Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms

Sandhya Krishnan[1], Sriram Krishnamoorthy[1], Gerald Baumgartner[1], Daniel Cociorva[1], Chi-Chung Lam[1], P. Sadayappan[1], J. Ramanujam[2], David E. Bernholdt[3], and Venkatesh Choppella[3]

[1] Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210, USA.
{krishnas,krishnsr,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. This paper describes an approach to synthesis of efficient out-of-core code for a class of imperfectly nested loops that represent tensor contraction computations. Tensor contraction expressions arise in many accurate computational models of electronic structure. The developed approach combines loop fusion with loop tiling and uses a performance-model driven approach to loop tiling for the generation of out-of-core code. Experimental measurements are provided that show a good match with model-based predictions and demonstrate the effectiveness of the proposed algorithm.

1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of the machine. Some applications process data by *streaming*: each input data item is only brought into memory once, processed, and then over-written by other data. Other applications, like Fast Fourier Transform calculations and those modeling electronic structure using Tensor Contractions, like coupled cluster and configuration interaction methods in quantum chemistry, employ algorithms that require more elaborate interactions between data elements; data cannot be simply streamed into processor memory from disk. In such contexts, it is necessary to develop so called *out-of-core* algorithms that explicitly orchestrate the movement of subsets of the data between main memory and secondary disk storage. These algorithms ensure that data is operated in chunks small enough to fit within the system's physical memory but large enough to minimize the cost of moving data between disk and main memory.

This paper presents an approach to automatically synthesize efficient out-of-core algorithms in the context of the Tensor Contraction Engine (TCE) program synthesis tool [1, 3, 2, 4]. The TCE targets a class of electronic structure calculations which involve many computationally intensive components expressed as tensor contractions. Although the current implementation addresses tensor contractions arising in quantum chemistry, the approach developed here is more general. It can be used to automatically generate efficient out-of-core code for a broad range of computations expressible as imperfectly nested loop structures and operating on arrays potentially larger than the physical memory size.

The paper is organized as follows. In the next section, we elaborate on the out-of-core code synthesis problem in the computational context of interest. Sec. 3 presents an empirically derived model of disk I/O costs, that drives the out-of-core optimization algorithm presented in Sec. 4. Sec. 5 presents experimental performance data on the application of the new algorithm, and conclusions are provided in Sec. 6.

2 The Computational Context

In this paper we address the data locality optimization problem in the context of disk-to-memory traffic. Our proposed optimizations are part of the Tensor Contraction Engine, which 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 TCE incorporates several compile-time optimizations, including algebraic transformations to minimize operation counts [9, 10], loop fusion to reduce memory requirements [6, 8, 7], space-time trade-off optimization [2], communication minimization [3] and data locality optimization [5, 4] of memory-to-cache traffic. Although there are many similarities between the two varieties of data locality optimizations, our previous approach is not effective for the disk-to-memory context due to the constraint that disk I/O must be in large contiguous blocks in order to be efficient.

In the class of computations considered, the final result to be computed can be expressed using a collection of multi-dimensional summations of the product of several input arrays.

As an example, we consider a transformation often used in quantum chemistry 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. In reality, these four arrays are identical; we identify them by different names in our example in order to be able to distinguish them in the text.

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

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

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

This operation-minimal approach results in the creation of three temporary intermediate arrays $T1$, $T2$, and $T3$: $T1(a,q,r,s) = \sum_p C4(a,p)A(p,q,r,s)$, $T2(a,b,r,s) = \sum_q C3(b,q)T1(a,q,r,s)$, 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 array 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 and loop tiling to reduce memory requirements. To illustrate the benefit of loop fusion, consider the first two steps in the AO-to-MO transformation: $T1(a,q,r,s) = \sum_p C4(a,p)A(p,q,r,s)$; $T2(a,b,r,s) = \sum_q C3(b,q)T1(a,q,r,s)$. Fig. 1(a) shows the loop structure for the direct computation as a two-step sequence: first produce the intermediate $T1(1:Na, 1:Nq, 1:Nr, 1:Ns)$ and then use $T1$ to produce $T2(1:Na, 1:Nb, 1:Nr, 1:Ns)$. We refer to this as an *abstract* form of a specification of the computation, because it cannot be executed in this form if the sizes of arrays are larger than limits due to the physical memory size. We later address the transformation of abstract forms into concrete forms that can be executed — the concrete forms have explicit disk I/O statements between disk-resident arrays and their in-memory counterparts.

Since all loops in either of the loop nests are fully permutable, and since there are no fusion-preventing dependences, the common loops a, q, r, and s can be fused. Once fused, the storage requirements for $T1$ can be reduced by contracting it to a scalar as shown in Fig. 1(b). Although the total number of arithmetic operations remains unchanged, the dramatic reduction in size of the intermediate array $T1$ implies that it can be completely stored in memory, without the need for any disk I/O for it. In contrast, if $Na \times Nq \times Nr \times Ns$ is larger than available memory, the unfused version will require that $T1$ be written out to disk after it is produced in the first loop, and then read in from disk for the second loop.

The code synthesis process in the Tensor Contraction Engine [1] involves multiple steps, including algebraic transformation, loop fusion and loop tiling. The loop fusion and loop tiling steps are coupled together. The fusion step provides the tiling step with a set of candidate fused loop structures with desirable properties; the tiling step seeks to find the best tile sizes for each of the fused loop structures supplied, and chooses the one that permits the lowest data movement cost overall. Our previous work had focused on tiling to minimize memory-to-cache traffic. In this paper, we describe an approach to minimize disk I/O time for situations where out-of-core algorithms are needed.

```
double T1(Na,Nq,Nr,Ns)
double T2(Na,Nb,Nr,Ns)
T1(*,*,*,*) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
 FOR q = 1, Nq
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR p = 1, Np
      T1(a,q,r,s)
        += C4(a,p) * A(p,q,r,s)
END FOR p,s,r,q,a
FOR a = 1, Na
 FOR b = 1, Nb
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR q = 1, Nq
      T2(a,b,r,s)
        += C3(b,q) * T1(a,q,r,s)
END FOR q,s,r,b,a
```

(a) Unfused form

```
double T1(1,1,1,1)
double T2(Na,Nb,Nr,Ns)
T1(1,1,1,1) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
 FOR q = 1, Nq
  FOR r = 1, Nr
   FOR s = 1, Ns
    FOR p = 1, Np
      T1(1,1,1,1)
        += C4(a,p) * A(p,q,r,s)
    END FOR p
    FOR b = 1, Nb
      T2(a,b,r,s)
        += C3(b,q) * T1(1,1,1,1)
    END FOR b
END FOR s,r,q,a
```

(b) Fused code

Fig. 1. Example of the use of loop fusion to reduce memory

We have previously addressed the issue of the data locality optimization problem arising in this synthesis context, focusing primarily on minimizing memory-to-cache data movement [5, 4]. In [5], we developed an integrated approach to fusion and tiling transformations for the class of loops arising in the context of our program synthesis system. 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 [4] removed the restrictions assumed in [5]. Its cost model was based on an idealized fully associative cache with a line size of one. A tile size search procedure estimated the total capacity miss cost for a large number of combinations of tile sizes for the various loops of an imperfectly nested loop set. After the best combination of tile sizes was found, tile sizes were adjusted to address spatial locality considerations. This was done by adjusting the tile sizes for any loop indexing the fastest varying dimension of any array to have a minimum value of the cache linesize. The approach cannot be effectively used for the out-of-core context because the “linesize” that must be used in the adjustment procedure would be huge — corresponding to a minimum disk read chunk-size to ensure good I/O bandwidth.

3 Modeling Performance of Disk I/O

The tile sizes for each array on disk are chosen so as to minimize the cost of I/O for that array within the memory constraint. The I/O cost is modeled based on empirically derived I/O characteristics of the underlying system. In this section, the I/O characteristics of the system are discussed and the I/O cost model is derived.

We evaluated the I/O characteristics of a Pentium II system that was available for exclusive use, without any disk interference from other users. The details of the system are shown in Table 1. Reads and writes were done for different block sizes at different

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

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

strides and the per-byte transfer time was measured. The block size was varied from 16KB to 2MB. The strides was set to be multiples of the block size used. The read and write characteristics are shown in Fig. 2(a) and Fig. 2(b).

The graphs show that reads and writes exhibit different characteristics. This is due to the difference in their semantics. The cost of reads, which are synchronous, includes the disk access time (seek time + latency). On the other hand, writes return after copying the data to a system buffer. The I/O subsystem subsequently initiates writes of the data from the system buffer to disk. The I/O subsystem reorders the writes, reducing the average disk access time. The reordering of write requests make the cost of writing data less dependent on the actual layout of data on disk. For reads, the data transfer for a requested block can happen before the actual request arrives, due to read ahead, also called prefetching. Most file systems read ahead to exploit access locality. Read-ahead leads to lower read costs than write costs.

The read characteristics show a clear difference between sequential access and strided access even at large block sizes. This effect can be attributed to the presence of disk access time in the critical path of reads. Also, since the block sizes requested are large, the disk access time approaches the average disk access time. This additional cost, incurred for every read request, approximately doubles with halving the block size.

Below a certain block size, the reads do not completely take advantage of read ahead. For example, with a stride of two, small block sizes lead to only alternate blocks being used, though contiguous blocks are read into memory by the I/O subsystem. This has the effect of increasing the total data read. Hence, for smaller block sizes, the cost per byte increases proportionally with stride.

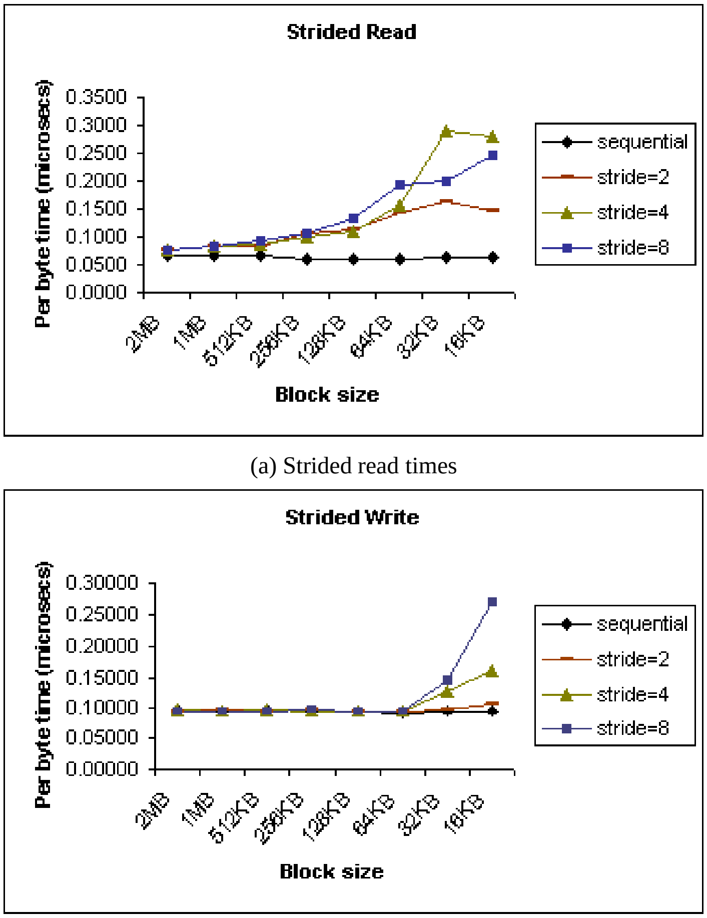
We model the per-byte cost of I/O as a linear function of block size and disk access time. Based on the observed trends in disk read time as a function of blocksize and stride, we develop a model below, for the time to access an arbitrary multi-dimensional "brick" of data from a multi-dimensional array with a linearized column-major layout on disk. Access of a brick of data will require a number of disk reads, where each read can only access a contiguous set of elements on disk.

Let $T_1, \ldots, T_4$ be the tile sizes for the four dimensions $N_1, \ldots, N_4$, respectively. The block size BS and stride S of access are determined as follows:

$$BS = \begin{cases} T_1 & \text{if } T_1 < N_1 \\ T_1 * T_2 & \text{if } T_1 = N_1 \text{ and } T_2 < N_2 \\ T_1 * T_2 * T_3 & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 < N_3 \\ T_1 * T_2 * T_3 * T_4 & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 = N_3 \end{cases}$$

$$S = \begin{cases} N_1/BS & \text{if } T_1 < N_1 \\ (N_1 * N_2)/BS & \text{if } T_1 = N_1 \text{ and } T_2 < N_2 \\ (N_1 * N_2 * N_3)/BS & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 < N_3 \\ (N_1 * N_2 * N_3 * N_4)/BS & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 = N_3 \end{cases}$$

The per-byte cost of reads is formulated as



(a) Strided read times

(b) Strided write times

Fig. 2. Strided read/write times on the Pentium II system.

$$Cost = \begin{cases} (seq + \frac{\text{avg. access time}}{BS}) & \text{if } BS \geq \text{ prefetch size} \\ (seq * S + \frac{\text{access time}}{64KB}) & \text{if } BS < \text{ prefetch size} \end{cases}$$

where prefetch size is the extent of read ahead done by the I/O subsystem. The average access time is the sum of average seek time and average latency, as provided in the specification of the disk and *seq* is the time per-byte for sequential reads. For the platform under consideration, the read ahead size was determined to be 64KB. The average access time was found to be 13 milliseconds and the per-byte sequential access time was determined as 65 nanoseconds.

Writes at different strides have the same cost as sequential writes for large block sizes. This is due to the fact that writes are not synchronous and no additional cost in the form of seek time has to be incurred. Below a certain block size, the lack of sufficient ‘locality’ increases the cost from the minimum possible. In contrast to reads, for small block sizes, the per-byte cost of write steadily increases with stride. This is due to the reordering of writes by the I/O subsystem, which diminishes the influence of actual of data layout on disk.

The graph shows a logarithmic relationship between the per-byte write time and the block size for small block sizes. The per-byte cost for small block sizes is formulated as a set of linear equations, one for each stride. Each of the equations is a function of the logarithm of the block size. The per-byte sequential write time on the system under consideration was determined to be 95 nanoseconds.

4 Out-of-Core Data Locality Optimization Algorithm

4.1 Disk File Layout

Given an imperfectly nested loop structure that specifies a set of tensor contractions, the arrays involved in the contractions fall into one of three categories: *a*) input arrays, which initially reside on disk, *b*) intermediate temporary arrays, which are produced and consumed within the specified computation and are not required at the end, and *c*) output arrays, which must be finally written out to disk.

If an intermediate array is too large to fit into main memory, it must be written out to disk and read back from disk later. Since it is a temporary entity, there is complete freedom in the choice of its disk layout. It is not necessary for it to be represented in any canonical linearized order such as row major order. A blocked representation on disk is often advantageous in conjunction with tiling of the loop computations. For example, consider the multiplication of two large $N \times N$ disk-resident arrays on a system with a memory size of M words, where $M < N^2$. Representing the arrays on disk as a sequence of $k \times k$ blocks, with $k^2 < M/3$, allows efficient read/write of blocks of the arrays to and from disk. We therefore allow the out-of-core synthesis algorithm to choose the disk representation of all intermediate arrays in a manner that minimizes disk I/O time. For the input and output arrays, the algorithm can be used in either of these modes:

- The layouts of the input and output arrays are *unconstrained*, and can be chosen by the algorithm to optimize I/O time, or
- The input and output arrays are *constrained* to be stored on disk in some pre-specified canonical representation, such as row-major order.

Even for the arrays that are not externally constrained to be in some canonical disk layout, some constraints are imposed on their blocked layout on disk and the tile sizes of loop indices that index them:

- All loop indices that index the same array dimension in multiple reference occurrences of an array must be tiled the same. This ensures that multiple reference instances of an array can all be accessed efficiently using the same blocked units on disk. Although the blocks of an array may be accessed in a different order for the different instances due to a different nesting of the loop indices, it is ensured that the basic unit of access from disk is always the same.

```
double A(1:Ni,1:Nk)
double B(1:Ni,1:Nj)
double C(1:Nj,1:Nk)

FOR i
 FOR j
  FOR k
   A(i,k) += B(i,j) * C(j,k)
END FOR k, j, i
```

(a) Abstract code for matrix multiplication. Ni=Nk=6000, Nj=2000. Memory limit=128MB.

```
double MA(1:Ti, 1:Tk)
double MB(1:Ti, 1:Tj)
double MC(1:Tj, 1:Tk)

FOR iT = 1, Ni, Ti
 FOR jT = 1, Nj, Tj
  FOR kT = 1, Nk, Tk
   MA(1:Ti,1:Tk) = Read disk array A(i,k)
   MB(1:Ti,1:Tj) = Read disk array B(i,j)
   MC(1:Tj,1:Tk) = Read disk array C(j,k)
   FOR iI = 1, Ti
    FOR jI = 1, Tj
     FOR kI = 1, Tk
      MA(1:Ti,1:Tk)
        += MB(1:Ti,1:Tj) * MC(1:Tj,1:Tk)
   END FOR kI, jI, iI
   Write MA(1:Ti,1:Tk) to disk A(i,k)
END FOR kT, jT, iT
```

(b) Tiled code with all reads and writes immediately surrounding the intra tile loops.

Fig. 3. Abstract and concrete code for matrix multiplication.

– Array dimensions of different arrays that are indexed by the same loop index variable must have the same blocking. This ensures that the unit of transfer from disk to memory for all the arrays match the tiling of the loop computations.

Before searching for the optimal tile sizes for the loops, we first need to identify constraints among loop indices that result from the above two conditions. For computing these constraints, we use a Union-Find data structure for grouping indices into equivalence classes.

First, we rename all loop indices to ensure that no two loop indices have the same name. Initially, each loop index is in its own equivalence class. In the symbol table entry for an array name, we keep track of the index equivalence classes of all array dimensions. Then, in a top-down traversals of the abstract syntax tree, for every loop index i and every array reference $A[\ldots,i,\ldots]$, we merge the equivalence class of i with the equivalence class of the array dimension indexed by i.

The index equivalence classes found by this procedure will be used to constrain the tile size search, such that all the indices in an equivalence class will be constrained to have the same tile size.

4.2 Tile Size Search Algorithm

We now describe our approach to addressing the out-of-core data locality optimization problem:

– **Input:** An *abstract* form of the computation for a collection of tensor contractions, as a set of imperfectly nested loops, operating directly on arrays that may be too large to fit in physical memory, e.g., as in Fig. 3(a).
– **Input:** Available physical memory on target system.
– **Output:** A *concrete* form for the computation, as a collection of tiled loops, with appropriate placements of disk I/O statements to move data between disk-resident arrays to corresponding memory-resident arrays, e.g., as in Fig. 3(b).

```
Index: {
    String name
    int range
    int tilesize
    int tilecount
}

CostModel: {
    double memCost(Index[] tiledIndices)
    double diskCost(Index[] tiledIndices)
}

bool MemoryExceeded (Index[] tiledIndices, CostModel C)
  return (C.memCost(tiledIndices) > memoryLimit)

TileSizeSearch (Index[] tiledIndices, CostModel C):
  foreach Index $I \in$ tiledIndices $I.tilecount = 1$
  while (MemoryExceeded (tiledIndices, C)) do
    foreach Index $I \in$ tiledIndices $I.tilecount += 1$
  if (foreach Index $I \in$ tiledIndices, $I.tilecount = 1$) then
    return
foreach Index $I \in$ tiledIndices
  while (not MemoryExceeded (tiledIndices, C))
    $I.tilecount --$
  $I.tilecount ++$
Repeat
  foreach Index $I \in$ tiledIndices
    $I.tilecount --$
    Diff[$I$] = $\Delta$diskCost $\div$ $\Delta$memCost
    $I.tilecount ++$
  Index Best_I = $I \in$ tiledIndices with max Diff[$I$]
  Best_I.tilecount $--$
  if (MemoryExceeded (tiledIndices, C))
    foreach Index $J \in$ tiledIndices($J \neq I$)
      while (MemoryExceeded (tiledIndices, C))
        $J.tilecount ++$
      EffDecrease[$J$] = $\Delta$diskCost
      Tiles[$J$] = $J.tilecount$
      Reset $J.tilecount$ to original value
    Index Best_J = $J \in$ tiledIndices
      with max EffDecrease[$J$]
    Best_J.tilecount = Tiles[Best_J]
Until (EffDecrease[Best_J] $\leq 0$)
```

Fig. 4. Procedure **TileSizeSearch** to determine optimal tile sizes that minimize disk access cost.

Given a concrete imperfectly nested loop, with proper placements of disk I/O statements, the goal of the tile size search algorithm is to minimize total disk access time under the constraint that memory limit is not be exceeded. The input to the algorithm is the cost model and the set of index equivalence classes, determined by the procedure explained in Sec. 4.1.

The disk access cost model depends on the mode of the algorithm, as explained in Sec. 4.1. For the unconstrained case, the disk cost for an array is proportional to the volume of data accessed, as each block access is sequential. On the other hand, for the constrained case, we use the I/O performance model described in Sec. 3.

Fig. 4 presents the pseudo code for the tile size search algorithm. It starts by initializing the number of tiles for each tiled index to 1. This is equivalent to all arrays being completely memory resident, and would be the optimal solution if the memory limit is not exceeded. Otherwise, the algorithm tries to find a feasible solution that satisfies the memory constraint, by iteratively incrementing the tile count for all indices. To illustrate the algorithm, consider the concrete code for matrix multiplication in Fig. 5(a). The memory cost equation for this code is:

$$T_i * N_k + T_i * T_j + T_j * N_k$$

As shown in Fig. 5(b), with a tile count of 4 for all indices i, j and k, the memory cost is 97MB, which is within the memory limit of 128MB. The algorithm then tries to reduce the number of tiles for any of the indices as much as possible, so that the solution just fits in memory. This is chosen as the starting point in the search space. For this example, the tile counts of 3, 4, and 1 are determined as a starting point for indices i, j and k, respectively.

From this starting point, the algorithm attempts to reduce the tilecount for each index by 1. It selects the index that provides the maximum improvement in disk cost and suffers the minimum penalty for memory cost. However, reducing the tile count for this index could cause the memory limit to be exceeded. The algorithm tries to repair

```
double MA(1:Ti,1:Nk)
double MB(1:Ti,1:Tj)
double MC(1:Tj,1:Nk)

FOR iT = 1, Ni, Ti
 MA(1:Ti,1:Nk) = Read disk array A(i,k)
 FOR jT = 1, Nj, Tj
  MB(1:Ti,1:Tj) = Read disk array B(i,j)
  MC(1:Tj,1:Nk) = Read disk array C(j,k)
  FOR kT = 1, Nk, Tk
   FOR iI = 1, Ti
    FOR jI = 1, Tj
     FOR kI = 1, Tk
      MA(1:Ti,1:Nk)
        += MB(1:Ti,1:Tj) * MC(1:Tj:1:Nk)
  END FOR kI, jI, iI, kT
 END FOR jT
 Write MA(1:Ti,1:Nk) to disk A(i,k)
END FOR iT
```

(a) Concrete code with I/O stmts moved outside.

Number of tiles			Block size			Memory Cost
i	*j*	*k*	*Ti*	*Tj*	*Tk*	
1	1	1	6000	2000	6000	457*MB*
⋮	⋮	⋮	⋮	⋮	⋮	⋮
4	4	4	1500	500	1500	97*MB*
3	4	4	2000	500	1500	122*MB*
3	4	3	2000	500	2000	122*MB*
⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	4	1	2000	500	6000	122*MB*

(b) Tile size search illustration.

Fig. 5. Concrete code and tile size search illustration for matrix multiplication

this situation by increasing the number of tiles for one of the other indices. It selects the index that provides the maximum effective improvement in disk access cost. If it cannot find such an index, the algorithm terminates.

5 Experimental Results

The algorithm from Sec. 4 was used to generate code for the AO-to-MO index transformation calculation described in Sec. 2. Measurements were made 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. Experiments were also carried out on more recent systems; while the overall trends are similar, we found significant variability in measured performance over multiple identical runs, due to disk activity from other users.

The out-of-core code generated by the proposed algorithm was compared with an unfused but tiled baseline version. With this version, it is necessary to write large intermediate arrays to disk when they are produced, and read them back again when they are consumed. The baseline version is representative of the current state of practice in implementing large out-of-core tensor contractions in state-of-the-art quantum chemistry packages.

Table 2 shows the measured I/O time for the unfused baseline version and the measured as well as predicted I/O time for the fused version for the case of unconstrained disk layout. For the baseline version, all arrays were assumed to be blocked on disk so that I/O was performed in large chunks in all steps. The size of the tensors(double precision) for the experiment were: $N_p = N_q = N_r = N_s = 80$ and $N_a = N_b = N_c = N_d = 70$.

	Unfused	T3 on disk	
	Measured time (seconds)	Measured time(seconds)	Predicted time(seconds)
Array A	42.4	43.648	42.6
Array t_1	195.67	-	-
Array t_2	66.74	-	-
Array t_3	48.26	74.09	74.32
Array B	29.078	22.346	38.6
Total time	382.15	140.08	155.52

Table 2. Predicted and Measured I/O Time: Unconstrained Layout

	Unfused	T1 on disk	
	Measured time (seconds)	Measured time(seconds)	Predicted time(seconds)
Array A	148.26	149.31	180.3
Array t_1	140.65	94.37	63.72
Array t_2	52.99	-	-
Array t_3	44.89	-	-
Array B	29.55	22.62	38.61
Total time	416.34	266.3	282.63

Table 3. Predicted and Measured I/O Time: Column-Major Layout of Input/Output Arrays

For these tensor sizes and an available memory of 128MB, 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 2 reports performance data for the fusion configuration that requires $T3$ to be disk-resident.

The I/O time for each array was separately accumulated. It can be seen that the out-of-core code version produced by the new algorithm has significantly lower disk I/O time than the baseline version. 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.

Table 3 shows performance data for the layout-constrained case. A column-major representation was used for the input array. For the fused/tiled version, we used the fusion configuration that results in $T1$ being disk-resident. Again, the version produced by the new algorithm is better than the baseline version. As can be expected, the disk I/O overheads for both versions are higher than the corresponding cases where the input array layout was unconstrained. The predicted and measured I/O times match to within 10%.

6 Conclusion

We have described an approach to the synthesis of out-of-core algorithms for a class of imperfectly nested loops. The approach was developed for the implementation in a component of a program synthesis system targeted at the quantum chemistry domain. However, 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. Experimental results were provided that showed a good match between predicted and measured performance. The performance achieved by the synthesized code was considerably better than that representative of codes incorporated into quantum chemistry packages today.

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. We would also like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

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, 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.
5. 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,* pp. 500–509, Sorrento, Italy, June 2001.
6. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, August 1999.
7. 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.
8. 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.
9. 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.
10. 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.