Efficient Search-Space Pruning for Integrated Fusion and Tiling Transformations

Xiaoyang Gao[1], Sriram Krishnamoorthy[1], Swarup Kumar Sahoo[1], Chi-Chung Lam[1], Gerald Baumgartner[2], J. Ramanujam[3], and P. Sadayappan[1]

[1] Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA
{gaox,krishnsr,sahoo,clam,saday}@cse.ohio-state.edu
[2] Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803, USA
gb@csc.lsu.edu
[3] Department of Electrical and Computer Engineering and
Center for Computation and Technology
Louisiana State University, Baton Rouge, LA 70803, USA
jxr@ece.lsu.edu

Abstract. Compile-time optimizations involve a number of transformations such as loop permutation, fusion, tiling, array contraction, etc. Determination of the choice of these transformations that minimizes the execution time is a challenging task. We address this problem in the context of tensor contraction expressions involving arrays too large to fit in main memory. Domain-specific features of the computation are exploited to develop an integrated framework that facilitates the exploration of the entire search space of optimizations. In this paper, we discuss the exploration of the space of loop fusion and tiling transformations in order to minimize the disk I/O cost. These two transformations are integrated and pruning strategies are presented that significantly reduce the number of loop structures to be evaluated for subsequent transformations. The evaluation of the framework using representative contraction expressions from quantum chemistry shows a dramatic reduction in the size of the search space using the strategies presented.

1 Introduction

Optimizing compilers incorporate a number of loop transformations such as permutation, tiling, fusion, etc. Considerable work has been done on improving locality and/or parallelism by loop fusion [8–11, 18]. Fusion often creates imperfectly nested loops, which are more complex to tile effectively than perfectly nested loops. Several works have addressed the tiling of imperfectly nested loops [2, 19]. Although there has been much progress in developing unified frameworks for modeling a variety of loop transformations [1, 2, 15], their use has so far been restricted to optimization of indirect performance metrics such as reuse distance, degree of parallelism, etc.

The development of model-driven optimization strategies that target direct performance metrics, remains a difficult task. In this paper, we address the problem in the specific domain of tensor contractions involving tensors too large to fit into physical memory. We use certain properties of the computations in this domain to integrate various transformations and investigate pruning strategies to reduce the search space to be explored.

The large sizes of the tensors involved require the development of *out-of-core* implementations that orchestrate the movement of data between disk and main memory. In this paper, we discuss the integration of loop fusion and tiling transformations with the objective of minimizing disk I/O cost. We first divide the input program into several independent loop nests, then enumerate the set of fusion structures of each loop nest. A generalized tiling approach is presented that significantly reduces the number of loop structures to be explored. It also enables subsequent optimizations of I/O placements and loop permutations. This approach enables an exploration of the entire search space using a realistic performance model, without the need to resort to heuristics and search of a limited subspace of the search space to limit search time.

The rest of this paper is organized as follows. In the next section, we elaborate on the computational context of interest and introduce some preliminary concepts; in addition, an overview of the program synthesis system and the overall approach are given. Section 3 describes a tree partitioning algorithm. In Section 4, we propose a loop structure enumeration algorithm and prove its completeness. The reductions in the space of loop structures to be explored is shown for representative computations in Section 5. Conclusions are provided in Section 6.

2 Background

The work presented in this paper is being developed in the context of the Tensor Contraction Engine (TCE) program synthesis tool [3–6, 13]. The TCE takes as input a high-level specification of a computation expressed as a set of tensor contraction expressions, and transforms it into efficient parallel code. The current prototype of the TCE incorporates several compile-time optimizations which are treated in a decoupled manner, with the transformations being performed in a pre-determined sequence. In [12], we presented an integrated approach to determine the tile sizes and I/O placements for a fixed loop structure. Techniques to prune the search space of possible I/O placements, orderings, loop permutations and tiling for given a choice of fusion of tensor contractions were presented in [17]. In this paper, we present a technique to enumerate the various fusion structures and develop an algorithm to significantly reduce the number of loop nests to be evaluated for each fusion structure.

2.1 Computational Context

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

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

Here, all arrays would be initially stored on disk. The indices p, q, r, and s have the same range N. The indices a, b, c, and d have the same range V. Typical values for N range from 60 to 1300; the value for V is usually between 50 and 1000.

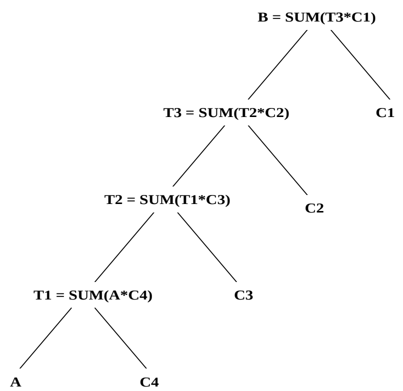
The calculation of B is done in four steps to reduce the number of floating point operations,

$$T1(a,q,r,s) = \sum_{p} C4(a,p) \times A(p,q,r,s)$$
$$T2(a,b,r,s) = \sum_{q} C3(b,q) \times T1(a,q,r,s)$$
$$T3(a,b,c,s) = \sum_{r} C2(c,r) \times T2(a,b,r,s)$$
$$B(a,b,c,d) = \sum_{s} C1(d,s) \times T3(a,b,c,s)$$

The sequence of contractions in this form can be represented by a operation tree, as shown in Fig. 1(a). The leaves correspond to the input arrays and the root corresponds to the output array. The intermediate arrays and output array are produced by the tensor contraction of their immediate children. The edges in the operation tree represent the *producer-consumer* relationship between contractions.

Assuming that the available memory space is less than V^4 (which is 3TB for $V = 800$), any of the logical arrays A, $T1$, $T2$, $T3$, and B is too large to entirely fit in memory. Therefore, if the computation is implemented as a succession of four independent steps, the intermediates $T1$, $T2$, and $T3$ have to be written to disk after they are produced, and read from disk before they are used in the next step. Furthermore, the amount of disk access volume could be much larger than the total volume of the data on disk. Since none of these arrays can be fully stored in memory, it is not possible to read each element only once from disk. Suitable fusion of the common loops between producing and consuming contractions can reduce the size of the intermediate array, making it feasible to retain it in memory. Henceforth, the term intermediate node will be used to refer to both the intermediate array produced in the corresponding interior node, and the contraction that produces it. The reference shall be clear from the context.

Given a choice of fusion, an intermediate node not fused with its parent divides the operation tree into two parts, both of which can be evaluated independently. Such an intermediate node is called a *cut-point*. A cut-point node is



(a) Operation tree for the four-index transform

for a,r,q,s,p
$\big[\, t1_{a,q,r,s} \mathrel{+}= A_{p,q,r,s} * C4_{a,p}$

for a,b,r,s,q
$\big[\, t2_{a,b,r,s} \mathrel{+}= t1_{a,q,r,s} * C3_{b,q}$

for a,b,c,r,s
$\big[\, t3_{a,b,c,s} \mathrel{+}= t2_{a,b,r,s} * C2_{c,r}$

for a,b,c,d,s
$\big[\, B_{a,b,c,d} \mathrel{+}= t3_{a,b,c,s} * C1_{d,s}$

(b) Corresponding unfused code structure

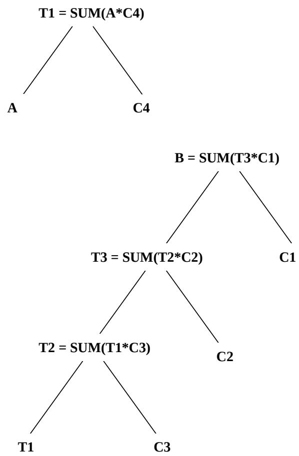
Fig. 1. Operation tree and unfused code structure for the four-index transform

assumed to be resident on disk. A connected operation tree without any interior cut-points is called a *fused sub-tree*. The divided operation tree for the four-index transform corresponding to $T1$ being a cut-point is shown in Fig. 2(a). The *loop nesting tree* (LNT) represents the loop structure of a fused sub-tree. Each node in a LNT is labeled by the indices of a set of fully permutable loops appearing together at the same level in the imperfectly nested loop structure. Loops in the children nodes are surrounded by loops in the parent node. Fig. 2(b) shows two possible LNT's for the two fused subtrees in Fig. 2(a), respectively. The corresponding code structure is shown in Fig. 2(c).

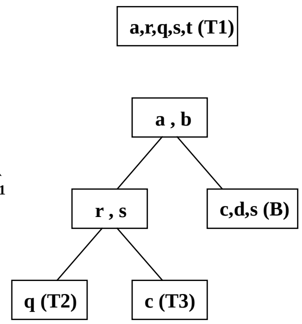
2.2 Overall approach

The program synthesis system takes an operation tree representing a set of tensor contractions as input, and generates an efficient loop structure with explicit disk I/O statements to implement the computation. The optimization process may be viewed in terms of the following steps:

1. Operation Tree Partitioning: In this step, we divide the original operation tree into several fused subtrees by identifying cut-points. The optimal loop structures for the subtrees are independent of each other, and are determined separately.
2. Loop Structures Enumeration: For each fused subtree, we enumerate candidate loop structures to be evaluated, as a set of LNT's.
3. Intra-Tile Loop Placements: For a given LNT, we tile all loops at each node and propagate intra-tile loops to all the nodes below it.
4. Disk I/O Placements and Orderings: We then explore various possible placements and orderings of disk I/O statements for each disk array in a tiled loop



(a) Divided operation trees



(b) Loop nesting trees

```
loopNest1 :
for a,r,q,s,p
[ t1[a,q,r,s] += A[p,q,r,s] * C4[a,p]

loopNest2 :
for a,b
[ for r,s
  [ for q
    [ t2[r,s] += t1[a,q,r,s] * C3[b,q]
    for c
    [ t3[c,s] += t2[r,s] * C2[c,r]
  for c,d,s
  [ B[a,b,c,d] += t3[c,s] * C1[d,s]
```

(c) Corresponding code structure

Fig. 2. Representations involved in generation of a fused code structure.

structure with a pruning strategy to determine the best placement and ordering.

5. Tile Size Selection: For each combination of loop transformations and I/O placements, the I/O cost is formulated as a non-linear optimization problem in terms of the tile sizes. The tile sizes that minimize the disk I/O cost are determined using a general-purpose non-linear optimization solver.
6. Code Generation: We calculate the disk access cost for each solution obtained, and generate code for the one with the minimal disk I/O cost.

The possible choices of fused subtrees are first enumerated. This is explained in Section 3. Given a fused sub-tree, the optimal loop structure and the corresponding cost can be determined by the following steps: 1) enumerating all candidate loop structures; 2) enumerating placements and ordering of disk I/O statments; 3) determining the tile size to minimize the disk I/O cost for each combination; 4) selecting the program structure with the minimal disk I/O cost. The algorithm for enumerating candidate loop structures is discussed in Section 4. The search space of disk I/O placements and orderings, loop permutations and tile sizes is pruned and modeled as a non-linear optimization problem in [17], which is then solved to determine the disk I/O cost. In this paper, we focus on determination of the fused sub-trees and the enumeration of candidate loop nesting trees to be evaluated.

3 Tree Partitioning

In this section, we discuss the procedure to enumerate the set of all fused subtrees to be evaluated. In general, fusing a loop between the producer of an intermediate array and its consumer eliminates the corresponding dimension of the array and reduces the array size. If the array fits in memory after fusion, no disk I/O is required for that array. On the other hand, if the array does not fit in the physical memory after fusion, the disk I/O cost will remain the same and there is not improvement in locality. Therefore, fusion of any loops corresponding to an intermediate node is assumed to cause the resulting intermediate to reside in memory. Alternatively, an intermediate node not fused with its parent (*cut-point*) is assumed to reside in disk.

An arbitrary operation tree with M intermediate nodes theoretically has $O(2^M)$ possible fused sub-trees, but not all of them are legal. If both the children of an intermediate node are fused with it, then the loops corresponding to the summation indices in the given node must be the outermost loops; and it can not be fused with its parent anymore. Thus, either the node itself or one of its children must be a cut-point.

Based on this property, we can restrict the number of top sub-trees to $O(M^2)$. The algorithm to enumerate the fused sub-trees rooted at a given node is shown in Algorithm 1. It proceeds in a bottom-up fashion, constructing all fused sub-trees rooted at a given node from those of its children. Given a node t with two children $left$ and $right$, we can extend a fused sub-tree from either $left$ or $right$ to include the given node. These sub-trees can further be extended to include the given node's parent. Besides, the given node can be considered as a cut-point. In this scenario, all possible pairs of left and right fused sub-trees may form a valid fused sub-tree for the given node. The field *t.TreeSet* represents the set of fused sub-trees which can be extended to include the parent of t.

4 Loop Structure Enumeration

In this section, we first present an algorithm that can generate a set of loop structures for a fused sub-tree. Then, we present the result that for any loop structure S of the fused sub-tree, we can find a corresponding loop structure S' in the generated set, so that S' can be transformed to S by use of a multi-level tiling strategy.

4.1 Enumeration Algorithm

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

- All contractions form a chain, called a *contraction chain*. For instance, Fig. 1 is such an operation tree, in which the contraction chain is $T1, T2, T3, B$.
- The contractions form two chains joining at the root node. In this case, the *contraction chain* is connected by these two chains. An example of such an operation tree is shown in Fig. 3, in which the contraction chain is $T1, T2, B, T3, T4$

Algorithm 1 EnumerateFusedSubtrees(t: the root of a subtree) returns $TreeSet$

```
t1 = the left child of t;    t2 = the right child of t;    TreeSet = empty
//Only one subtree
if both t1 and t2 are input nodes then
   Create a new Tree Tr with Tr.CutpointSet = ∅
   Insert Tr into TreeSet
end if
//Extending subtrees from the child not an input
if t1 is an input node and t2 is an intermediate node then
   childSet = t2.TreeSet
   Create a new Tree Tr with Tr.CutpointSet = {t2}
   Insert Tr into TreeSet
end if
if t2 is an input node, and t1 is an intermediate node then
   childSet = t1.TreeSet
   Create a new Tree Tr with Tr.CutpointSet = {t1}
   Insert Tr into TreeSet
end if
for each subtree st in childSet do
   Create a new Tree Tr with Tr.CutpointSet = st.CutpointSet
   Insert Tr into TreeSet
end for
t.TreeSet = TreeSet
//Entending subtrees from either child, and cutting another child off
if both t and t2 are intermediate nodes then
   childSet1 = t1.TreeSet
   for each subtree st in childSet1 do
      Create a new Tree Tr with Tr.CutpointSet = {st.CutpointSet, t2}
      Insert Tr into TreeSet
   end for
   childSet2 = t2.TreeSet
   for each subtree st in childSet2 do
      Create a new Tree Tr with Tr.CutpointSet = {st.CutpointSet, t1}
      Insert Tr into TreeSet
   end for
   Create a new Tree Tr with Tr.CutpointSet = {t1, t2}
   Insert Tr into TreeSet
   t.TreeSet = TreeSet
   //Merging subtrees from both children, and extending the result
   for each pair of subtrees st1 in childSet1 and st2 in childSet2 do
      Create a new Tree Tr
      Tr.CutpointSet = {st1.CutpointSet, st2.CutpointSet}
      Insert Tr into TreeSet
   end for
end if
```

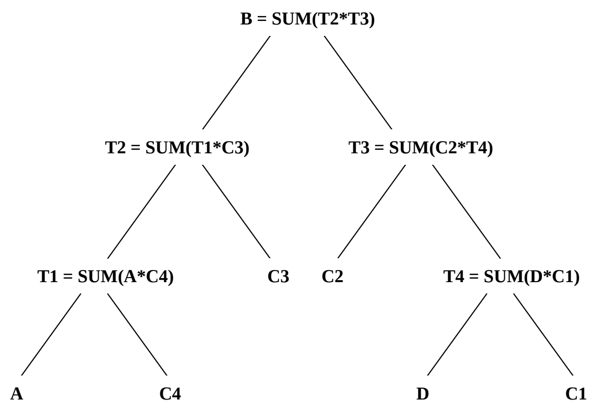


Fig. 3. An operation tree with two chains

Given an operation tree that has n contraction nodes $t_1, t_2, ..., t_n$, let $t_i.indices$ denote all loop indices surrounding the contraction node t_i. First, we create a contraction chain of the operation tree. It corresponds to a sequence of perfectly nested loops. Many different choices exist for the ordering of the fusions within this sequence of perfectly nested loop nests. Each of the perfectly nested loops corresponding to a contraction can be considered an independent loop nesting tree. The fusion of sub-trees producing and consuming an intermediate array creates an imperfectly nested loop nest, in which some of the common loops are merged. The process of construction of the loop nesting trees corresponding to a fused sub-tree can be modeled as a paranthesization problem. Consider the sequence of contraction nodes T1, T2, T3, and B in the operation tree shown in Fig. 1. $((T1(T2\,T3))B)$ corresponds to a parenthesization in which the contractions producing T3 and consuming T3 are fused first and the resulting loop nest is fused with the contractions producing T1 and B, in that order. Fig. 4 shows one possible parenthesization for the four-index transform and the corresponding loop nesting tree.

We enumerate all possible parenthesizations of the contraction chain. For each parenthesization, a maximally fused loop structure is created by a recursive construction procedure. We call it *maximally fused* since, in the construction procedure, each intermediate node will have its indices fused as much as possible with its parent. The construction procedure is shown in Algorithm 2. It takes a parenthesization P as input, and generate a corresponding LNT. A parenthesization of a contraction chain with n nodes has $n-1$ pairs of parentheses. Each pair of parentheses includes two elements, left and right element. Each element is either a single contraction node, or a parenthesization of a sub-chain within a pair of parentheses.

Fig. 4 illustrates this proceduce for the $((T1(T2\,T3))B)$ parenthesization of the four-index transform.

Algorithm 2 Construction(P)

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

$l = P.left$
$r = P.right$
if l is a parenthesization **then**
  lt = Construction($left$)
else if l is a contraction **then**
  lt = Create a new LNT node
  $lt.indices = l.indices$
  $lt.children = null$
  $lt.contraction = l$ {lt is a leaf, which includes a contraction node in it}
end if
if r is a parenthesization **then**
  rt = Construction($right$)
else if r is a contraction **then**
  rt = Create a new LNT node
  $rt.indices = r.indices$
  $rt.children = null$
  $rt.contraction = r$ {rt is a leaf, which includes a contraction node in it}
end if
$comindices = lt.indices \cap rt.indices$
$lt.indices = lt.indices - comindices$
$rt.indices = rt.indices - comindices$
lnt = Create a new LNT node
$lnt.indices = comindices$
$lnt.children = \{lt, rt\}$
return lnt

4.2 Completeness

In this section, we state results that are useful in proving that the set of *maximally fused* loop structures generated by the enumeration algorithm above can represent all loop structures of a fused subtree.

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

1. Take lnt as input, and create a parenthesization P of the contraction chain using the generation routine provided in Algorithm 3.
2. Apply the construction procedure in Algorithm 2 on P to generate a maximally fused loop structure lnt'.

Obviously, lnt' is the set of *maximally fused* loop structures generated by the enumeration algorithm. We note that lnt' can be translated to lnt by sinking indices at upper levels down.

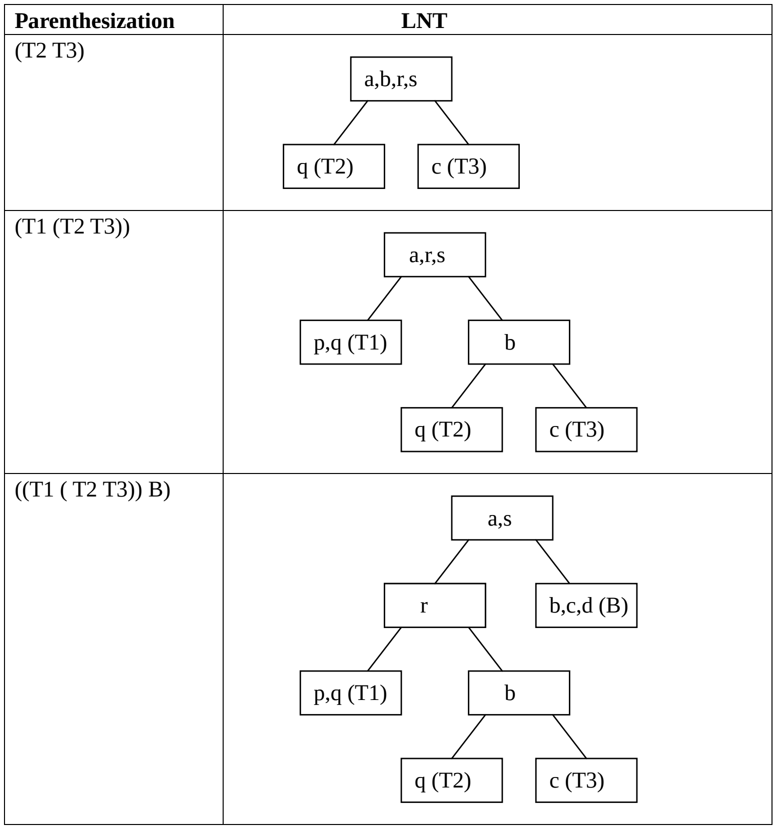


Fig. 4. Construction of a maximally fused loop structure for a particular parenthesization of the four-index transform.

Lemma 1. For any pair of contraction nodes t_i and t_j, let $common(lnt, t_i, t_j)$ be defined as the loops shared by t_i and t_j in lnt. We have $common(lnt, t_i, t_j) \subseteq common(lnt', t_i, t_j)$.

Lemma 2. If $common(lnt, t_i, t_j) \subset common(lnt', t_i, t_j)$, then we can transform lnt' to form lnt'' by sinking indices down, so that $common(lnt, t_i, t_j) = common(lnt'', t_i, t_j)$

Applying the sinking operation in Lemma 2 for each pair of contraction nodes $(t_i,\ t_j)$, we can transform lnt' to lnt'', which satisfies the condition: $\forall(t_i, t_j), common(lnt, t_i, t_j) = common(lnt'', t_i, t_j)$. After that, if a node r has no indices in $r.indices$, we remove r from lnt'', and put all children of r to its parent. Then, lnt'' is same as lnt.

Using a *multi-level tiling strategy*, a maximally fused loop strcuture can be transformed into an arbitrarily fused loop structure by appropriate choice of tile

Algorithm 3 Parenthesize(lnt)

//Given an LNT, the algorithm map it to a corresponding parenthesization

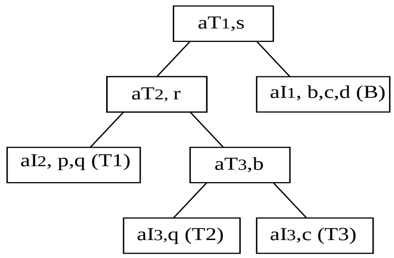
```
if lnt.children ≠ null then
  P = null
  for each child c in lnt.children do
    P' = Parenthesize(c)
    if P is null then
      P = P'
    else
      P = new Parenthesization(P, P')
    end if
  end for
else
  P = c.contraction {c is a leaf and includes a contraction node}
end if
return P
```

sizes. *Multi-level tiling* can transform the LNT of a loop structure as follows. Each loop present in the root is split into two components, an inter-tile loop and an intra-tile loop. The intra-tile loop is placed on child nodes of the root. Then the loops present at each of the child nodes, including the intra-tile loops from the root, are again split and intra-tile loops are placed on their respective child nodes. This process is porformed recursively until leaf nodes are encountered. The loop structure corresponding to the LNT can also be transformed accordingly. Fig. 5 shows the tiling of loop a in the LNT in Fig. 4 and the relationship between different tiles, where $a.range$ represents the range of loop a.

The sinking operation in an LNT can be modeled as a *multi-level tiling* in the loop structure. If we tile a fused loop with a tile size equal to its loop range, it leads to the same result as sinking the loop index from the original node to its children. Let S and S' be loop structures representd by lnt and lnt' respectively. Since we can transform lnt' to lnt by sinking operations, we can also transform S' to S by suitable multi-level tiling. We use an example to show the details of the transformation procedure below.

An arbitrary fully fused loop structure S for the four-index transform is shown in Fig. 6(a), and the corresponding maximally fused loop structure S' may be seen in Fig. 6(b). After we apply multi-level tiling, S' is translated to the form shown in Fig. 7(a). In addition, if we set ranges of inter-tile loops as shown below, and remove all loops with $range = 1$, S' can be rewritten in the form shown in Fig. 7(b), which is exactly the same as S. The indexing of the intermediate arrays has been shown in a generic fashion.

$$aT_2 = aT_3 = sT_1 = sT_2 = sT_3 = rT_2 = qT_1 = 1; aT_1 = a.range; rI_1 = r.range$$



(a) Multi-level tiling loop a

$$aT_1.range \times aI_1.range = a.range$$

$$aT_2.range \times aI_2.range = aI_1.range$$

$$aT_3.range \times aI_3.range = aI_2.range$$

(b) Range of different level tiles

Fig. 5. An example of multi-level tiling in LNT

```
for a
  for r
    for q,s,p
      t1[s,q] += A[p,q,r,s] * C4[a,p]
    for b,s,q
      t2[b,r,s] += t1[s,q] * C3[b,q]
  for b,c,r,s
    t3[b,c,s] += t2[b,r,s] * C2[c,r]
  for b,c,d,s
    B[a,b,c,d] += t3[b,c,s] * C1[d,s]
```

(a) Arbitrary fused loop structure: S

```
for a,s
  for r
    for q
      for p
        t1 += A[p,q,r,s] * C4[a,p]
      for b
        t2[b] += t1 * C3[b,q]
    for b,c
      t3[b,c] += t2[b] * C2[c,r]
  for b,c,d
    B[a,b,c,d] += t3[b,c] * C1[d,s]
```

(b) Maximally fused loop structure: S'

Fig. 6. An arbitrary loop structure and the corresponding maximally fused structure

4.3 Complexity

The total number of loop structures generated by the enumeration algorithm is the same as the number of parenthesizations of the contraction chain. For a contraction chain with n nodes, the number of all possible parenthesizations is called the n^{th} *Catalan Number*. It is exponential in n, and the upper bound is $O(4^n/n^{3/2})$. In contrast, the number of possible loop structures is potentially exponential in the total number of distinct loop indices in the n intermediate nodes, a considerably larger number. The fused operation tree is not very long for most representative computations. In most practical applications, a fused subtree usually has no more than 5 contractions in a single chain. Note that the n^{th} Catalan Number is not very large when n is small. The first six Catalan Numbers are listed here: $1, 1, 2, 5, 14, 42, \ldots$.

```
for aT1,sT1
  for rT1,aT2,sT2
    for qT1,rT2,aT3,sT3
      for p,qI1,rI2,aI3,sI3
        t1[aI,qI,rI,sI] += A[p,q,r,s] * C4[a,p]
      for b, qI1,rI2,aI3,sI3
        t2[aI,b,rI,sI] += t1[aI,qI,rI,sI] * C3[b,q]
    for b, c, rI1, aI2,sI2
      t3[aI,b,c,sI] += t2[aI,b,rI,sI] * C2[c,r]
  for aI1, b, c, d, sI1
    B[a,b,c,d] += t3[aI,b,c,sI] * C1[d,s]
```

(a) After inserting intra-tile loops

```
for aT1
  for rT1
    for p,qI1,sI3
      t1[aI,qI,rI,sI] += A[p,q,r,s] * C4[a,p]
    for b,qI1,sI3
      t2[aI,b,rI,sI] += t1[aI,qI,rI,sI] * C3[b,q]
  for b,c,rI1,aI2,sI2
    t3[aI,b,c,sI] += t2[aI,b,rI,sI] * C2[c,r]
  for b,c,d,sI1
    B[a,b,c,d] += t3[aI,b,c,sI] * C1[d,s]
```

(b) After selecting proper tile counts

Fig. 7. Translate S' to S by multi-level tiling strategy

5 Experimental Results

The enumeration algorithm discussed in Section 4.1 generates a set of candidates loop structures to be considered for data locality optimization. Without this algorithm, and generalized tiling, the set of loop structures to be evaluated might be too large, precluding their complete evaluation and necessitating the use of heuristics.

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

1. **Four-index transform (4index):** Introduced in Section 2.
2. **CCSD:** The second and the third computations are from the class of Coupled Cluster (CC) equations [7, 14, 16] for ab initio electronic structure modeling. The sequence of tensor contraction expressions extracted from this computation is shown as follows:

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

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

$$S(h3,h4,p1,p2) = \sum_{p9,h6,h8} (y\_ooovvv(h8,h6,h4,p9,p1,p2) \times \\ \sum_{h10} \Big(t\_vo(p9,h10) \times \sum_{p7} (t\_vo(p7,h8) \times \\ \sum_{p5} (t\_vo(p5,h6) \times v\_oovv(h10,h3,p7,p5))\Big)\Big)\Big)$$

We evaluated the fused subtree corresponding to the entire operation tree without any cut-points. The number of all possible loop structures and the number of candidate loop structures enumerated by our approach are shown

Table 1. Effectiveness of pruning of loop structures.

	#Contractions	#Loop structures		Reduction
		Total	Pruned	
4index	4	241	5	98%
CCSD	3	69	2	97%
CCSDT	4	182	5	98%

in Table 1. It can be seen that a very large fraction of the set of possible loop structures, up to 98%, is pruned away using the approach developed in this paper.

6 Conclusions

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

Acknowledgments This work is supported in part by the National Science Foundation through awards 0121676, 0121706, 0403342, 0508245, 0509442, 0509467, and 0541409.

References

1. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. of ACM Intl. Conf. on Supercomputing*, 2000.
2. N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loops nests. In *Proc. of SC 2000*, 2000.
3. 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 SC 2002*, November 2002.
4. D. Cociorva, G. Baumgartner, C. Lam, J. Ramanujam P. Sadayappan, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. In *Proc. of ACM SIGPLAN PLDI 2002*, pages 177–186, 2002.
5. 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 *Proc. of IPDPS*, 2003.

6. 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 *Proc. of the Intl. Conf. on High Performance Computing*, volume 2228, pages 237–248. Springer-Verlag, 2001.
7. T. Crawford and H. F. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In K. Lipkowitz and D. Boyd, editor, *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley, 2000.
8. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.*, 64(1):108–134, 2004.
9. G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective Loop Fusion for Array Contraction. In *Proc. of the Fifth LCPC Workshop*, 1992.
10. K. Kennedy. Fast greedy weighted fusion. In *Proc. of ACM Intl. Conf. on Supercomputing*, 2000.
11. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proc. of Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, 1993.
12. S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. In *Proc. of IPDPS*, page 34b, 2004.
13. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999.
14. T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff, editor, *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pages 47–109. Kluwer Academic, 1997.
15. A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Computing*, 24(3-4):445–475, May 1998.
16. J. M. L. Martin. Benchmark Studies on Small Molecules. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, and H. F. Schaefer III, editors, *Encyclopedia of Computational Chemistry*, volume 4, pages 115–128. John Wiley, 1998.
17. S. K. Sahoo, S. Krishnamoorthy, R. Panuganti, and P. Sadayappan. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *Proc. of Supercomputing (SC 2005)*, 2005.
18. S. Singhai and K. S. McKinley. Loop Fusion for Parallelism and Locality. In *Proc. of Mid-Atlantic States Student Workshop on Programming Languages and Systems*, 1996.
19. Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. of ACM SIGPLAN PLDI*, 1999.