

Memory-Constrained Data Locality Optimization for Tensor Contractions

Alina Bibireata¹, Sandhya Krishnan¹, Gerald Baumgartner¹, Daniel Cociorva¹,
Chi-Chung Lam¹, P. Sadayappan¹, J. Ramanujam², David E. Bernholdt³, and
Venkatesh Choppella³

¹ Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210, USA.
{bibireat,krishnas,gb,cociorva,clam,saday}@cis.ohio-state.edu

² Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803, USA.
jxr@ece.lsu.edu

³ Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA.
{bernholdtde,choppellav}@ornl.gov

Abstract. The accurate modeling of the electronic structure of atoms and molecules involves computationally intensive tensor contractions over large multi-dimensional arrays. Efficient computation of these contractions usually requires the generation of temporary intermediate arrays. These intermediates could be extremely large, requiring their storage on disk. However, the intermediates can often be generated and used in batches through appropriate loop fusion transformations. To optimize the performance of such computations a combination of loop fusion and loop tiling is required, so that the cost of disk I/O is minimized. In this paper, we address the memory-constrained data-locality optimization problem in the context of this class of computations. We develop an optimization framework to search among a space of fusion and tiling choices to minimize the data movement overhead. The effectiveness of the developed optimization approach is demonstrated on a computation representative of a component used in quantum chemistry suites.

1 Introduction

An increasing number of large-scale scientific and engineering applications are highly data intensive, operating on large data sets that range from gigabytes to terabytes, thus exceeding the physical memory of the machine.

Scientific applications, in particular electronic structure codes widely employed in quantum chemistry [12, 13], computational physics, and material science, require elaborate interactions between subsets of data; data cannot be simply brought into the physical memory once, processed, and then over-written by new data. Subsets of data are repeatedly moved back and forth between a small memory pool, limited physical memory, and a large memory pool, the unlimited disk. The cost introduced by these data movements has a large impact on the overall execution time of the computation. In such contexts, it is necessary to develop *out-of-core* algorithms that explicitly orchestrate the

movement of subsets of data within the memory-disk hierarchy. These algorithms must ensure that data is processed in subsets small enough to fit the machine’s main memory, but large enough to minimize the cost of moving data between disk and memory.

This paper presents an approach to the automated synthesis of *out-of-core* programs with particular emphasis on the Tensor Contraction Engine (TCE) program synthesis system [1, 3, 2, 5, 4]. The TCE targets a class of electronic structure calculations, which involve many computationally intensive components expressed as tensor contractions (essentially generalized matrix products involving higher dimensional matrices). Although the current implementation addresses tensor contraction expressions arising in quantum chemistry, the approach developed here has broader applicability; we believe it can be extended to automatically generate efficient out-of-core code for a range of computations expressible as imperfectly nested loop structures operating on arrays potentially larger than the physical memory size.

The evaluation of such expressions involves explicit decisions about:

- the structure of loops, including tiling strategies
- the evaluation order of intermediate arrays
- memory operations (allocate, deallocate, reallocate)
- disk operations (read, write)

The fundamental compiler transforms that we apply here are loop fusion and loop tiling:

- **Loop Fusion:** The evaluation of the tensor contraction expressions often results in the generation of large temporary arrays that would be too large to be produced entirely in main memory by a “producer” loop nest and then consumed by a “consumer” loop nest. By suitably fusing common loops in the producer and consumer loop nests, it is feasible to reduce the dimensionality of the buffer array used in memory and store the intermediate array on disk. Thus, a smaller in-memory array may be used to produce the full disk array in chunks.
- **Loop Tiling:** It enables data locality to be enhanced, so that the cost of moving data to/from disk is decreased.

For minimizing the disk access cost under a given memory constraint, the compiler needs to search among many possible loop fusion structures, tile sizes, and placements of temporaries on disk. Conceptually, it is necessary to search along all three dimensions simultaneously. A decoupled approach that first searches for a fusion structure that minimizes the memory usage, followed by tiling and disk placements [5], may produce code with a sub-optimal disk-access cost as an example in the next section illustrates. A decoupled approach that first optimizes disk access by tiling the loops and placing temporaries on disk, followed by loop fusion for reducing memory usage, may fail to find a solution that fits into memory since the constraints imposed by tiling prohibit many possible fusions. However, a simultaneous search along all three dimensions is computationally infeasible.

In this paper, we present an integrated approach in which we first search for possible fusion structures together with disk placements. The result of this search is a set of candidate loop structures with different memory requirements and different combinations of disk placements for the temporaries. For each of the solutions from this search

we then search for the tile sizes that minimize the disk access cost [6]. We present two algorithms for the combined fusion and disk placement search: an optimal algorithm that is guaranteed to find the solution that will have the minimum disk access cost after tiling and a heuristic algorithm that is more efficient but may result in a suboptimal solution after tiling.

The rest of the paper is organized as follows. In the next section, we discuss the class of computations that we consider and discuss an example from computational chemistry. In Sec. 3, we introduce the main concepts used in this paper. Sec. 4 presents an optimal fusion plus tiling algorithm. Sec. 5 presents a suboptimal, but empirically efficient fusion plus tiling algorithm. Sec. 6 presents experimental evidence of the performance of this algorithm, and conclusions are provided in Sec. 7.

2 The Computational Context

We consider the class of computations in which the final result to be computed can be expressed in terms of tensor contractions, essentially a collection of multi-dimensional summations of the product of several input arrays. There are many different ways to compute the final result due to commutativity, associativity and distributivity. The ways to compute the final result could differ widely in the number of floating point operations required, in the amount of memory needed, and in the amount of disk-to-memory traffic.

As an example, consider a transformation often used in quantum chemistry codes 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, $A(p,q,r,s)$ is an input four-dimensional array (assumed to be initially stored on disk), and $B(a,b,c,d)$ is the output transformed array, which needs to be placed on disk at the end of the calculation. The arrays $C1$ through $C4$ are called transformation matrices.

The indices $p, q, r,$ and s denote the total number of orbitals, and have the same range N equal to $O + V$, where O is the number of occupied orbitals in the chemistry problem and V is the number of unoccupied (virtual) orbitals. Similarly, the indices $a, b, c,$ and d have the same range equal to V . Typical values for O range from 10 to 300, and the number of virtual orbitals V is usually between 50 and 1000.

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

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

The result of this operation-minimal approach is the creation of three temporary intermediate arrays $T1, T2,$ and $T3$ as follows: $T1(a,q,r,s) = \sum_p C4(a,p)A(p,q,r,s)$, $T2(a,b,r,s) = \sum_q C3(b,q)T1(a,q,r,s)$, and $T3(a,b,c,s) = \sum_r C2(c,r)T2(a,b,r,s)$. Assuming that the available memory limit on the machine running this calculation is less

than V^4 (which is 3TB for $V = 800$), any of the logical arrays A , $T1$, $T2$, $T3$, 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 once 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 containing A , $T1$, $T2$, $T3$, and B . Since none of these arrays can be fully stored in memory, it may not be possible to perform all multiplication operations by reading each element of the input arrays from disk only once.

We use loop fusion to reduce the memory requirements for the temporary arrays and loop fusion together with loop tiling to reduce the disk access volume. For illustrating the interactions between fusion and tiling consider the following simple example with only two contractions:

$$D_{ij} = \sum_k A_{ik} \times \left(\sum_l B_{kl} \times C_{jl} \right).$$

To prevent the intermediate array $t[k, j] = \sum_l B_{kl} \times C_{jl}$ from having to be written to disk in case it does not fit in memory, we need to fuse loops between the producer and the consumer of $t[k, l]$. This results in the intermediate array being formed and used in a pipelined fashion. For every loop that is fused between the producer and the consumer of an intermediate, the corresponding dimension can be removed from the intermediate. E.g., in the loop structure in Fig. 1(a), the intermediate $t[k, l]$ could be reduced to a scalar, while in the loop structure in Fig. 2(a), it could only be reduced to a vector $t[k]$.

<pre> FOR i, j D[i,j] = 0 FOR j, k t = 0 FOR l t += C[j,l] * B[k,l] FOR i D[i,j] += A[i,k] * t </pre> <p>(a) Memory minimal loop structure</p>	<pre> FOR iT, jT Initialize(D[iI,jI],0.0) Write D[iI,jI] to D[iT + iI,jT + jI] FOR jT, kT Initialize(t[jI,kI],0.0) FOR lT C[jI,lI] = Read C[jT + jI,lT + lI] B[kI,lI] = Read B[kT + kI,lT + lI] FOR jI, kI, lI t[jI,kI] += C[jI,lI] * B[kI,lI] FOR iT D[iI,jI] = Read D[iT + iI,jT + jI] A[iI,kI] = Read A[iT + iI,kT + kI] FOR jI, kI, iI D[iI,jI] += A[iI,kI] * t_2[jI,kI] Write D[iI,jI] to D[iT + iI,jT + jI] </pre> <p>(b) Tiled loop structure</p>
--	--

Fig. 1. Illustration of the decoupled approach for a simple example

Notice that for reducing the memory requirements of the temporary to a scalar in Fig. 1(a), it is necessary to have the file read operations for B and C inside the innermost loop. This results in the input arrays to be read redundantly multiple times. In this example, B is read once for every iteration of the j loop, while C is read once for every iteration of the k loop.

<pre> FOR j FOR k t[k] = 0.0 FOR l C = Read C[j,l] FOR k B = Read B[k,l] t[k] += B * C FOR i D = 0.0 FOR k A = Read A[i,k] D += A * t[k] Write D to D[i,j] </pre>	<pre> FOR jT FOR kT, jI, kI t[kT + kI, jI] = 0.0 FOR lT C[jI, lI] = Read C[j, l] FOR kT B[kI, lI] = Read B[k, l] FOR jI, lI, kI t[kT + kI, jI] += B[kI, lI] * C[jI, lI] FOR iT FOR jI, iI D[iI, jI] = 0.0 FOR kT A[iI, kI] = Read A[i, k] FOR jI, iI, kI D[iI, jI] += A[iI, kI] * t[kT + kI, jI] Write D[iI, jI] to D[i, j] </pre>
<p>(a) Best loop structure with temporary in memory</p>	<p>(b) Tiled loop structure</p>

Fig. 2. Illustration of the integrated approach for a simple example

The number of redundant read operations can be reduced by tiling the loops and reading entire tiles in one operation as illustrated in Fig. 1(b). B , e.g., is now only read redundantly once for every iteration of the jT tiling loop. In exchange, the memory requirement increases since all fused array dimensions get expanded to tile size. The disk access volume for a given loop structure can, therefore, be minimized by increasing the tile sizes until the memory is exhausted.

In our previous decoupled approach to fusion and tiling, we first fused the loops in order to minimize the memory usage. The memory-minimal loop structure was then tiled to minimize the disk access cost, as shown in Fig. 1. We found that for some examples, this resulted in suboptimal solutions, since there were too many redundant read operations for the input arrays. Also, the memory-minimal loop structure often results in the summation loop being the outer-most loop for a contraction. This requires the initialization of the result array to be outside the non-summation tiling loops, which then requires both a read and a write operation for the result array. This is illustrated with array D in Fig. 1(b).

Minimizing the disk access cost before fusion by deciding which temporaries to put on disk is not possible, since the resulting constraints on the loop structure might prevent the solution from fitting in memory. Also, since fusion can eliminate the need of writing some temporaries to disk, it can help reduce the disk access cost. What is, therefore, needed is an integrated approach in which we minimize the disk access cost under a memory constraint. The loop structure in Fig. 2 is the result of such an integrated approach.

It is not feasible, to simultaneously search for all possible loop structures and all possible tile sizes. Instead, we first produce a set of candidate loop structures and decide which of the temporaries are written to disk for a given loop structure. For each candidate solution in this set, we then determine the tile sizes that minimize the disk access cost. Finally, we select the tiled loop structure with the minimal disk access cost. We have previously described the tile size search and the proper placement of I/O oper-

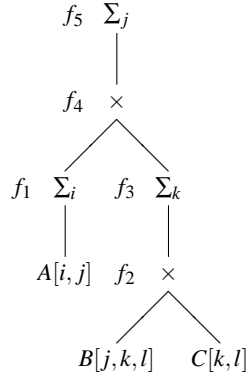
ations in the tiled loop structure [6]. In this paper, we concentrate on the algorithms for finding the candidate solutions for the tile size search.

3 Preliminaries

Before describing the algorithms, we first need to present the notions of expression trees, fusions, and nestings. Since these concepts, as well as the algorithms, are not limited to tensor contraction expressions, we describe them in the context of arbitrary sums-of-products expressions. For more detailed explanation, readers are referred to [7–11]. As an example to illustrate the concepts, we use the multi-dimensional summation shown in Figure 3(a) represented by the expression tree in Figure 3(b). One way to fuse the loops is shown in Figure 3(c).

$$W[k] = \sum_i \sum_j \sum_l (A[i, j] \times B[j, k, l] \times C[k, l])$$

(a) A multi-dimensional summation



(b) An expression tree for computing (a)
(c) A loop fusion configuration for (b)

```

Initialize f1[j]
for i
  for j
    [ A = Read A[i, j]
      f1[j] += A
    ]
  Initialize f5[k]
  for k
    for l
      [ C[l] = Read C[k, l]
        for j
          [ Initialize f3
            for l
              [ B = Read B[j, k, l]
                f2 = B × C[l]
                f3 += f2
              ]
            f4 = f1[j] × f3
          ]
          f5[k] += f4
        ]
      ]
    ]
  ]

```

Fig. 3. An example multi-dimensional summation.

Indexset sequence. To describe 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 $\langle ik, j \rangle$. Let g and g' be indexset sequences. We denote by $Set(g)$ the union of all indexsets in g , i.e., $Set(g) = \bigcup_{1 \leq r \leq |g|} g[r]$. For instance, $Set(f) = Set(\langle j, i, k \rangle) = \{i, j, k\}$.

Fusion. 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 fusion scopes in the

subtree from largest to smallest; two loops with the same fusion scope have the same rank (i.e., are in the same indexset). In the example, the fusion between B and f_2 is $\langle k, jl \rangle$.

Nesting. Similarly, a *nesting* of the loops at a node v can be defined 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. In the example, the loop nesting at f_2 is $\langle k, jl \rangle$ (because the fused k -loop covers one more node, namely C).

The “more-constraining” relation on nestings. A nesting h at a node v is said to be *more or equally constraining than* another nesting h' at the same node, denoted $h \sqsubseteq h'$, if any loop fusion configuration for the rest of the expression tree that works with h also works with h' . This relation allows us to do effective pruning among the large number of loop fusion configurations for a subtree.

4 Optimal Fusion + Tiling Algorithm

We derive the memory usage and the disk access volume of arrays in tiled, imperfectly nested loops as follows. Without tiling, the memory usage of an array is the product of the ranges of its unfused dimensions. With tiling, the tile sizes of the fused dimensions also contribute to the product. The disk access volume is the size of the array times the trip counts of the loops surrounding the read/write statement but not corresponding to the dimensions of the array. Without tiling, the trip counts of such extra loops are simply their index ranges. With tiling, the trip counts become their index ranges divided by their tile sizes. In addition, if partial sums are produced and written to disk, they need to be read back into memory, thus doubling the disk access volume.

$$MemUsage(A, f) = \prod_{i \in FusedDimens(A, f)} T_i \times \prod_{i \in UnfusedDimens(A, f)} N_i$$

$$DiskCost(A, f) = WriteFactor(A, f) \times \prod_{i \in A.dimens} N_i \times \prod_{i \in ExtraLoops(A, f)} N_i / T_i$$

where

$$WriteFactor(A, f) = \begin{cases} 2 & \text{if } f \text{ is the fusion between } produce\ A \text{ and } write\ A \\ & \text{and } A.dimens \subset Set(f) \\ 1 & \text{otherwise} \end{cases}$$

$$FusedDimens(A, f) = A.dimens \cap Set(f)$$

$$UnfusedDimens(A, f) = A.dimens - Set(f)$$

$$ExtraLoops(A, f) = Set(f) - A.dimens$$

and f is the fusion between *read* A and *consume* A , between *produce* A and *consume* A , or between *produce* A and *write* A .

As an example, for a disk-resident array $X[i, j, k]$, if the fusion between *produce* X and *write* X is $g = \langle ij \rangle$, then we have from the above equations:

$$FusedDimens(X, g) = \{i, j\}$$

$$UnfusedDimens(X, g) = \{k\}$$

$$MemUsage(X, g) = T_i \times T_j \times N_k$$

$$WriteFactor(X, g) = 1$$

$$ExtraLoops(X, g) = \emptyset$$

$$DiskCost(X, g) = N_i \times N_j \times N_k$$

Note that if an intermediate array is written to disk, it would have two potentially-different *MemUsage*: one for before writing to disk and one after reading back from disk. Similarly, it would have two *DiskCost*: one for writing it and one for reading it.

Since *MemUsage* and *DiskCost* depend on tile sizes, it may appear we cannot compare *MemUsage* and *DiskCost* between different fusions without knowing the tile sizes. However, some comparison is still possible. Continuing with the above example, if the fusion between *produce X* and *write X* is $g' = \langle il \rangle$, then:

$$MemUsage(X, g') = T_i \times N_j \times N_k$$

$$DiskCost(X, g') = N_i \times N_j \times N_k \times N_l / T_i$$

No matter what tile sizes are used for g' , we can use the same tile sizes for g and assure that $MemUsage(X, g) \leq MemUsage(X, g')$ and $DiskCost(X, g) \leq DiskCost(X, g')$ because $T_j \leq N_j$ and $N_l / T_i \geq 1$. Hence, fusion g' for array X is inferior to fusion g and can be pruned away.

Generalizing from this example, we obtain the sufficient conditions for a fusion to result in less or equal *MemUsage* or *DiskCost* than another one.

$$LeqMemUsage(A, f, f') = FusedDimens(A, f) \supseteq FusedDimens(A, f')$$

$$LeqDiskCost(A, f, f') = ExtraLoops(A, f) \subseteq ExtraLoops(A, f')$$

The first condition above implies $UnfusedDimens(A, f) \subseteq UnfusedDimens(A, f')$ and hence $MemUsage(A, f) \leq MemUsage(A, f')$ for same set of tile sizes because $T_i \leq N_i$ for any index i . Similarly, the second condition above (for $LeqDiskCost(A, f, f')$) implies $WriteFactor(A, f) \leq WriteFactor(A, f')$ and $DiskCost(A, f) \leq DiskCost(A, f')$ for same set of tile sizes because $N_i / T_i \geq 1$ for any index i .

In our example, both $LeqMemUsage(X, g, g')$ and $LeqDiskCost(X, g, g')$ are true because $FusedDimens(X, g) = \{i, j\}$ is a superset of $FusedDimens(X, g') = \{i\}$ and $ExtraLoops(X, g) = \emptyset$ is a subset of $ExtraLoops(X, g') = \{l\}$.

To apply *LeqMemUsage* and *LeqDiskCost* to compare different solutions corresponding to different fusion configurations for a subtree, we need to consider the different combinations of whether each array is disk-resident or not.

$$LeqMemUsage(s, s') \equiv$$

\forall array A in the subtree rooted at $s.root$,

{

$$\begin{aligned}
& \text{LeqMemUsage}(A, s.A.f_r, s'.A.f_r) \text{ and} \\
& \text{LeqMemUsage}(A, s.A.f_w, s'.A.f_w) && \text{if } s.A.ondisk \text{ and } s'.A.ondisk \\
& \text{FusedDimens}(A, s.A.f_r) \supset \text{FusedDimens}(A, s'.A.f_c) \text{ and} \\
& \text{FusedDimens}(A, s.A.f_w) \supset \text{FusedDimens}(A, s'.A.f_c) && \text{if } s.A.ondisk \text{ and not } s'.A.ondisk \\
& \text{LeqMemUsage}(A, s.A.f_c, s'.A.f_r) \text{ and} \\
& \text{LeqMemUsage}(A, s.A.f_c, s'.A.f_w) && \text{if not } s.A.ondisk \text{ and } s'.A.ondisk \\
& \text{LeqMemUsage}(A, s.A.f_c, s'.A.f_c) && \text{if not } s.A.ondisk \text{ and} \\
& && \text{not } s'.A.ondisk \\
& \} \\
\text{LeqDiskCost}(s, s') \equiv & \\
& \forall \text{ array } A \text{ in the subtree rooted at } s.root, \\
& \{ \\
& \text{LeqDiskCost}(A, s.A.f_r, s'.A.f_r) \text{ and} \\
& \text{LeqDiskCost}(A, s.A.f_w, s'.A.f_w) && \text{if } s.A.ondisk \text{ and } s'.A.ondisk \\
& \text{not } s.A.ondisk && \text{otherwise} \\
& \} \\
\text{where}
\end{aligned}$$

$s.A.ondisk$ means array A is disk-resident in solution s
 $s.A.f_r$ is the fusion between *read* A and *consume* A in solution s
 $s.A.f_w$ is the fusion between *produce* A and *write* A in solution s
 $s.A.f_c$ is the fusion between *produce* A and *consume* A in solution s

For input or final-result arrays where fusions f_r or f_w do not apply, or for intermediate disk-resident arrays where fusion f_r is yet to be decided, such fusions are considered empty sets.

Making use of the above results, we can compare and prune solutions as follows. A solution that has higher or equal memory usage and disk access cost and a more or equally constraining nesting than another solution is considered inferior and can be pruned away safely. Between solutions for the entire tree and between solutions for a subtree whose root array is disk-resident and its fusion f_r is undecided, pruning without the condition of a more or equally constraining nesting is also safe.

$$\begin{aligned}
\text{Inferior}(s', s) \equiv & \\
& \text{LeqMemUsage}(s, s') \text{ and} \\
& \text{LeqDiskCost}(s, s') \text{ and} \\
& (s'.root.nesting \sqsubseteq s.root.nesting \text{ or} \\
& \quad s.root = \text{Root} \text{ or} \\
& \quad (s.root.ondisk \text{ and } s'.root.ondisk \text{ and } s.root.f_r = s'.root.f_r = \emptyset))
\end{aligned}$$

A dynamic programming, bottom-up algorithm using the *Inferior* condition as a pruning rule works as follows. For each leaf node (corresponding to an input array) in the tree, one solution is formed for each possible fusion f_r (or f_c if it is not disk-resident) with its parent and then inferior solutions are pruned away. For each intermediate array A in the tree, all possible legal fusions f_w and f_c , for writing A to disk or not respectively, are considered in deriving new solutions from the children of A . Solutions that write A to disk are pruned against each other before all possible legal fusions f_r are enumerated

to derive new solutions. Then all inferior solutions for the subtree rooted at A , whether writing A to disk or not, are pruned away. For the root of tree, if it is to be written to disk, all possible legal fusions f_w are considered in deriving new solutions. Finally, all inferior solutions for the entire tree are pruned away.

Although this approach is guaranteed to find an optimal solution, it could be expensive. The reason is the condition $LeqMemUsage(s, s')$ requires each and every array in the subtree in solution s to have lower or equal memory usage than the corresponding array in solution s' , and similarly for $LeqDiskCost(s, s')$ in terms of disk access cost. If either the memory usage or the disk access cost of any array in s is incomparable to the corresponding array in s' , no solution derived from s for a larger subtree would be comparable to any solution derived from s' . Thus, in the worse case, the number of unpruned solutions for the entire tree could grow exponentially in the number of arrays. Due to its exponential complexity, we have yet to implement this approach.

5 Efficient Fusion + Tiling Algorithm

Since the optimal fusion and tiling algorithm is impractical to implement, due to its large number of unpruned solution, we have devised a sub-optimal, efficient algorithm to solve the fusion and tiling problem. The central idea of this algorithm is to first fix a tile size T common to all the tiled loops, and, based on this tile size, determine a set of candidate solutions by a bottom-up tree traversal. In the second part of the algorithm, the tile sizes are allowed to vary, and optimal tile sizes are determined for all candidate solutions. The candidate solution with the lowest disk cost is finally chosen as the best overall solution.

Our current implementation of the first part of the algorithm uses $T = 1$. With $WriteFactor(A, f)$, $FusedDimens(A, f)$, $UnfusedDimens(A, f)$, and $ExtraLoops(A, f)$ defined according to Section 4, the memory usage and disk cost for an array A become:

$$MemUsage(A, f) = \prod_{i \in UnfusedDimens(A, f)} N_i$$

$$DiskCost(A, f) = WriteFactor(A, f) \times \prod_{i \in Set(f)} N_i$$

where f is the fusion between *read* A and *consume* A , between *produce* A and *consume* A , or between *produce* A and *write* A .

When an intermediate array is stored on disk, it has two $MemUsage$: one for before writing to disk and one after reading back from disk. In this case, we define $MemUsage$ as the maximum of the two values. Similarly, the array has two $DiskCost$: one for writing it and one for reading it. We define the total disk cost of an intermediate array that is stored on disk as the sum of the disk costs for writing it and for reading it back.

With these definitions, we calculate the memory usage and disk cost of a solution s corresponding to a given fusion configuration for a subtree:

$$MemUsage(s) = \sum_{A \text{ in the subtree rooted at } s.root} MemUsage(A, f_s)$$

$$DiskCost(s) = \sum_{A \text{ in the subtree rooted at } s.root} DiskCost(A, f_s)$$

where f_s is the fusion between *read A* and *consume A*, between *produce A* and *consume A*, or between *produce A* and *write A* given the fusion configuration of the solution s .

Different solutions corresponding to different fusion configurations for a subtree are now easily comparable:

$$LeqMemUsage(s, s') \equiv \{MemUsage(s) \leq MemUsage(s')\}$$

$$LeqDiskCost(s, s') \equiv \{DiskCost(s) \leq DiskCost(s')\}$$

Making use of the above results, we can introduce pruning rules similar to those of the optimal algorithm: a solution that has higher or equal memory usage and disk access cost and a more or equally constraining nesting than another solution is considered inferior and can be pruned away safely.

$$\begin{aligned} Inferior(s', s) \equiv & \\ & LeqMemUsage(s, s') \text{ and} \\ & LeqDiskCost(s, s') \text{ and} \\ & (s.root.nesting \sqsubseteq s'.root.nesting \text{ or} \\ & s.root = Root \text{ or} \\ & (s.root.ondisk \text{ and } s'.root.ondisk \text{ and } s.root.f_r = s'.root.f_r = 0)) \end{aligned}$$

A dynamic programming, bottom-up algorithm using the *Inferior* condition as a pruning rule works in the same fashion as the optimal algorithm described in Section 4. The major difference between the optimal algorithm and the efficient algorithm is that the *Inferior*(s, s') condition is more relaxed in the latter: we no longer require that the *MemUsage* and *DiskCost* inequalities be valid for all individual arrays in the subtree rooted at $s.root$. Instead, only the sums of *MemUsage* and *DiskCost* over the entire subtree need to be compared.

The result of this approach is a set of candidate solutions that are characterized by pairs of the form ($MemUsage(s), DiskCost(s)$). The algorithm described above prunes away all solutions that have higher *MemUsage* and *DiskCost* under the tile size constraint $T = 1$. For each candidate solution in the set, we then search for the tile sizes that minimize the disk access cost [6]. Increasing the tile sizes causes the disk access cost to decrease and the memory usage to increase, since array dimensions that have been eliminated by fusion get expanded to tile size. Finally, we select the solution with the least disk access cost.

6 Experimental Evaluations

We used the algorithm from Sec. 5 to generate code for the AO-to-MO index transformation calculation described in Sec 2. The algorithm generated 77 candidate solutions that would then be run through the tiling algorithm. We present two representative solutions generated by this algorithm.

The solution shown in Fig. 4 places only temporary $T2$ on disk, while the solution shown in Fig. 5 places only the temporary $T1$ on disk. After tile size search, the tiled code with the least disk access cost was the one based on the solution with $T2$ on disk. The optimal code is shown in Fig. 6.

```

FOR r, s
  FOR a, q
    T1[a,q] = 0.0
  FOR p
    C4[a] = Read C4[p,a]
    FOR q
      A = Read A[p,q,r,s]
      FOR a
        T1[a,q] += A * C4[a]
  FOR b
    FOR a
      T2[a] = 0.0
    FOR q
      C3 = Read C3[q,b]
      FOR a
        T2[a] += T1[a,q] * C3
  Write T2 to T2[a,b,r,s]

FOR a, b, c
  FOR s
    T3 = 0.0
  FOR r
    C2 = Read C2[r,c]
    FOR s
      T2 = Read T2[a,b,r,s]
      T3 += T2 * C2
  FOR d
    B = 0.0
    FOR s
      C1 = Read C1[s,d]
      B += T3 * C2
  Write B to B[a,b,c,d]

```

Fig. 4. Fused Structure with temporary T2 on disk

```

FOR q, r, s
  FOR a
    T1[a] = 0.0
  FOR p
    A = Read A[p,q,r,s]
    FOR a
      C4 = Read C4[p,a]
      T1[a] += A * C4
  Write T1[a] to T1[a,q,r,s]

FOR b
  C3[q] = Read C3[q,b]
  FOR a
    FOR s, c
      T3[s,c] = 0.0
    FOR r
      C2[c] = Read C2[r,c]
      FOR s
        T2 = 0.0
        FOR q
          T1 = Read T1[a,q,r,s]
          T2 += T1 * C3[q]
        FOR c
          T3[s,c] += T2 * C2[c]
    FOR d
      FOR c
        B[c] = 0.0
      FOR s
        C1 = Read C1[s,d]
        FOR c
          B[c] += T3[s,c] * C1
      Write B[c] to B[a,b,c,d]

```

Fig. 5. Fused Structure with temporary T1 on disk

```

FOR rT, sT
  FOR aT, qT, rI, sI, aI, qI
    T1[aT+aI,qT+qI,rI,sI] = 0.0
  FOR pT
    C4[pI,aT+aI] = Read C4[p,a]
    FOR qT
      A[pI,qI,rI,sI] = Read A[p,q,r,s]
      FOR aT, rI, sI, pI, qI, aI
        T1[aT+aI,qT+qI,rI,sI] +=
          A[pI,qI,rI,sI] * C4[pI,aT+aI]
  FOR bT
    FOR aT, rI, sI, bI, aI
      T2[aT+aI,bI,rI,sI] = 0.0
    FOR qT
      C3[qI,bI] = Read C3[q,b]
      FOR aT, rI, sI, bI, qI, aI
        T2[aT+aI,bI,rI,sI] +=
          T1[aT+aI,qT+qI,rI,sI]*C3[qI,bI]
    Write T2[aT+aI,bI,rI,sI] to
      T2[a,b,r,s]

FOR aT, bT, cT
  FOR sT, aI, bI, cI, sI
    T3[aI,bI,cI] = 0.0
  FOR rT
    C2[rI,cI] = Read C2[r,c]
  FOR sT
    T2[aI,bI,rI,sI] =
      Read T2[a,b,r,s]
    FOR aI, bI, cI, rI, sI
      T3[aI,bI,cI] +=
        T2[aI,bI,rI,sI]*C2[rI,cI]
  FOR dT
    FOR aI, bI, cI, dI
      B[aI,bI,cI,dI] = 0.0
    FOR sT
      C1[dI,sI] = Read C1[d,s]
      FOR aI, bI, cI, dI, sI
        B[aI,bI,cI,dI] +=
          T3[aI,bI,cI]*C2[dI,sI]
    Write B[aI,bI,cI,dI] to B[a,b,c,d]

```

Fig. 6. Loop Structure after tiling

Table 1. Configuration of the system whose I/O characteristics were studied.

Processor	OS	Compiler	Memory	Hard disk
Pentium II 300 MHz	Linux 2.4.18-3	gcc version 2.96	128MB	Maxtor 6L080J4

Measurements were taken on a Pentium II system with the configuration shown in Table 1. The codes were all compiled with the Intel Fortran Compiler for Linux. Although this machine is now very old and much slower than PCs available today, it was convenient to use for our experiments in an uninterrupted mode, with no interference to the I/O subsystem from any other users.

Table 2. Predicted and Measured I/O Time: a solution generated by the new fusion-datalocality algorithm for the AO-to-MO transform example.

	Predicted Results(seconds)	Measured Results(seconds)
Array A	21.3	31
Array B	18.25	14
Array T2	40.14	41
Arrays C1,C2,C3,C4	0.052	0.72
Total time	79.74	86.7

Table 2 shows the measured I/O time for the AO-to-MO transform where the sizes of the tensors (double precision) considered were: $N_p = N_q = N_r = N_s = 80$ and $N_a = N_b = N_c = N_d = 70$. We used 100MB as the memory limit. The I/O time for each array was separately accumulated. The predicted values match quite well with the measured time. The match is better for the overall I/O time than for some individual arrays. This is because disk writes are asynchronous and may be overlapped with succeeding disk reads — hence the measurements of I/O time attributable to individual arrays is subject to error due to such overlap, but the total time should not be affected by the interleaving of writes with succeeding reads. For these tensor sizes and an available memory of 100MB, it is possible to choose fusion configurations so that the sizes of any two out of the three intermediate arrays can be reduced to fit completely in memory, but it is impossible to find a fusion configuration that fits all three intermediates within memory. Thus, it is necessary to keep at least one of them on disk, and incur disk I/O cost for that array.

Table 3 shows the predicted I/O times and the improvement factor of the integrated fusion+tiling algorithm over the decoupled algorithm for the AO-to-MO transformation example for different array sizes and memory limits. For the arrays sizes $a = 70$ and $p = 80$, actual measurements were performed using the 100MB, 500MB, and 2000MB memory limits and, in all cases, for the integrated algorithm, the predicted results matched the actual results. For the memory limits of 500MB and 2000MB and the small array sizes, both the decoupled and the integrated algorithm were able to fit all the temporaries in memory, and thus no significant improvement was achieved.

Table 3. Comparison of predicted I/O time for the AO-to-MO transform example.

Ranges	Decoupled	Integrated	Improvement factor
100MB			
a=70, p=80	1.882×10^2 sec	0.747×10^2 sec	2.52
a=200, p=300	7.390×10^4 sec	0.850×10^4 sec	8.70
a=500, p=600	5.520×10^6 sec	2.300×10^5 sec	24.00
500MB			
a=70, p=80	0.395×10^2 sec	0.395×10^2 sec	1.00
a=200, p=300	4.830×10^4 sec	0.850×10^4 sec	5.70
a=500, p=600	3.560×10^6 sec	2.300×10^5 sec	15.50
2000MB			
a=70, p=80	0.395×10^2 sec	0.395×10^2 sec	1.00
a=200, p=300	3.780×10^4 sec	0.850×10^4 sec	4.45
a=500, p=600	2.140×10^6 sec	2.109×10^5 sec	10.14

Depending on the size of the problem, as the memory pressure increases, the improvement factor of the integrated algorithm over the decoupled algorithm increases significantly. This is to be expected, because the decoupled algorithm introduces more redundant reads and writes than the integrated algorithm. With high memory pressure, the tiles cannot be made very large, which results in an insufficient reduction of the redundant disk accesses.

The measured results and the predicted results match well and the integrated fusion+tiling algorithm outperforms the decoupled data locality algorithm.

7 Conclusion

We have described an optimization approach for synthesizing efficient out-of-core algorithms in the context of the Tensor Contraction Engine. We have presented two algorithms for performing an integrated fusion and tiling search. Our algorithms produce a set of candidate solutions, each with a fused loop structure and read and write operations for temporaries. After determining the tile sizes that minimize the disk access cost, the optimal solution is chosen. We have demonstrated with experimental results, that the integrated approach outperforms a decoupled approach of first determining the fused loop structure and then searching for the optimal tile sizes.

Acknowledgments We thank the National Science Foundation for its support of this research through the Information Technology Research program (CHE-0121676 and CHE-0121706), NSF grants CCR-0073800 and EIA-9986052, and the U.S. Department of Energy through award DE-AC05-00OR22725.

References

1. G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc Supercomputing 2002*, Nov. 2002.

2. 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. *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002, pp. 177–186.
3. D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. *Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2003.
4. D. Cociorva, J. Wilkins, C.-C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proc. 15th ACM International Conference on Supercomputing(ICS'01)*, pp. 500–509, Sorrento, Italy, June 2001.
5. 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. *Proc. of the Intl. Conf. on High Performance Computing*, Dec. 2001, Lecture Notes in Computer Science, Vol. 2228, pp. 237–248, Springer-Verlag, 2001.
6. S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D.E. Bernholdt, and V. Choppella. Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms. In *Proc. of the Intl. Conf. on High Performance Computing*, Dec. 2003, Hyderabad, India.
7. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, August 1999.
8. C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. *Proc. 12th LCPC Workshop* San Diego, CA, Aug. 1999.
9. C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Proc. Intl. Conf. on High Perf. Comp.*, Dec. 1999.
10. C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Par. Proc. Lett.*, (7) 2, pp. 157–168, 1997.
11. C. Lam, P. Sadayappan and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. *Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.
12. T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pp. 47–109, Kluwer Academic, 1997.
13. J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.), *Encyclopedia of Computational Chemistry*. Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115–128, 1998.