)	5.94sec	41,299,817	10,093,396	74,386,673
CCSD singles (F-ifort)	21.8sec	619,024,721	482,381,831	1,144,772,703
CCSD doubles (UF-dgemm)	2.74min (ifort)	14,404,377,306	767,589,367	151,488,884
CCSD doubles (UF-ifort)	20.2min	10,782,143,759	1,843,304,123	61,543,123,898
CCSD doubles (F-ifort)	2.74min	21,619,692,309	2,065,750,526	3,826,376,040

without tiling are shown in Table 8.9 and the codes with both tiling and dgemm calls are shown in Table 8.10. The numbers for the *fused-dgemm* version (without any tiling) are not available since the code generator fails to produce that code.

We observe that fusion results in bad data layout when we have reasonably large arrays, thereby resulting in poor cache behavior. By tiling however, we begin to get back the performance due to enhancing data locality as shown in Tables 8.9, 8.10 for both the dgemm and non-dgemm code versions. The non-dgemm fused-tiled code performs better when compiled with gcc. Also, interestingly, for the non-dgemm versions, we see the performance drop beyond tile size 8. We have not figured out the reason for this behavior yet.

The TCE-generated Fortran code is also compared with the optimized C code generated by Pluto [BHRS08a], an automatic parallelizer based on the polyhedral model that optimizes sequences of im-

TABLE 8.10. Performance of the generated Fused-tiled Fortran code for O=48, V=96.

Equation	TileSize	nodgemm (gcc)	dgemm (ifort)
CCSD singles	4	11.5sec	12.6sec
	8	8.94sec	6.5sec
	12	9.53sec	5.05sec
	16	13.8sec	4.76sec
	24	23.4sec	4.12sec
CCSD doubles	4	23.4min	5.75min
	8	26.5min	3.65min
	12	31.1min	3.2min
	16	67.6min	3.03min
	24	116.26min	2.97min

perfectly nested loops, simultaneously for parallelism and locality through tiling transformations. Given an input sequential (C) code, it can automatically generate tiled parallel OpenMP code for multi-core processors. We use Pluto to generate tiled (sequential) code optimized for locality using the (C) code generated from the fusion algorithm as input. The input is generated using our standalone implementation of the loop fusion algorithm in ML implementation. The tiled code generated by Pluto is compared against the fused code generated by the TCE and also against the TCE-generated fused code which is then tiled. We manually verified that the code structure and the layout of intermediate tensors are identical between the generated Fortran and C codes.

The running times for the TCE generated CCSD singles and doubles code versions are shown in Tables 8.9, 8.10 and those generated using Pluto in Table 8.11. Even when only considering the non-dgemm code, for CCSD singles, both the Pluto and TCE generated versions have identical performance; and for larger equations like CCSD doubles, for fused as well as unfused code, Pluto is considerably slower than the TCE. For the dgemm code versions, the TCE significantly outperforms Pluto since the later does not have any mechanism for leveraging the use of optimized dgemm routines.

TABLE 8.11. Running times of generated code optimized with Pluto v0.9.(O=48, 0+V=96).

Equation	Version	Memory (F)	Memory (UF)	RunTime (UF)	RunTime (F)
CCSD singles	POT	91.32MiB	1.03GiB	17.02sec	13.04sec
	PDT	891.32MiB	1.03GiB	17.06sec	22.84sec
	PD	891.32MiB	1.03GiB	6.4sec	22.6sec
CCSD doubles	POT	2.26GiB	8.82GiB	111min	74min
	PDT	2.26GiB	8.82GiB	115min	75min
	PD	2.26GiB	8.82GiB	27min	75min

The *fused* and *unfused* C codes generated by the standalone ML implementation of the loop fusion algorithm are passed as inputs to Pluto. We using different combinations of optimization options provided by Pluto. The performance of the transformed codes (given as follows) generated by Pluto are then measured

- PD: Use Pluto smartfuse option. A heuristic fusion algorithm is applied.
- PDT: PD together with tiling for locality and tile a second time (for L2 cache).
- POT: Tiled for locality and tile a second time for L2 cache (PDT) together with each of the following:
  - Disable Pluto's fusion when fused code is provided as input to Pluto.
  - Use PDT when unfused code is provided as input to Pluto.

8.2.2 Multi-core and GPGPU code

To further demonstrate the effectiveness of our fusion algorithm and compilation framework, we used handwritten code for the 4-index transformation equation shown as follows and optimized it for single-core, multi-core, and GPGPU execution:

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

In the handwritten code, all loop bounds are compile-time constants and loops may or may not have been fused. This code is then optimized by our loop fusion algorithm that can handle handwritten code, such that the storage requirements for intermediate tensors are minimized. This results in increased temporal locality between producer and consumer, but causes poor temporal locality within a contraction because of the elimination of entire tensor dimensions. Dimensions that were eliminated were then manually expanded to tiles to improve temporal locality within a contraction while staying within the memory limit. The inner-most loop nests, which now represent the tensor contraction on a tile, were then manually replaced by efficient index permutation and matrix multiplication (dgemm) library calls. We generated versions of the code for single-core, multi-core, and NVIDIA GPGPU architectures.

We compared the performance of the code resulting from our optimization approach with code produced by state-of-art compilers employing polyhedral model optimizations. For single-core and multi-core code, we compared against code generated by Pluto, Version 0.9.0-36-g5fb218a [BHRS08a]. For GPUs, we compared against PPCG, Version c7179a0 [VJC+13]. As input to these tools we used both *unfused* and *fused* code. In all cases, the code generated by our optimization approach outperformed the code generated by the polyhedral model optimizers by at least a factor of two.

A naive way to implement the computation is to perform each contraction inside a set of perfectly-nested loops and reuse the resulting array from one contraction to the other. We refer to this implementation of the 4-index transform as the *unfused* version. In order to enhance the performance further, each contraction inside a set of perfectly-nested loops is replaced with a dgemm call as shown in Figure 8.9. After each contraction, an index permutation on the resulting intermediate array is performed in order to get it into the right layout for the dgemm call that performs the next contraction. By applying the loop fusion algorithm, the producer and consumer loops of a memory resident intermediate array are fused thereby reducing the

memory requirements by eliminating the array dimensions indexed by the fused loops. The intermediate arrays resulting from each contraction are thus produced and consumed in a pipelined fashion, and they reuse the same reduced array space. We then tile the fused loops in order to enhance data locality. We refer to this implementation as the *fused-tiled* version and is shown in Figure 8.10.

Loop fusion results in bad data layout when we have reasonably large arrays, thereby resulting in poor cache behavior. We would need additional data layout optimizations after fusion and before tiling to improve the performance. Our measurements show that performing data layout optimization after fusion and then followed by tiling brings the performance of fused code close to that of the unfused code. These data layout optimizations were done manually. They involve permuting array indices of temporary arrays that are generated during the computation to bring them into a proper layout for subsequent BLAS (dgemm) routines in the remaining computation. With tiling, since the array accesses are on a per-tile basis, copying each tile into a buffer and then operating on the buffer provides better cache behavior.

A tile of the 4 dimensional input array A and the smaller array $C1$ need to be copied into a contiguous buffer. The additional overhead of a copy operation and that of maintaining an additional buffer is required since array elements belonging to tiles of these arrays are not contiguous in the corresponding memory locations which is a requirement for a dgemm call. For each version, we have a CPU implementation that runs on a single, four core(s) and a GPU implementation evaluated using the NVIDIA Tesla C2075 [NVD] GPU.

```
dgemm('T', 'N', V, O3, O, 1.0, C4, V, A, O3, 0.0, T1, O3);
index_permute(T1, T1_T, V, O, O, O, 2, 1, 3, 4, 1.0);
dgemm('T', 'N', V, VO2, O, 1.0, C3, V, T1_T, VO2, 0.0, T2, VO2);
index_permute(T2, T2_T, V, V, O, O, 3, 1, 2, 4, 1.0);
dgemm('T', 'N', V, V2O, O, 1.0, C2, V, T2_T, V2O, 0.0, T3, V2O);
index_permute(T3, T3_T, V, V, V, O, 4, 2, 1, 3, 1.0);
dgemm('T', 'N', V, V3, O, 1.0, C1, V, T3_T, V3, 0.0, B, V3);
index_permute(B, B_T, V, V, V, V, 4, 2, 3, 1, 1.0);
```

FIGURE 8.9. Unfused Code

In this section, we present measurements of the memory reduction that can be achieved with loop fusion and the performance measurements of both these versions. For the *unfused* GPU version, all the input arrays are copied to the GPU and the result array B is copied back to host memory after the last contraction is finished. Index permutation for each of intermediate arrays is performed on the GPU using index permutation routines available in Accelereyes ArrayFire library. Similarly for the GPU version of *fused-tiled* code, the

buffers holding tiles of input arrays are also copied to the GPU. The result array is copied back to the host (CPU) memory whenever each tile of it is being produced. We use page-locked (pinned) host memory when allocating arrays on CPU initially for achieving better bandwidth during data copy between CPU and GPU memories.

```
for (s = 0; s < O; s = (s + T)) {
  copy_tile_A(A, ABuffer);
  copy_tile_C1(C1, C1Buffer);

  for (a = 0; a < V; a = a+T) {
     dgemm('N', 'N', T, O2T, O, 1.0, C4 + (a * O),
          O, ABuffer, O2T, 1.0, t1, O2T);
     index_permute(t1, t1_t, dim1, dim2, dim3, dim4, 2, 3, 4, 1, 1.0);

     for (b = 0; b < V; b = b+T) {
        dgemm('N', 'N', T, OT2, O, 1.0, (C3 + (b * O)),
             O, (t1_t + 0), OT2, 1.0, t2, OT2);
        index_permute(t2, t2_t, dim1, dim2, dim3, dim4, 2, 3, 4, 1, 1.0);

        for (c = 0; c < V; c = c+T) {
           dgemm('N', 'N', T, T3, O, 1.0, (C2 + (c * O)),
                O, t2_t , T3, 1.0, t3, T3);
           index_permute(t3, t3_t , dim1, dim2, dim3,
                       dim4, 2, 3, 4, 1, 1.0);

           dgemm('N', 'N', V, T3, T, 1.0, C1Buffer, T,
                t3_t, T3, 1.0, (B+index), T3);
         }
     }
  }
}
index_permute(B, B_T, V, V, V, V, 4, 2, 3, 1, 1.0);
```

FIGURE 8.10. Fused-tiled Code

8.2.3 Code Versions

We tested the following code versions.

- Baseline
  - *Sequential code:* This version contains simple loop nests with a contraction inside the innermost loop. This code is also provided as input to Pluto, PPCG.
  - *Multi-core (OpenMP) code:* This version contains $omp\ parallel\ for$ pragmas around outermost loop nests.

- *TCE optimized code:* Our implementation built using the ROSE compiler framework can generate either unfused or fused code but does not yet replace innermost loop nests with library calls. We, therefore, inserted index permutation and BLAS calls manually.

- *Pluto, PPCG optimized code:* Since the Pluto and PPCG polyhedral compilers use C code as input, we used our ML implementation to generate the sequential (baseline) code, which is then provided as input to Pluto and PPCG. x Pluto is used to generate optimized sequential and multi-core (tiled) code. PPCG is used to generate optimized GPU (tiled) code.

- *Cases discussed in measurements:*

  - For CPU measurements: Unfused code, Fused code without tiling, and Fused and Tiled code.

  - For GPU measurements: Fused code, Fused and Tiled code, Unfused code, and Unfused and Tiled code that was automatically tiled for out-of-gpu-core execution using the SCIGPUGEMM library.

Each of the tests (except the case where V=120, O+V=180) was run for 5 times and averaged. The results for the CPU versions are shown in Tables 8.12 and 8.13.In all the tables shown from here onwards, if the tile size is not available, it means that the highlighted row corresponds to the *unfused* case. When fusing away dimensions of large intermediate arrays, we found fusion to have a negative impact on the data layout of the arrays thereby resulting in poor cache behavior even though it reduces memory requirements. By tiling however, we begin to get back the performance due to enhancing data locality. This is demonstrated from the results in Table 8.12. As the tile size gets bigger the performance of the *fused-tiled* case is almost equal to that of the *unfused* version. Also with bigger tile sizes, the advantage of memory reduction due to fusion is lost. For larger tile sizes, the *fused-tiled* version uses more memory than *unfused* version due to the allocation of separate contiguous buffers that hold tiles of the large input matrix A. However, for all practical simulations, when the molecule sizes get large enough as shown in Table 8.13, the *unfused* version does not fit in physical memory anymore.

It is safe to say that as the computation scales up further, disk I/O would be required and a larger negative impact on performance can be expected. As shown in Table 8.13, the computation relies on virtual memory and the performance drops significantly as indicated by the difference between wall-clock and CPU times.

Here applying fusion reduces the storage requirements for the intermediate arrays produced after each contraction allowing the computation to fit within the available memory. We then perform tiling and choose a tile size which allows the computation to fit within available memory. As observed in Table 8.13, tile size 28 gives the best performance while allowing the computation to fit within available memory limit. Larger tile sizes again force the computation to tap into the virtual memory space or rely on disk I/O. The performance of these code versions on the GPU is similar and are discussed shortly.

TABLE 8.12. Sequential Runs on a CPU.

V	O+V	TileSize	WallTime	IndexPermute	CopyTime	gflops	Memory
70	80	—	3.96secs	0.70secs	—	3.70	0.58 GB
70	80	8	4.46secs	0.35secs	0.35secs	3.28	0.53 GB
70	80	16	4.18secs	0.46secs	0.28secs	3.50	0.57 GB
70	80	20	4secs	0.47secs	0.27secs	3.66	0.60 GB
70	80	28	4secs	0.58secs	0.24secs	3.66	0.67 GB
70	80	40	3.91secs	0.65secs	0.19secs	3.75	0.79 GB
70	80	48	3.98secs	0.68secs	0.19secs	3.67	0.89 GB
100	120	—	22.15secs	3.10secs	—	4.52	2.84 GB
100	120	8	30.58secs	1.94secs	2.17secs	3.28	2.41 GB
100	120	16	25.72secs	2.03secs	1.48secs	3.90	2.56 GB
100	120	20	25.35secs	2.56secs	1.36secs	3.95	2.64 GB
100	120	28	24.17secs	2.76secs	1.28secs	4.14	2.82 GB
100	120	40	23.22secs	2.83secs	1.04secs	4.31	3.15 GB
100	120	60	23.07secs	3.08secs	0.90secs	4.34	3.84 GB
120	160	—	67.60secs	7.41secs	—	5.06	8.55 GB
120	160	8	98.39secs	4.82secs	6.95secs	3.48	6.70 GB
120	160	16	86.34secs	5.81secs	4.70secs	3.96	7.02 GB
120	160	20	80.48secs	6.65secs	4.53secs	4.25	7.20 GB
120	160	28	76.50secs	7.06secs	4.09secs	4.47	7.59 GB
120	160	40	73.33secs	7.96secs	3.32secs	4.66	8.26 GB
120	160	60	70.75secs	7.85secs	3.04secs	4.83	9.64 GB

The running times for the Pluto optimized sequential and multi-core codes are shown in Table 8.14 and Table 8.15. The running times for the TCE optimized sequential codes are shown in Table 8.16 and Table 8.18 and those for the TCE optimized multi-core codes are shown in Table 8.17 and Table 8.19. As we observe from these tables, in all cases, the TCE optimized codes outperform Pluto by a factor of upto 5.6X for sequential code and upto 13.3X for multi-core code. For completeness, the running times for the unoptimized sequential codes are shown in Table 8.20 and Table 8.23 and those for the unoptimized multi-core codes are shown in Table 8.21 and Table 8.22. The performance of *fused-tiled* code on the GPU is shown in Table 8.24.

TABLE 8.13. Sequential Code Performance on CPU for V=120, O+V=180

TileSize	WallTime	CPUTime	IndexPermute	CopyTime	gflops	Memory
—	27.85min	1.83min	2.05min	—	0.30	13.04 GB
8	2.29min	2.27min	5.59secs	12.65secs	3.60	9.75 GB
16	1.93min	1.91min	6.01secs	8.38secs	4.28	10.19 GB
20	1.87min	1.85min	8.25secs	7.26secs	4.42	10.43 GB
28	1.77min	1.75min	8.34secs	6.72secs	4.66	10.97 GB
32	2.38min	1.8min	9.17secs	41.59secs	3.48	11.26 GB
40	8.11min	1.91min	1.34min	3.7min	1.02	11.88 GB
48	12.13min	2.02min	48.13secs	6.1min	0.69	12.57 GB
60	20min	2.15min	1.94min	8.15min	0.42	13.72 GB

TABLE 8.14. Pluto Optimized Sequential Code

O+V	V	Unfused	Fused
80	70	8.48secs	10.61secs
120	100	49.80secs	90.21secs
160	120	379.04secs	402.16secs

TABLE 8.15. Pluto Optimized Multi-core Code

O+V	V	Unfused	Fused
80	70	4.88secs	36.63secs
120	100	29.70secs	169.86secs
160	120	216.02secs	442.56secs

TABLE 8.16. TCE Optimized Sequential Untiled Code

O+V	V	Unfused	Fused
80	70	4.02secs	8.86secs
120	100	21.94secs	55.78secs
160	120	66.70secs	181.05secs

TABLE 8.17. TCE Optimized Multi-core Untiled Code

O+V	V	Unfused	Fused
80	70	2.21secs	8.72secs
120	100	10.12secs	46.21secs
160	120	27.99secs	146.79secs

TABLE 8.18. TCE Optimized Sequential Tiled Code

O+V	V	T	Fused-tiled
80	70	16	4.42secs
80	70	20	4.13secs
80	70	28	4.10secs
120	100	16	25.72secs
120	100	20	25.45secs
120	100	28	24.22secs
120	100	32	25.35secs
120	100	40	23.24secs
120	100	60	22.87secs
160	120	16	88.66secs
160	120	20	81.67secs
160	120	28	77.05secs
160	120	32	79.28secs
160	120	40	74.39secs
160	120	60	71.34secs

TABLE 8.19. TCE Optimized Multi-core Fused-tiled Code

O+V	V	T	Fused-tiled
80	70	16	2.48secs
80	70	20	2.37secs
80	70	28	2.36secs
120	100	16	13.65secs
120	100	20	13.75secs
120	100	28	12.51secs
120	100	32	13secs
120	100	40	11.61secs
120	100	60	11.50secs
160	120	16	45.88secs
160	120	20	41.09secs
160	120	28	37.08secs
160	120	32	38.43secs
160	120	40	35.88secs
160	120	60	33.18secs

TABLE 8.20. Unoptimized Sequential Untiled Code

O+V	V	Unfused	Fused
80	70	8.54secs	10.58secs
120	100	50.08secs	90.71secs
160	120	380.54secs	401.31secs

TABLE 8.21. Unoptimized Multi-core Untiled Code

O+V	V	Unfused	Fused
80	70	11.59secs	35.94secs
120	100	55.97secs	166.95secs
160	120	724.33secs	439.40secs

TABLE 8.22. Unoptimized Multi-core Fused-tiled Code

O+V	V	T	Fused-tiled
80	70	16	19secs
80	70	20	13.52secs
80	70	28	15.77secs
120	100	16	119.55secs
120	100	20	94.83secs
120	100	28	134.71secs
120	100	32	175.48secs
120	100	40	201.30secs
120	100	60	320.14secs
160	120	16	375.42secs
160	120	20	302.84secs
160	120	28	353.24secs
160	120	32	705.20secs
160	120	40	606.82secs
160	120	60	859.03secs

TABLE 8.23. Unoptimized Sequential Fused-tiled Code

O+V	V	T	Fused-tiled
80	70	16	26.49secs
80	70	20	25.20secs
80	70	28	36.36secs
120	100	16	178.25secs
120	100	20	167.15secs
120	100	28	247.80secs
120	100	32	342.12secs
120	100	40	296.51secs
120	100	60	483.46secs
160	120	16	580.48secs
160	120	20	549.09secs
160	120	28	781.01secs
160	120	32	1163.23secs
160	120	40	901.50secs
160	120	60	1430.56secs

TABLE 8.24. Performance of Fused-tiled Code on GPU

O+V	V	T	Fused-tiled
80	70	16	1.51secs
80	70	20	1.40secs
80	70	28	1.29secs
80	70	32	1.28secs
80	70	40	1.22secs
80	70	48	1.23secs
120	100	16	7.69secs
120	100	20	6.79secs
120	100	28	6.41secs
120	100	32	6.14secs
120	100	40	6.02secs
120	100	48	6.02secs
120	100	60	5.80secs

We now compare the different code versions. For sequential and multi-core (OpenMP) versions, baseline code is compared with Pluto generated code. Unfused code uses 8.5GB of memory and fused code uses 4.88GB. As seen in Fig. 8.11, the performance of the code generated by Pluto is only as good as the performance of the input code provided to it. The exception is in the case of OpenMP unfused code where Pluto does better optimization, a factor of 2X.

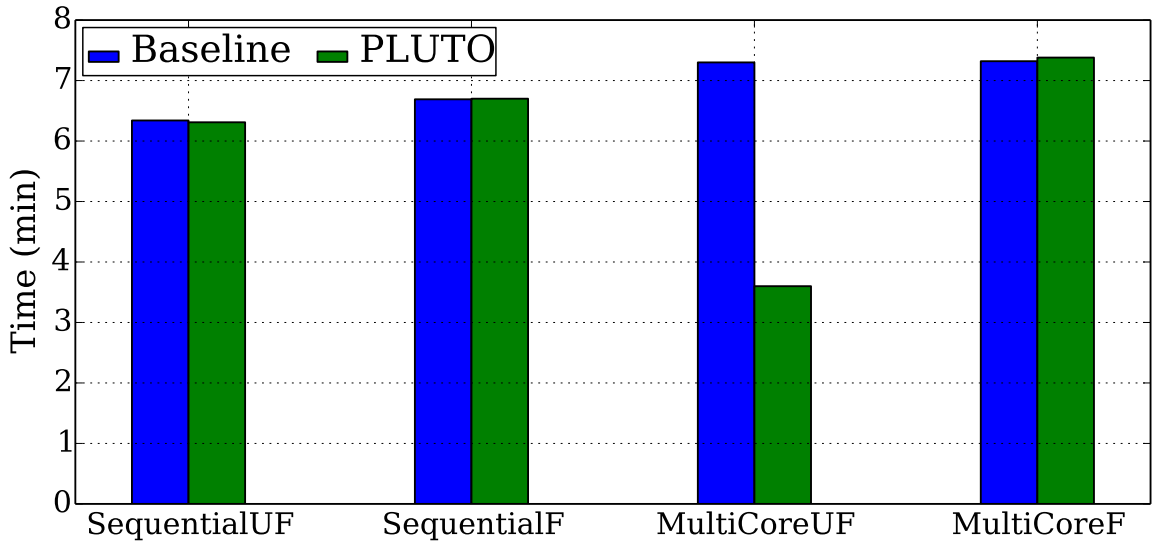


FIGURE 8.11. Baseline vs Pluto Run Times for V=120, O+V=160

TCE optimized code is compared with Pluto generated code. Here *unfused* and *fused* code refers to baseline, fused codes where loop nests are replaced with a combination of index permutation and BLAS (dgemm, dgemv) calls. Multi-core code refers to TCE optimized sequential code which is compiled with multi-threaded version of the Intel MKL library (which uses OpenMP for parallel execution). Pluto generates optimized sequential and OpenMP tiled code using baseline unfused, fused code as inputs. The generated code is compared with the TCE optimized code. As shown in Table. 8.25 and Fig. 8.12, the TCE optimized code outperforms Pluto generated code by a factor of 5.7-7.7X in case of *unfused* code and a 2.2-3X in case of *fused* code.

TABLE 8.25. TCE Optimal vs Pluto (secs) for V=100, O+V=120

SeqUnfused	SeqFused	OpenMPUnfused	OpenMPFused
66.7	181.01	27.99	146.79
379.04	402.16	216.02	442.56

TCE optimized untiled code (in-core) is compared across all versions: sequential MKL, multi-threaded MKL and GPU code. Baseline (unoptimized) version is also used. Maximum memory used by the *unfused*

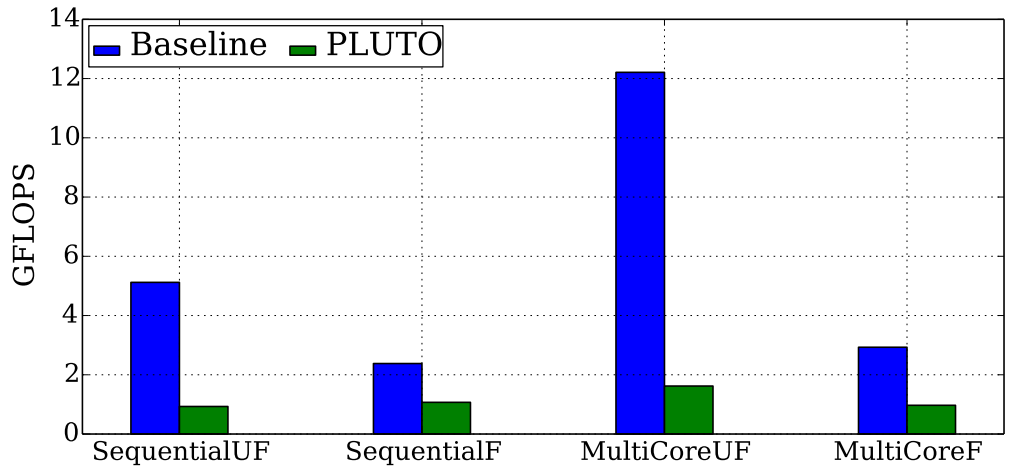


FIGURE 8.12. TCE Optimal vs Pluto (FLOPS) for V=100, O+V=120

code is 2.83GB and *fused* code is 1.54GB. The running times are shown in Table 8.26. As shown in Fig. 8.13, the GPU version outperforms the remaining versions in the case of *unfused* code. For *fused* code, the GPU version under performs, but tiling the fused code gets back the lost performance as shown in Table 8.24.

TABLE 8.26. TCE Untiled-In-Core (secs) for V=100, O+V=120

Type	Sequential	OpenMP	SeqMKL	MT-MKL	GPU
Unfused	50.08	55.97	21.94	10.12	4.09
Fused	90.71	166.95	55.78	46.21	92.5

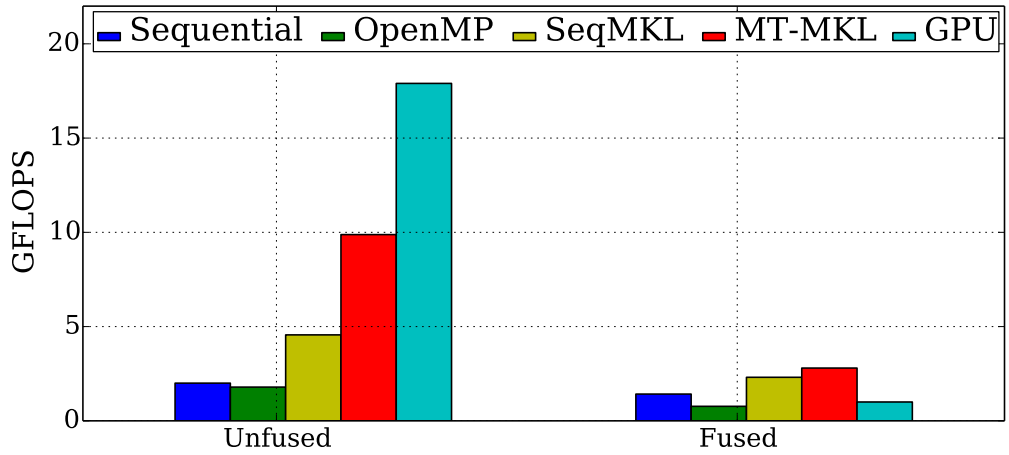


FIGURE 8.13. TCE Untiled-In-Core (FLOPS) for V=100, O+V=120

CPU out-of-core measurements are discussed now. Maximum memory used by:

- Unfused code: 13.04GB (Exceeds memory limit)
- Fused code: 10.43GB (Stays within memory limit)

- Fused Tiled code: 10.96GB (Stays within memory limit)

Fused-tiled code refers to *fused* code where fused loops are tiled. The innermost tiling loops are replaced with index permutation and BLAS calls.

As shown in Table. 8.27 and Fig. 8.14, *fused* code shows an improvement over unfused code by a factor of 3X. *Fused-tiled* code (for tile size 28) has a speedup of 15X (Sequential) and 27X (OpenMP) over *unfused* code and a speedup of 4.8X (Sequential) and 9.7X (OpenMP) over *fused* code. Since the *unfused* code taps into virtual memory, the overall performance decreases.

TABLE 8.27. CPU Out-Of-Core (min) for V=120, O+V=180

Type	Unfused	Fused	Fused (T=28)
SeqMKL	27.85	8.74	1.8
MT-MKL	24.98	8.78	0.9

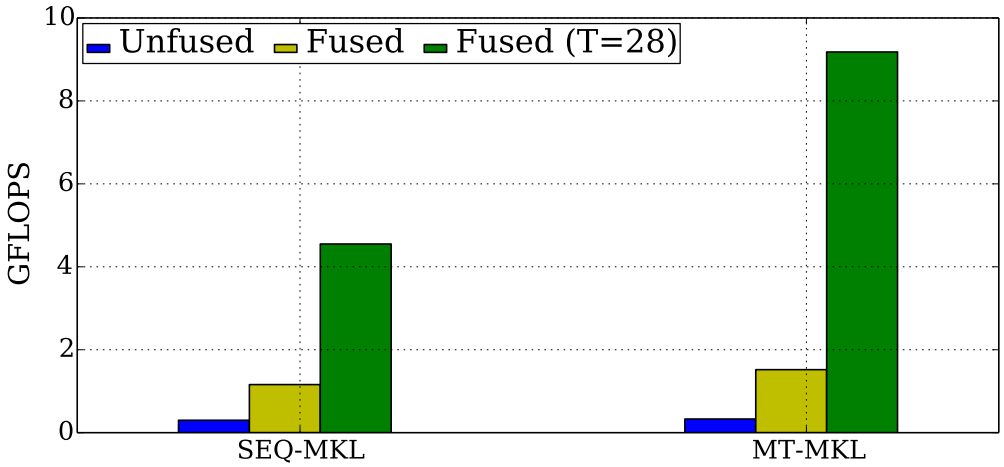


FIGURE 8.14. CPU Out-Of-Core (FLOPS) for V=120, O+V=180

We now discuss the measurements of the GPU code using the sizes *O=100,O+V=120*. The maximum memory used by *unfused* code is 2.83GB, *fused* code is 1.54GB, and *fused-tiled* code is 2.6GB. TCE optimized GPU code uses CUDA BLAS and GPU index permutation routines from the Accelereyes ArrayFire library. This code is compared with PPCG generated CUDA code. The input to PPCG is baseline *unfused*, *fused* code containing classic loop nests with a contraction inside. PPCG generates tiled CUDA code as output. The tile size used here is 32. Fig. 8.28 shows the difference in performance between TCE optimized GPU code and PPCG generated CUDA code. The code generated by PPCG appears to be too complex to insert any index permutation and BLAS calls. The reason the PPCG unfused case runs out of memory is

TABLE 8.28. Comparison with PPCG on GPU

O+V	V	Unfused-TCE	Fused-TCE	Unfused-PPCG	Fused-PPCG
80	70	0.82sec	18.88sec	3.6min	2.36hours
120	100	4.1sec	92.55sec	Out-of-memory	>2hours

because PPCG does not handle allocate, deallocate statements (if you try to manage memory). Pluto also has the same issue.

We now discuss the GPU out-of-core measurements. The GPU memory on the C2075 GPU is advertised as 6GB, but the usable memory is 5.2GB. The tile size used for *fused* code is 28. Memory usage for *unfused* code is 5.34GB, *fused* code is 3.3GB and *fused-tiled* code is 4.66GB. Since the *unfused* code does not fit in GPU memory, it is tiled using the SciGPUGEMM [SCI] library. The library provides wrapper APIs for level 3 BLAS operations which automatically ship blocks of tensors between CPU, GPU memory when the input or output tensors in a contraction do not fit in GPU memory. The table in Fig. 8.29 below shows that the speed up of fused tiled code over unfused is 3X.

TABLE 8.29. Performance of GPU Out-Of-Core Code

O+V	V	Unfused-TCE Tiled	Fused-TCE Tiled
145	90	30.32sec	11.78sec

8.3 Evaluation of the Loop Fusion Optimization for GPGPUs

Using the same experimental setup shown in Figure. 8.8, we performed micro-benchmarks of the contraction of a two-dimensional tensor with a 4-dimensional tensor for all dimension sizes ranging from 5 to 120 in steps of 5 with four different layouts of the four-dimensional tensor. We also performed measurements of binary addition, index permutation, and transfer between GPU and CPU memory for the four-dimensional tensors of these sizes and layouts.

Our micro-benchmarks showed:

- On the GPU, addition is 3–5 times faster than DGEMM.
- Addition on the GPU is 2–4 times faster than addition on the CPU.
- On the GPU, index permutation is 3–7 times faster than DGEMM for tensor sizes up to approximately 1.3GB. Similarly, for these sizes, index permutation on the GPU is 4–7 times faster than index permutation on the CPU. For these sizes, addition and index permutation performance on the GPU are almost identical.

- Index permutation on the GPU for tensor sizes of more than 1.3GB is 1.5–2 times slower than that on the CPU and also 1.5–2 times slower than DGEMM on the GPU.

- For tensor sizes of less than approximately 400kB, sequential DGEMM on the CPU is faster than multi-core DGEMM on the CPU and than DGEMM on the GPU.

- For larger tensors, DGEMM on the GPU is 2–3 times faster than multi-core DGEMM and 6–8 times faster than sequential DGEMM on the CPU.

For very small contractions, with dimension sizes of less than 15, the performance of sequential DGEMM on the CPU is faster than DGEMM on the GPU. Therefore, we perform such contractions on the CPU and avoid the data movement costs between CPU and GPU. For larger tensors, we perform the entire computation on the GPU provided all the data fits into GPU memory.

For the tensor expression $(AB + CD + EF)$, we implemented both the *unfused* and *fused* plus *tiled* (where the tile size used is 20) variants, each of which were further measured under the following scenarios:

1. The entire computation is performed on the GPU.

2. Only one contraction fits in GPU memory at a time and the addition is performed on the GPU.

3. Only one contraction fits in GPU memory at a time and the addition is performed on the CPU.

Our measurements in Table 8.30 show that with fusion the computation time increases under any scenario and that fusion is not beneficial compared to performing the addition on the CPU. We have also demonstrated the effectiveness of our fusion algorithm for the case where the unfused version is too large to fit in GPU memory using the 4-index transformation equation. Table 8.29 shows a factor of three speed up of the fused-tiled code over unfused tiled code.

TABLE 8.30. Performance of the tensor expression $AB + CD + EF$

Scenario	Unfused	Fused-tiled
1	5.99sec	15.09sec
2	11.64sec	34.27sec
3	10.2sec	31.53sec

Our measurements show:

- It is not beneficial to use fusion to allow additions to be performed on the GPU, since the reduction in locality in the contractions is more expensive than the cost of transferring the results of a contraction to the CPU and performing addition there.
- When already using fusion and tiling, it is not beneficial to reduce the tile size further in order to fit addition on the GPU for the same reasons.
- If the result of an addition is needed on the GPU for another contraction, e.g., for $(AB + CD)E$, it is better to perform the addition on the GPU.
- If all the computation fits on the GPU, it is not beneficial to transfer intermediates to the CPU in order to overlap addition on the CPU with contractions on the GPU, since the transfer cannot be sufficiently overlapped with the contraction on the GPU.
- Similar to addition, index permutation is usually faster on the GPU. Except for very large tensors, index permutation should be performed where the result is needed. For very large tensors, index permutation is faster on the CPU.

Chapter 9
Conclusions and Future Directions

This dissertation addresses certain performance optimization issues for complex tensor contraction expressions that arise in quantum chemistry. We have made significant performance improvements to the loop fusion optimization algorithm that allow large tensor contraction equations to be optimized with complex (2-dimensional) loop fusion cost models without relying on optimization heuristics. We have developed a loop fusion cost model for memory minimization with dynamic memory allocation that calculates memory usage precisely. However, we found that for the tensor contraction equations encountered in quantum chemistry, the additional precision in the memory usage calculation does not warrant the higher computational cost. We also have developed a loop fusion optimization algorithm that can be applied to simple handwritten tensor contraction code as an alternative to the loop fusion algorithm for expression trees representing tensor contraction equations. This would allow translating, for example, handwritten in-core tensor contraction code into out-of-core code.

Finally, we have described an optimization framework for generating GPGPU code from tensor contraction equations. Our overall framework employs model-driven search-based algorithms for enumerating fused loop structures, tile sizes, and choices of index permutation and matrix multiplication library (BLAS) calls. Depending on the size of the tensors, the optimization framework decides whether the entire computation should be performed on the CPU or the GPU, whether tensor contractions should be performed on the GPU while additions are performed by the CPU, and whether loops should be fused. While we do not have these algorithms implemented yet, our measurements demonstrate that the choices for our loop fusion cost model would result in the enumeration of all important loop structures. The tiling and layout optimization algorithms can then select the optimal loop structure out of the candidates generated by the fusion algorithm.

9.1 Future Directions

The goals for the TCE are to be both a platform for experimenting with novel optimization algorithms and to be used by quantum chemists for generating efficient simulation models. To achieve these goals, however, there are several remaining software development and research tasks:

- **Implementation of GPU Cost Model for Loop Fusion** Our GPU cost model for the loop fusion algorithm needs to be refined, implemented, and tested. This also requires support in the fusion tree data structure and in SAGE tree generation to allow CUDA code to be generated.

- **Layout Optimization and Library Call Selection** Since the layout of intermediates is typically not constrained, layout optimization finds the layouts that minimizes the cost of DGEMM and index permutation calls as well as communication calls. This algorithm needs to be reimplemented to operate on ROSE abstract syntax trees. It also depends on new extensive measurements of the performance of any library calls and the communication cost for different types of parallelism and distributed matrix multiplication. Finally, the abstract syntax tree must be transformed to replace inner loop nests with the appropriate DGEMM and index permutation calls.

- **Detailed GPU Cost Models** We need detailed GPU cost models for the tiling and layout optimization algorithms. As an alternative, it is worth exploring a combination of tiling and layout optimization into a single traversal of the code. One possible enhancement to our framework is suggested by the approach of DePrince and Hammond [DH11]. For a single equation our approach should subsume or improve on their solution. For example, the iterative part of CCSD(T) consists of three equations. By mapping contractions of size N^4 from the smaller equations (in addition to the N^4 cost additions) onto the CPU, they improve the utilization of the CPU. A more general approach could select a small number of terms of a larger equation to perform on the CPU while the GPU computes the rest of the equation. This assignment of contractions to the CPU instead of the GPU could be performed as part of tiling optimization.

- **Symmetries and Block-Sparse Tensors** In coupled cluster chemistry models, tensors are frequently block sparse with large irregular block sizes because of the spatial symmetry of molecules and may also exhibit permutational symmetry, in which some slices of a higher-dimensional tensor are anti-symmetric (e.g., $T_{ij}^{ab} = -T_{ji}^{ab}$). In Atomic Orbital (AO) representations, tensors are block-sparse

with a small uniform block size. To enable research on cost models for symmetric and block-sparse tensors, we would need support for representing these symmetries and for generating appropriate code. We already have syntactic support in our tensor expression language for declaring symmetry and storage properties, but we would also need a type system for computing the symmetry properties of intermediates from their subtrees, support for symmetries in the cost models for our optimization algorithms, and tree transformations for generating the appropriate code.

- **Polyhedral Model Optimization Support** Pluto transforms C programs for generating OpenMP parallel code for multi-cores. The current implementation performs these transformation directly without generating abstract syntax trees as intermediate data structures. The Pluto port to ROSE is redesigned to work on abstract syntax trees. While our measurements demonstrated that for dense tensors our optimization framework out-performs Pluto, for tensors with symmetries polyhedral model optimization will likely be more competitive or preferable. Domain-specific optimizations in conjunction with Pluto may result in the best performance. Pluto can be incorporated with our search-based optimizations by providing search algorithms that iterate over tile sizes or other parameters and by letting Pluto find the optimal transformations for the given parameters. We might also need analyses for extracting the loop nests from a larger fused loop structure that will then be optimized by Pluto.

- **Code Generation for Parallelism** Similarly as for generating distributed code, we need an internal representation for describing how an individual contraction is to be parallelized. A tree transformation algorithm interprets this data structure and generates the appropriate parallel code for the desired form of parallelism (multiple processors, cores, and/or GPU). Later, the optimization algorithms can then either generate this representation or make the appropriate calls to the code transformations directly. Ideally, this internal representation and the code transformation would be general enough to allow for non-SPMD execution as well. E.g., for spatial symmetry it would be beneficial if different (groups of) processors perform the contractions for different spatial symmetry blocks. However, a non-SPMD execution model requires dynamic load balancing (so that processors and key system resources are not wasted due to idleness) and leads to complex synchronization issues.

- **Code Generation for Multi-Level Parallel Systems with Distributed Memory** We plan to handle code generation for a cluster of multi-core/SMP nodes that communicate via message-passing using

MPI or through the use of Global Arrays. The processors in each SMP node use the shared memory in the SMP node. Our approach to generating code for multi-level parallel systems is to view it as multi-level tiling, where the inner-level tiles of an outer-level tile (executing on different cores in a physical shared-memory domain) can share data directly, but direct sharing of data is not feasible between outer-level tiles (mapped to different SMP nodes on a cluster). We will use the Pluto framework to find a communication minimizing set of affine transformations (or, equivalently, tiling hyperplanes) for each statement to minimize the volume of communication between tiles and also to improve data reuse in each tile. The Pluto framework finds bands of (one or more) permutable loops that can be tiled and also identifies points in execution where synchronization is required. Determining the best choice of combinations of fusion and tiling structures and sets of tile sizes at different levels of tiling is a key problem. We plan to use a combination of model-driven and empirical search for this, which has been developed in [GKS+07]. For this, we will develop an infrastructure to create a characterization, using empirical evaluation, of the performance variation within each multicore/SMP node as a function of tile sizes. Using this characterization and an outer-level tiling of the computation's iteration space, the completion time on the multi-level parallel system can be modeled using the techniques in [GKS+07]. This can be used to guide the search. A critical transformation to optimize loop codes is loop fusion in conjunction with tiling. Loop fusion improves reuse across the different statements, while tiling improves reuse between iterations of the same statement. We plan to implement an enumeration-based approach to differentiate between the different fusion structures. In addition, we plan to develop, using the compiler framework, methods to determine the local data needed for data accessed in the tile (as a function of tile sizes) and to generate code to move data between SMP nodes (e.g., as in [CG06]).

Bibliography

[ACC] Accelereyes Arrayfire Library. http://www.accelereyes.com/arrayfire/c.

[AMP01] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *Int. J. Parallel Program.*, 29:493–544, October 2001.

[AO06] T.J. Ashby and M.F.P. O'Boyle. Iterative collective loop fusion. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 202–216. Springer Berlin / Heidelberg, 2006. 10.1007/11688839_17.

[AS87] Andrew W. Appel and Kenneth J. Supowit. Generalizations of the Sethi-Ullman algorithm for register allocation. *Software – Practice and Experience*, 17(6):417–421, June 1987.

[BAB+05a] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, February 2005.

[BAB+05b] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276 –292, February 2005.

[BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[BBA05] Bob Blainey, Christopher Barton, and José Amaral. Removing impediments to loop fusion through code transformations. In Bill Pugh and Chau-Wen Tseng, editors, *Languages and Compilers for Parallel Computing*, volume 2481 of *Lecture Notes in Computer Science*, pages 309–328. Springer Berlin / Heidelberg, 2005. 10.1007/11596110_21.

[BBC+02] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proc. of Supercomputing 2002*, November 2002.

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proc. CC 2008 - International Conference on Compiler Construction*, pages 132–146. LNCS, Springer-Verlag, 2008.

[BHH+10] Muthu Baskaran, Albert Hartono, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[BHRS08a] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[BHRS08b] Uday Bondhugula, Albert Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, 2008.

[Bib04] A. Bibireata. Memory-constrained data locality optimization for tensor contractions. Master's thesis, The Ohio State University, Columbus, Ohio, January 2004.

[BKC+03] A. Bibireata, S. Krishnan, D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Memory-constrained data locality optimization for tensor contractions. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing*, College Station, Texas, October 2003.

[BP10] Gergö Barany and Adrian Prantl. Source-level support for timing analysis. In *ISoLA (2)*, pages 434–448, 2010.

[BRS10] Muthu Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 244–263, 2010.

[BSC+00] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 39 –48, 2000.

[CBL] CUBLAS Library. https://developer.nvidia.com/cublas.

[CBL+02a] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam. Memory-constrained communication minimization for a class of array computations. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing*, volume 2481 of *Lecture Notes in Computer Science*, pages 1–15, College Park, Maryland, July 2002. Springer-Verlag.

[CBL+02b] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, June 2002.

[CE05] R. Choy and A. Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2):331 –341, February 2005.

[CG06] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7 pp.–, April 2006.

[CGK+03] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam. Global communication optimization for tensor contraction expressions under memory constraints. In *Proceedings of Seventeenth International Parallel and Distributed Processing Symposium (IPDPS '03)*, page 37b, Nice, France, April 2003. IEEE Computer Society Press.

[CGS98] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[CM95] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[CSV+07] T. D. Crawford, C. D. Sherrill, E. F. Valeev, J. T. Fermann, R. A. King, M. L. Leininger, S. T. Brown, C. L. Janssen, E. T. Seidl, J. P. Kenny, and W. D. Allen. PSI3: An open-source ab initio electronic structure package. *J. Comp. Chem.*, 28:1610–1616, 2007.

[CWB+01] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Towards automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization. In *Proceedings of the Intl. Conf. on High Performance Computing, Lecture Notes in Computer Science*, volume 2228, pages 237–248, Hyderabad, India, December 2001. Springer-Verlag.

[CWL+01] D. Cociorva, J. Wilkins, C. Lam, P. Sadayappan G. Baumgartner, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proc. of the Fifteenth ACM International Conference on Supercomputing (ICS'01)*, pages 500–509. ACM, 2001.

[Dar99] Alain Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 149–, Washington, DC, USA, 1999. IEEE Computer Society.

[DCHD90] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[DDE+05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293 –312, February 2005.

[DH11] A.E. DePrince and J.R. Hammond. Quantum chemical many-body theory on heterogeneous nodes. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 131–140, July 2011.

[DM98] L. Dagum and R. Menon. OpenMP: An industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[DRP96] Luiz De Rose and David Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 10th international conference on Supercomputing*, ICS '96, pages 309–316, New York, NY, USA, 1996. ACM.

[FF96] J. Foresman and A. Frisch. *Exploring Chemistry with Electronic Structure Methods: A Guide to Using Gaussian*. Gaussian, Inc., 2 edition, 1996.

[FHM99] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the 12th international symposium on System synthesis*, ISSS '99, pages 71–, Washington, DC, USA, 1999. IEEE Computer Society.

[FJ98] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381 –1384 vol.3, May 1998.

[FJ05] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, February 2005.

[FL91] C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, CA, 1991.

[GHS06] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, volume 4015 of *Lecture Notes in Computer Science*, pages 178–195. Springer Berlin / Heidelberg, 2006. 10.1007/11964681_11.

[GKS+07] X. Gao, S. Krishnamoorthy, S.K. Sahoo, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Efficient search-space pruning for integrated fusion and tiling transformations. *Concurrency and Computation: Practice and Experience*, 19(18):2425–2443, December 2007.

[GMM98] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 228–239, New York, NY, USA, 1998. ACM.

[GOST93] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, London, UK, 1993. Springer-Verlag.

[GSL+05] X. Gao, S.K. Sahoo, Q. Lu, G. Baumgartner, C. Lam, J. Ramanujam, and P. Sadayappan. Performance modeling and optimization of parallel out-of-core tensor contractions. In *Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming*, pages 266–276, Chicago, IL, June 2005.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 1–8, New York, NY, USA, 1978. ACM.

[HBB+09a] Albert Hartono, Muthu Manikandan Baskaran, C. Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ACM International Conference on Supercomputing*, 2009.

[HBB+09b] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[Hir03] So Hirata. Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[HLG+06] Albert Hartono, Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Marcel Nooijen, Gerald Baumgartner, David E. Bernholdt, Venkatesh Choppella, Russell M. Pitzer, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *International Conference on Computational Science*, pages 267–275, 2006.

[HLH+09] Albert Hartono, Qingda Lu, Thomas Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E. Bernholdt, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry*, 113(45):12715–12723, 2009.

[HSN+05] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. Bernholdt, S. Hirata, C. Lam, R. Pitzer, J. Ramanujam, and P. Sadayappan. Automated operation minimization of tensor contraction expressions in electronic structure calculations. In *International Conference on Computational Science*, volume 1, pages 155–164, 2005.

[HVP+06] Qubo Hu, Arnout Vandecappelle, Martin Palkovic, Per Gunnar Kjeldsberg, Erik Brockmeyer, and Francky Catthoor. Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 606–611, Piscataway, NJ, USA, 2006. IEEE Press.

[IBM] IBM XL C and C++ Compilers Family. http://www-03.ibm.com/software/products/en/ccompfami.

[ICC] Intel C/C++ Compiler. http://software.intel.com/en-us/intel-compilers.

[JJPX01] Jeremy Johnson, Robert W. Johnson, David A. Padua, and Jianxin Xiong. Searching for the best FFT formulas with the SPL Compiler. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 112–126, London, UK, 2001. Springer-Verlag.

[KAP97] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 346–357, New York, NY, USA, 1997. ACM.

[KBC+01] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61:1803–1826, 2001.

[KBC+05] K. Kennedy, B. Broom, A. Chauhan, R.J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387 –408, February 2005.

[KCA03] P.G. Kjeldsberg, F. Catthoor, and E.J. Aas. Data dependency size estimation for use in memory optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(7):908 – 921, july 2003.

[KKB+03] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, P. Sadayappan C. Lam, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Data locality optimization for synthesis of efficient out-of-core algorithms. In *Proc. of 10th Annual International Conference on High Performance Computing (HiPC)*, volume 2913 of *Lecture Notes in Computer Science*, pages 406–417, Hyderabad, India, December 2003. Springer-Verlag.

[KKB+04] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms for tensor contractions using a nonlinear optimization solver. In *The 18th International Parallel and Distributed Processing Symposium*, 2004.

[KKB+06] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms for tensor contractions using a nonlinear optimization solver. *Journal of Parallel and Distributed Computing*, 66(5):659–673, May 2006.

[KM93] Ken Kennedy and Kathryn S. Mckinley. Typed fusion with applications to parallel and sequential code generation. Technical report, Department of Computer Science, Rice University, 1993.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, 1994. Springer-Verlag.

[KPCM99] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 482–491, New York, NY, USA, 1999. ACM.

[Lam99] Chi-Chung Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, Ohio, August 1999. Also available as Technical Report No. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University, August 1999.

[LCBS99] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage requirement for a class of loops implementing multi-dimensional integrals. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, pages 350–364, San Diego, CA, 1999.

[LGK+12] Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *Journal of Parallel and Distributed Computing*, 72(3):338–352, 2012.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[Li93] Wei Li. *Compiling for NUMA parallel machines*. PhD thesis, Cornell University, Ithaca, NY, USA, 1993. UMI Order No. GAX94-06185.

[LKS06] Qingda Lu, Sriram Krishnamoorthy, and P. Sadayappan. Combining analytical and empirical approaches in tuning matrix transposition. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 233–242, New York, NY, USA, 2006. ACM.

[LLL01] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 103–112, New York, NY, USA, 2001. ACM.

[LRB+11] C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, July 2011.

[LRW91] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. *SIGOPS Oper. Syst. Rev.*, 25:63–74, April 1991.

[LSW97] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parall. Process. Lett.*, 7(2):157–168, 1997.

[LZR+12] Pai-Wei Lai, Huaijian Zhang, Samyam Rajbhandari, Edward Valeev, Karol Kowalski, and P. Sadayappan. Effective Utilization of Tensor Symmetry in Operation Optimization of Tensor Contraction Expressions. *Procedia Computer Science*, 9(0):412421, 2012. Proceedings of the International Conference on Computational Science, {ICCS} 2012.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Trans. Parallel Distrib. Syst.*, 8:193–209, February 1997.

[MCG04] P. Marchal, F. Catthoor, and J.I. Gomez. Optimizing the memory bandwidth with loop fusion. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 188 – 193, sept 2004.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18:424–453, July 1996.

[MHCF98] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *Int. J. Parallel Program.*, 26:641–670, December 1998.

[MJJ+00] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.

[MKA11] W. Ma, S. Krishnamoorthy, and G. Agrawal. Practical loop transformations for tensor contraction expressions on multi-level memory hierarchies. In *To be published in Proc. CC 2011 - International Conference on Compiler Construction*. LNCS, Springer-Verlag, 2011.

[MKL] Intel Math Kernel Library (Intel MKL). http://software.intel.com/en-us/intel-mkl.

[MKV+13] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. Optimizing tensor contraction expressions for hybrid CPU-GPU execution. *Cluster Computing*, 16(1):131–155, 2013.

[MP99a] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 53–65, New York, NY, USA, 1999. ACM.

[MP99b] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 434–443, New York, NY, USA, 1999. ACM.

[NVC] NVIDIA CUDA Compiler. https://developer.nvidia.com/cuda-llvm-compiler.

[NVD] Nvidia Tesla C2075. http://www.nvidia.com/object/workstation-solutions-tesla.html.

[PH02] Geoff Pike and Paul N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–12, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[PMHC03] Venkata K. Pingali, Sally A. Mckee, Wilson C. Hsieh, and John B. Carter. Restructuring computations for temporal data cache locality. *International Journal of Parallel Programming*, 31:2003, 2003.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Procweneedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[QK05] Apan Qasem and Ken Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In *In LCPC*, 2005.

[QK06] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 249–258, New York, NY, USA, 2006. ACM.

[QK08] Apan Qasem and Ken Kennedy. Model guided empirical tuning of loop fusion. *Int. J. High Perform. Syst. Archit.*, 1:183–198, December 2008.

[RHKN01] J. Ramanujam, Jinpyo Hong, Mahmut Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 359–364, New York, NY, USA, 2001. ACM.

[RHKN06] J. Ramanujam, Jinpyo Hong, Mahmut K, and A. Narayan. Estimating and reducing the memory requirements of signal processing codes for embedded systems, 2006.

[RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108 – 120, 1992.

[RSE] ROSE Compiler Framework. http://rosecompiler.org.

[RSM] R-Stream High-Level Compiler. https://www.reservoir.com/product/r-stream.

[RT99] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 168–182, London, UK, 1999. Springer-Verlag.

[SBB+93] M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, , and J. Montgomery. General atomic and molecular electronic structure system (gamess). *Journal on Computational Chemistry*, 14:1347–1363, 1993.

[SCFS98] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 24–33, New York, NY, USA, 1998. ACM.

[SCI] SciGPU-GEMM Library. https://code.google.com/p/scigpugemm.

[SGW+] J.F. Stanton, J. Gauss, J.D. Watts, M. Nooijen, N. Oliphant, S.A. Perera, P.G. Szalay, W.J. Lauderdale, S.A. Kucharski, S.R. Gwaltney, S. Beck, A. Balková, D.E. Bernholdt, K.K. Baeck, P. Rozyczko, H. Sekino, C. Hober, , and R.J. Bartlett. ACES II. Quantum Theory Project, University of Florida. Integral packages included are VMOL (J. Alml of and P.R. Taylor); VPROPS (P. Taylor) ABACUS; (T. Helgaker, H.J. Aa. Jensen, P. Jørgensen, J. Olsen, and P.R. Taylor).

[SL99] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 215–228, New York, NY, USA, 1999. ACM.

[SM96] Sharad Singhai and Kathryn Mckinley. Loop fusion for data locality and parallelism. In *In Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems, New Paltz*, pages 148–150, 1996.

[SM97] Sharad Singhai and Kathryn S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[Son00] Yonghong Song. *Compiler algorithms for efficient use of memory systems*. PhD thesis, Purdue University, West Lafayette, IN, USA, 2000. AAI3033175.

[SU70] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(1):715–728, October 1970.

[SWL03] Yonghong Song, Cheng Wang, and Zhiyuan Li. Locality enhancement by array contraction. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 132–146, Berlin, Heidelberg, 2003. Springer-Verlag.

[SXWL01] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 50–64, New York, NY, USA, 2001. ACM.

[VBG+10] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[VBJC03] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Multi-dimensional incremental loop fusion for data locality. In *In Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pages 17–27, 2003.

[VJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013. Selected for presentation at the HiPEAC 2013 Conf.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. *SC Conference*, 0:38, 1998.

[WKM+10] H.-J. Werner, P. J. Knowles, F. R. Manby, M. Schütz, P. Celani, G. Knizia, T. Korona, R. Lindh, A. Mitrushenkov, G. Rauhut, T. B. Adler, R. D. Amos, A. Bernhardsson, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, E. Goll, C. Hampel, A. Hesselmann, G. Hetzer, T. Hrenar, G. Jansen, C. Köppl, Y. Liu, A. W. Lloyd, R. A. Mata, A. J. May, S. J. McNicholas, W. Meyer, M. E. Mura, A. Nicklass, P. Palmieri, K. Pflüger, R. Pitzer, M. Reiher, T. Shiozaki, H. Stoll, A. J. Stone, R. Tarroni, T. Thorsteinsson, M. Wang, and A. Wolf. Molpro, version 2010.1, a package of ab initio programs, 2010. see http://www.molpro.net.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[WMC96] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[XJJP01] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: a language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM.

[Xue05] Jingling Xue. Aggressive loop fusion for improving locality and parallelism. In Yi Pan, Daoxu Chen, Minyi Guo, Jiannong Cao, and Jack Dongarra, editors, *Parallel and Distributed Processing and Applications*, volume 3758 of *Lecture Notes in Computer Science*, pages 224–238. Springer Berlin Heidelberg, 2005. 10.1007/11576235_28.

[YLR+03] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 63–76, New York, NY, USA, 2003. ACM.

[ZMS+04] YongKang Zhu, Grigorios Magklis, Michael L. Scott, Chen Ding, and David H. Albonesi. The energy impact of aggressive loop fusion. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 153–164, Washington, DC, USA, 2004. IEEE Computer Society.

[ZYK+05] Yuan Zhao, Qing Yi, Ken Kennedy, Dan Quinlan, and Richard Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical report, Lawrence Livermore National Laboratory, 2005.

Vita

Ajay Panyala was born in Hyderabad, India, in 1986. He obtained his bachelor's degree in Computer Science and Engineering in 2007 from Jawaharlal Nehru Technological University (JNTU), Hyderabad. He entered the Master's program at Lousiana State University in Fall 2007 and switched to the Doctoral program in Spring 2008. His research interest falls in the area of Compiler Optimizations for High Performance Computing.