Optimization of Memory Usage Requirement for a Class of Loops Implementing Multi-Dimensional Integrals *

Chi-Chung Lam[1], Daniel Cociorva[2], Gerald Baumgartner[1], and P. Sadayappan[1]

[1] Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
{clam,gb,saday}@cis.ohio-state.edu
[2] Department of Physics
The Ohio State University, Columbus, OH 43210
cociorva@pacific.mps.ohio-state.edu

Abstract. Multi-dimensional integrals of products of several arrays arise in certain scientific computations. In the context of these integral calculations, this paper addresses a memory usage minimization problem. Based on a framework that models the relationship between loop fusion and memory usage, we propose an algorithm for finding a loop fusion configuration that minimizes memory usage. A practical example shows the performance improvement obtained by our algorithm on an electronic structure computation.

1 Introduction

This paper addresses the optimization of a class of loop computations that implement multi-dimensional integrals of the product of several arrays. Such integral calculations arise, for example, in the computation of electronic properties of semiconductors and metals [1, 7, 15]. The objective is to minimize the execution time of such computations on a parallel computer while staying within the available memory. In addition to the performance optimization issues pertaining to inter-processor communication and data locality enhancement, there is an opportunity to apply algebraic transformations using the properties of commutativity, associativity and distributivity to reduce the total number of arithmetic operations.

Given a specification of the required computation as a multi-dimensional sum of the product of input arrays, we first determine an equivalent sequence of multiplication and summation formulae that computes the result using a minimum number of arithmetic operations. Each formula computes and stores some intermediate results in an intermediate array. By computing the intermediate results once and reusing them multiple times, the number of arithmetic operations can be reduced. In previous work, this operation minimization problem was proved to be NP-complete and an efficient pruning search strategy was proposed [10].

The simplest way to implement an optimal sequence of multiplication and summation formulae is to compute the formulae one by one, each coded as a set of perfectly

* Supported in part by the National Science Foundation under grant DMR-9520319.

nested loops, and to store the intermediate results produced by each formula in an intermediate array. However, in practice, the input and intermediate arrays could be so large that they cannot fit into the available memory. Hence, there is a need to fuse the loops as a means of reducing memory usage. By fusing loops between the producer loop and the consumer loop of an intermediate array, intermediate results are formed and used in a pipelined fashion and they reuse the same reduced array space. The problem of finding a loop fusion configuration that minimizes memory usage without increasing the operation count is not trivial. In this paper, we develop an optimization framework that appropriately models the relation between loop fusion and memory usage. We present an algorithm that finds an optimal loop fusion configuration that minimizes memory usage.

Reduction of arithmetic operations has been traditionally done by compilers using the technique of common subexpression elimination [4]. Chatterjee et al. consider the optimal alignment of arrays in evaluating array expression on massively parallel machines [2, 3]. Much work has been done on improving locality and parallelism by loop fusion [8, 14, 16]. However, this paper considers a different use of loop fusion, which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. Traditional compiler research does not address this use of loop fusion because this problem does not arise with manually-produced programs. The contraction of arrays into scalars through loop fusion is studied in [6] but is motivated by data locality enhancement and not memory reduction. Loop fusion in the context of delayed evaluation of array expressions in APL programs is discussed in [5] but is also not aimed at minimizing array sizes. We are unaware of any work on fusion of multi-dimensional loop nests into imperfectly-nested loops as a means to reduce memory usage.

The rest of this paper is organized as follows. Section 2 describes the operation minimization problem. Section 3 studies the use of loop fusion to reduce array sizes and presents algorithms for finding an optimal loop fusion configuration that minimizes memory usage under the static memory allocation model. Due to space limitations, the extensions of the framework to the dynamic memory allocation model and to parallel machines are omitted in this paper but can be found in [13]. Section 4 provides conclusions.

2 Operation Minimization

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

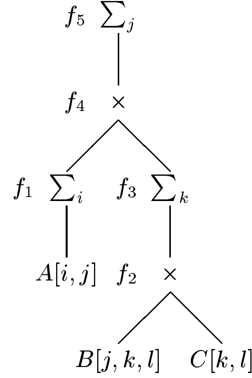
Consider, for example, the multi-dimensional integral shown in Figure 1(a). If implemented directly as expressed (i.e. as a single set of perfectly-nested loops), the computation would require $2 \times N_i \times N_j \times N_k \times N_l$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of the distributive

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

(a) A multi-dimensional integral

$$
\begin{aligned}
f_1[j] &= \textstyle\sum_i A[i,j] \\
f_2[j,k,l] &= B[j,k,l] \times C[k,l] \\
f_3[j,k] &= \textstyle\sum_l f_2[j,k,l] \\
f_4[j,k] &= f_1[j] \times f_3[j,k] \\
W[k] = f_5[k] &= \textstyle\sum_j f_4[j,k]
\end{aligned}
$$

(b) A formula sequence for computing (a)



(c) An expression tree representation of (b)

Fig. 1. An example multi-dimensional integral and two representations of a computation.

law of multiplication over addition is satisfactory for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $2 \times N_j \times N_k \times N_l + 2 \times N_j \times N_k + N_i \times N_j$ operations is given in Figure 1(b). It expresses the sequence of steps in computing the multi-dimensional integral as a sequence of formulae. Each formula computes some intermediate result and the last formula gives the final result. A formula is either a product of two input/intermediate arrays or a integral/summation over one index, of an input/intermediate array. A sequence of formulae can also be represented as an expression tree. For instance, Figure 1(c) shows the expression tree corresponding to the example formula sequence.

The problem of finding a formula sequence that minimizes the number of operations has been proved to be NP-complete [10]. A pruning search algorithm for finding such a formula sequence is given below.

1. Form a list of the product terms of the multi-dimensional integral. Let X_a denote the a-th product term and X_a.*dimens* the set of index variables in $X_a[...]$. Set r and c to zero. Set d to the number of product terms.
2. While there exists an summation index (say i) that appears in exactly one term (say $X_a[...]$) in the list and $a > c$, increment r and d and create a formula $f_r[...] = \sum_i X_a[...]$ where f_r.*dimens* $= X_a$.*dimens* $- \{i\}$. Remove $X_a[...]$ from the list. Append to the list $X_d[...] = f_r[...]$. Set c to a.
3. Increment r and d and form a formula $f_r[...] = X_a[...] \times X_b[...]$ where $X_a[...]$ and $X_b[...]$ are two terms in the list such that $a < b$ and $b > c$, and give priority to the terms that have exactly the same set of indices. The indices for f_r are f_r.*dimens* $= X_a$.*dimens* $\cup X_b$.*dimens*. Remove $X_a[...]$ and $X_b[...]$ from the list. Append to the list $X_d[...] = f_r[...]$. Set c to b. Go to step 2.
4. When steps 2 and 3 cannot be performed any more, a valid formula sequence is obtained. To obtain all valid sequences, exhaust all alternatives in step 3 using depth-first search.

3 Memory Usage Minimization

In implementing the computation represented by an operation-count-optimal formula sequence (or expression tree), there is a need to perform loop fusion to reduce the sizes of the arrays. Without fusing the loops, the arrays would be too large to fit into the available memory. There are many different ways to fuse the loops and they could result in different memory usage. This section addresses the problem of finding a loop fusion configuration for a given formula sequence that uses the least amount of memory. Section 3.1 introduces the memory usage minimization problem. Section 3.2 describes some preliminary concepts that we use to formulate our solutions to the problem. Sections 3.3 presents an algorithm for finding a memory-optimal loop fusion configuration under static memory allocation model. Section 3.4 illustrates how the application of the algorithm on an example physics computation improves its performance.

3.1 Problem Description

Consider again the expression tree shown in Figure 1(c). A naive way to implement the computation is to have a set of perfectly-nested loops for each node in the tree, as shown in Figure 2(a). The brackets indicate the scopes of the loops. Figure 2(b) shows how the sizes of the arrays may be reduced by the use of loop fusions. It shows the resulting loop structure after fusing all the loops between A and f_1, all the loops among B, C, f_2, and f_3, and all the loops between f_4 and f_5. Here, A, B, C, f_2, and f_4 are reduced to scalars. After fusing all the loops between a node and its parent, all dimensions of the child array are no longer needed and can be eliminated. The elements in the reduced arrays are now reused to hold different values at different iterations of the fused loops. Each of those values was held by a different array element before the loops were fused (as in Figure 2(a)). Note that some loop nests (such as those for B and C) are reordered and some loops within loop nests (such as the j-, k-, and l-loops for B, f_2, and f_3) are permuted in order to facilitate loop fusions.

For now, we assume the leaf node arrays (i.e., input arrays) can be generated one element at a time (by the gen$v$ function for array v) so that loop fusions with their parents are allowed. This assumption holds for arrays in which the value of each element is a function of the array subscripts, as in many arrays in the physics computations that we work on. As will be clear later on, the case where an input array has to be read in or produced in slices or in its entirety can be handled by disabling the fusion of some or all the loops between the leaf node and its parent.

Figure 2(c) shows another possible loop fusion configuration obtained by fusing all the j-loops and then all the k-loops and l-loops inside them. The sizes of all arrays except C and f_5 are smaller. By fusing the j-, k-, and l-loops between those nodes, the j-, k-, and l-dimensions of the corresponding arrays can be eliminated. Hence, B, f_1, f_2, f_3, and f_4 are reduced to scalars while A becomes a one-dimensional array.

In general, fusing a t-loop between a node v and its parent eliminates the t-dimension of the array v and reduces the array size by a factor of N_t. In other words, the size of an array after loop fusions equals the product of the ranges of the loops that are not fused with its parent. We only consider fusions of loops among nodes that are all transitively

```
for i
  for j
    A[i,j]=genA(i,j)
for j
  for k
    for l
      B[j,k,l]=genB(j,k,l)
for k
  for l
    C[k,l]=genC(k,l)
initialize f1
for i
  for j
    f1[j]+=A[i,j]
for j
  for k
    for l
      f2[j,k,l]=B[j,k,l]×C[k,l]
initialize f3
for j
  for k
    for l
      f3[j,k]+=f2[j,k,l]
for j
  for k
    f4[j,k]=f1[j]×f3[j,k]
initialize f5
for j
  for k
    f5[k]+=f4[j,k]
```

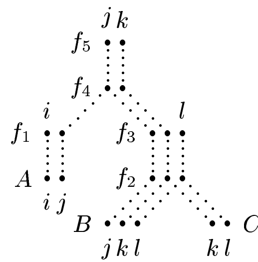
(a)

```
initialize f1
for i
  for j
    A=genA(i,j)
    f1[j]+=A
initialize f3
for k
  for l
    C=genC(k,l)
    for j
      B=genB(j,k,l)
      f2=B×C
      f3[j,k]+=f2
initialize f5
for j
  for k
    f4=f1[j]×f3[j,k]
    f5[k]+=f4
```

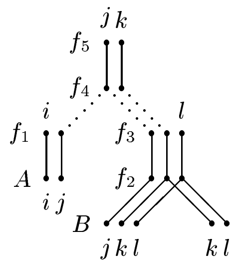
(b)

```
for k
  for l
    C[k,l]=genC(k,l)
initialize f5
for j
  for i
    A[i]=genA(i,j)
  initialize f1
  for i
    f1+=A[i]
  for k
    initialize f3
    for l
      B=genB(j,k,l)
      f2=B×C[k,l]
      f3+=f2
    f4=f1×f3
    f5[k]+=f4
```

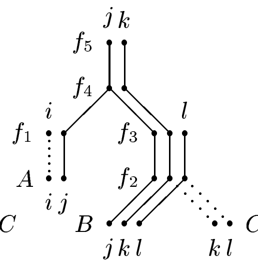
(c)



(d)



(e)



(f)

Fig. 2. Three loop fusion configurations for the expression tree in Figure 1.

related by (i.e., form a transitive closure over) parent-child relations. Fusing loops between unrelated nodes (such as fusing siblings without fusing their parent) has no effect on array sizes. We also restrict our attention for now to loop fusion configurations that do not increase the operation count.

In the class of loops considered in this paper, the only dependence relations are those between children and parents, and array subscripts are simply loop index variables[1]. So, loop permutations, loop nests reordering, and loop fusions are always legal as long as child nodes are evaluated before their parents. This freedom allows the loops to be permuted, reordered, and fused in a large number of ways that differ in memory usage. Finding a loop fusion configuration that uses the least memory is not trivial. We believe this problem is NP-complete but have not found a proof yet.

Fusion graphs. Let T be an expression tree. For any given node $v \in T$, let *subtree*(v) be the set of nodes in the subtree rooted at v, v.*parent* be the parent of v, and v.*indices* be the set of loop indices for v (including the summation index v.*sumindex* if v is a summation node). A loop fusion configuration can be represented by a graph called a *fusion graph*, which is constructed from T as follows.

1. Replace each node v in T by a set of vertices, one for each index $i \in v$.*indices*.
2. Remove all tree edges in T for clarity.
3. For each loop fused (say, of index i) between a node and its parent, connect the i-vertices in the two nodes with a *fusion edge*.
4. For each pair of vertices with matching index between a node and its parent, if they are not already connected with a fusion edge, connect them with a *potential fusion edge*.

Figure 2 shows the fusion graphs alongside the loop fusion configurations. In the figure, solid lines are fusion edges and dotted lines are potential fusion edges, which are fusion opportunities not exploited. As an example, consider the loop fusion configuration in Figure 2(b) and the corresponding fusion graph in Figure 2(e). Since the j-, k-, and l-loops are fused between f_2 and f_3, there are three fusion edges, one for each of the three loops, between f_2 and f_3 in the fusion graph. Also, since no loops are fused between f_3 and f_4, the edges between f_3 and f_4 in the fusion graph remain potential fusion edges.

In a fusion graph, we call each connected component of fusion edges (i.e., a maximal set of connected fusion edges) a *fusion chain*, which corresponds to a fused loop in the loop structure. The *scope of a fusion chain c*, denoted *scope*(c), is defined as the set of nodes it spans. In Figure 2(f), there are three fusion chains, one for each of the j-, k-, and l-loops; the scope of the shortest fusion chain is $\{B, f_2, f_3\}$. The scope of any two fusion chains in a fusion graph must either be disjoint or a subset/superset of each other. Scopes of fusion chains do not partially overlap because loops do not (i.e., loops must be either separate or nested). Therefore, any fusion graph with fusion chains whose scopes are partially overlapping is illegal and does not correspond to any loop fusion configuration.

[1] When array subscripts are not simple loop variables, as many researchers have studied, more dependence relations exist, which prevent some loop rearrangement and/or loop fusions. In that case, a restricted set of loop fusion configurations would need to be searched in minimizing memory usage.

Fusion graphs help us visualize the structure of the fused loops and find further fusion opportunities. If we can find a set of potential fusion edges that, when converted to fusion edges, does not lead to partially overlapping scopes of fusion chains, then we can perform the corresponding loop fusions and reduce the size of some arrays. For example, the i-loops between A and f_1 in Figure 2(f) can be further fused and array A would be reduced to a scalar. If converting all potential fusion edges in a fusion graph to fusion edges does not make the fusion graph illegal, then we can completely fuse all the loops and achieve optimal memory usage. But for many fusion graphs in real-life loop configurations (including the ones in Figure 2), this does not hold. Instead, potential fusion edges may be mutually prohibitive; fusing one loop could prevent the fusion of another. In Figure 2(e), fusing the j-loops between f_1 and f_4 would disallow the fusion of the k-loops between f_3 and f_4. Although a fusion graph specifies what loops are fused, it does not fully determine the permutations of the loops and the ordering of the loop nests.

3.2 Preliminaries

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

Scope and fusion scope of a loop. The *scope of a loop* of index i in a subtree rooted at v, denoted $\mathit{scope}(i, v)$, is defined in the usual sense as the set of nodes in the subtree that the fused loop spans. That is, if the i-loop is fused, $\mathit{scope}(i, v) = \mathit{scope}(c) \cap \mathit{subtree}(v)$, where c is a fusion chain for the i-loop with $v \in \mathit{scope}(c)$. If the i-loop of v is not fused, then $\mathit{scope}(i, v) = \emptyset$. We also define the *fusion scope of an i-loop* in a subtree rooted at v as $\mathit{fscope}(i, v) = \mathit{scope}(i, v)$ if the i-loop is fused between v and its parent; otherwise $\mathit{fscope}(i, v) = \emptyset$. As an example, for the fusion graph in Figure 2(e), $\mathit{scope}(j, f_3) = \{B, f_2, f_3\}$, but $\mathit{fscope}(j, f_3) = \emptyset$.

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 $|g|$ the number of indexsets in g, $g[r]$ the r-th indexset in g, and $\mathit{Set}(g)$ the union of all indexsets in g, i.e. $\mathit{Set}(g) = \bigcup_{1 \le r \le |g|} g[r]$. For instance, $|f| = 2$, $f[1] = \{i, k\}$, and $\mathit{Set}(f) = \mathit{Set}(\langle j, i, k\rangle) = \{i, j, k\}$. We say that g' is a *prefix* of g if $|g'| \le |g|$, $g'[|g'|] \subseteq g[|g'|]$, and for all $1 \le r < |g'|$, $g'[r] = g[r]$. We write this relation as $\mathit{prefix}(g', g)$. So, $\langle\rangle$, $\langle i\rangle$, $\langle k\rangle$, $\langle ik\rangle$, $\langle ik, j\rangle$ are prefixes of f, but $\langle i, j\rangle$ is not. The *concatenation* of g and an indexset x, denoted $g + x$, is defined as the indexset sequence g'' such that if $x \ne \emptyset$, then $|g''| = |g| + 1$, $g''[|g''|] = x$, and for all $1 \le r < |g''|$, $g''[r] = g[r]$; otherwise, $g'' = g$.

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). For example, in Figure 2(f), the fusion between f_2

and f_3 is $\langle jkl \rangle$ and the fusion between f_4 and f_5 is $\langle j, k \rangle$ (because the fused j-loop covers two more nodes, A and f_1). Formally, a fusion between a node v and v.*parent* is an indexset sequence f such that

1. $\mathit{Set}(f) \subseteq v.\mathit{indices} \cap v.\mathit{parent}.\mathit{indices}$,
2. for all $i \in \mathit{Set}(f)$, the i-loop is fused between v and v.*parent*, and
3. for all $i \in f[r]$ and $i' \in f[r']$,
   (a) $r = r'$ iff $\mathit{fscope}(i, v) = \mathit{fscope}(i', v)$, and
   (b) $r < r'$ iff $\mathit{fscope}(i, v) \supset \mathit{fscope}(i', v)$.

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. For example, in Figure 2(e), the loop nesting at f_3 is $\langle kl, j \rangle$, at f_4 is $\langle jk \rangle$, and at B is $\langle jkl \rangle$. Formally, a nesting of the loops at a node v is an indexset sequence h such that

1. $\mathit{Set}(h) = v.\mathit{indices}$ and
2. for all $i \in h[r]$ and $i' \in h[r']$,
   (a) $r = r'$ iff $\mathit{scope}(i, v) = \mathit{scope}(i', v)$, and
   (b) $r < r'$ iff $\mathit{scope}(i, v) \supset \mathit{scope}(i', v)$.

By definition, the loop nesting at a leaf node v must be $\langle v.\mathit{indices} \rangle$ because all loops at v have empty scope.

Legal fusion. A legal fusion graph (corresponding to a loop fusion configuration) for an expression tree T can be built up in a bottom-up manner by extending and merging legal fusion graphs for the subtrees of T. For a given node v, the nesting h at v summarizes the fusion graph for the subtree rooted at v and determines what fusions are allowed between v and its parent. A fusion f is legal for a nesting h at v if $\mathit{prefix}(f, h)$ and $\mathrm{set}(f) \subseteq v.\mathit{parent}.\mathit{indices}$. This is because, to keep the fusion graph legal, loops with larger scopes must be fused before fusing those with smaller scopes, and only loops common to both v and its parent may be fused. For example, consider the fusion graph for the subtree rooted at f_2 in Figure 2(e). Since the nesting at f_2 is $\langle kl, j \rangle$ and $f_3.\mathit{indices} = \{j, k, l\}$, the legal fusions between f_2 and f_3 are $\langle \rangle$, $\langle k \rangle$, $\langle l \rangle$, $\langle kl \rangle$, and $\langle kl, j \rangle$. Notice that all legal fusions for a node v are prefixes of a *maximal legal fusion*, which can be expressed as $\mathit{MaxFusion}(h, v) = \max\{f \mid \mathit{prefix}(f, h) \text{ and } \mathrm{set}(f) \subseteq v.\mathit{parent}.\mathit{indices}\}$, where h is the nesting at v. In Figure 2(e), the maximal legal fusion for C is $\langle kl \rangle$, and for f_2 is $\langle kl, j \rangle$.

Resulting nesting. Let u be the parent of a node v. If v is the only child of u, then the loop nesting at u as a result of a fusion f between u and v can be obtained by the function $\mathit{ExtNesting}(f, u) = f + (u.\mathit{indices} - \mathit{Set}(f))$. For example, in Figure 2(e), if the fusion between f_2 and f_3 is $\langle kl \rangle$, then the nesting at f_3 would be $\langle kl, j \rangle$.

Compatible nestings. Suppose v has a sibling v', f is the fusion between u and v, and f' is the fusion between u and v'. For the fusion graph for the subtree rooted at u (which is merged from those of v and v') to be legal, $h = \mathit{ExtNesting}(f, u)$ and $h' = \mathit{ExtNesting}(f', u)$ must be *compatible* according to the condition: for all $i \in h[r]$ and $j \in h[s]$, if $r < s$ and $i \in h'[r']$ and $j \in h'[s']$, then $r' \leq s'$. This requirement ensures an i-loop that has a larger scope than a j-loop in one subtree will not have a smaller

scope than the j-loop in the other subtree. If h and h' are compatible, the resulting loop nesting at u (as merged from h and h') is h'' such that for all $i \in h''[r'']$ and $j \in h''[s'']$, if $i \in h[r]$, $i \in h'[r']$, $j \in h[s]$, and $j \in h'[s']$, then $[r'' = s'' \Rightarrow r = s$ and $r' = s']$ and $[r'' \leq s'' \Rightarrow r \leq s$ and $r' \leq s']$. Effectively, the loops at u are re-ranked by their combined scopes in the two subtrees to form h''. As an example, in Figure 2(e), if the fusion between f_1 and f_4 is $f = \langle j \rangle$ and the fusion between f_3 and f_4 is $f' = \langle k \rangle$, then $h = \mathit{ExtNesting}(f, f_4) = \langle j, k \rangle$ and $h' = \mathit{ExtNesting}(f', f_4) = \langle k, j \rangle$ would be incompatible. But if f is changed to $\langle \rangle$, then $h = \mathit{ExtNesting}(f, f_4) = \langle jk \rangle$ would be compatible with h', and the resulting nesting at f_4 would be $\langle k, j \rangle$. A procedure for checking if h and h' are compatible and forming h'' from h and h' is provided in Section 3.3.

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 for all legal fusion graph G for T in which the nesting at v is h, there exists a legal fusion graph G' for T in which the nesting at v is h' such that the subgraphs of G and G' induced by $T - \mathit{subtree}(v)$ are identical. In other words, $h \sqsubseteq h'$ means that 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 in Section 3.3. It can be proved that the necessary and sufficient condition for $h \sqsubseteq h'$ is that for all $i \in m[r]$ and $j \in m[s]$, there exist r', s' such that $i \in m'[r']$ and $j \in m'[s']$ and $[r = s \Rightarrow r' = s']$ and $[r < s \Rightarrow r' \leq s']$, where $m = \mathit{MaxFusion}(h, v)$ and $m' = \mathit{MaxFusion}(h', v)$. Comparing the nesting at f_3 between Figure 2(e) and (f), the nesting $\langle kl, j \rangle$ in (e) is more constraining than the nesting $\langle jkl \rangle$ in (f). A procedure for determining if $h \sqsubseteq h'$ is given in Section 3.3.

3.3 Static Memory Allocation

Under the static memory allocation model, since all the arrays in a program exist during the entire computation, the memory usage of a loop fusion configuration is simply the sum of the sizes of all the arrays (including those reduced to scalars). Figures 3 shows a bottom-up, dynamic programming algorithm for finding a memory-optimal loop fusion configuration for a given expression tree T. For each node v in t, it computes a set of solutions $v.\mathit{solns}$ for the subtree rooted at v. Each solution s in $v.\mathit{solns}$ represents a loop fusion configuration for the subtree rooted at v and contains the following information for s: the loop nesting $s.\mathit{nesting}$ at v, the fusion $s.\mathit{fusion}$ between v and its parent, the memory usage $s.\mathit{mem}$ so far, and the pointers $s.\mathit{src1}$ and $s.\mathit{src2}$ to the corresponding solutions for the children of v.

The set of solutions $v.\mathit{solns}$ is obtained by the following steps. First, if v is a leaf node, initialize the solution set to contain a single solution using **InitSolns**. Otherwise, take the solution set from a child $v.\mathit{child1}$ of v, and, if v has two children, merge it (using **MergeSolns**) with the compatible solutions from the other child $v.\mathit{child2}$. Then, prune the solution set to remove the inferior solutions using **PruneSolns**. A solution s is *inferior* to another unpruned solution s' if $s.\mathit{nesting}$ is more or equally constraining than $s'.\mathit{nesting}$ and s does not use less memory than s'. Next, extend the solution set by considering all possible legal fusions between v and its parent (see **ExtSolns**). The size

```
MinMemFusion (T):
    InitFusible (T)
    foreach node v in some bottom-up traversal of T
      if v.nchildren = 0 then
        S1 = InitSolns (v)
      else
        S1 = v.child1.solns
        if v.nchildren = 2 then
          S1 = MergeSolns (S1, v.child2.solns)
        S1 = PruneSolns (S1, v)
      v.solns = ExtendSolns (S1, v)
    T.root.optsoln = the single element in T.root.solns
    foreach node v in some top-down traversal of T
      s = v.optsoln
      v.optfusion = s.fusion
      s1 = s.src1
      if v.nchildren = 1 then
        v.child1.optsoln = s1
      else
        v.child1.optsoln = s1.src1
        v.child2.optsoln = s1.src2

InitFusible (T):
    foreach v ∈ T
      if v = T.root then
        v.fusible = ∅
      else
        v.fusible = v.indices ∩ v.parent.indices

InitSolns (v):
    s.nesting = ⟨v.fusible⟩
    InitMemUsage (s)
    return {s}

MergeSolns (S1, S2):
    S = ∅
    foreach s1 ∈ S1
      foreach s2 ∈ S2
        s.nesting = MergeNesting (s1.nesting, s2.nesting)
        if s.nesting ≠ ⟨⟩ then   // if s1 and s2 are compatible
          s.src1 = s1
          s.src2 = s2
          MergeMemUsage (s1, s2, s)
          AddSoln (s, S)
    return S

PruneSolns (S1, v):
    S = ∅
    foreach s1 ∈ S1
      s.nesting = MaxFusion (s1.nesting, v)
      AddSoln (s, S)
    return S

ExtendSolns (S1, v):
    S = ∅
    foreach s1 ∈ S1
      foreach prefix f of s1.nesting
        s.fusion = f
        s.nesting = ExtNesting (f, v.parent)
        s.src1 = s1
        size = FusedSize (v, f)
        AddMemUsage (v, f, size, s1, s)
        AddSoln (s, S)
    return S
```

```
AddSoln (s, S):
    foreach s′ ∈ S
      if Inferior (s, s′) then
        return
      else if Inferior (s′, s) then
        S = S − {s′}
    S = S ∪ {s}

MergeNesting (h, h′):
    g = ⟨⟩
    r = r′ = 1
    x = x′ = ∅
    while r ≤ |h| or r′ ≤ |h′|
      if x = ∅ then
        x = h[r++]
      if x′ = ∅ then
        x′ = h′[r′++]
      y = x ∩ x'
      if y = ∅ then
        return ⟨⟩   // h and h′ are incompatible
      g = g + y
      y = x − x′
      x′ = x′ − x
      x = y
    end while
    return g   // h and h′ are compatible

h ⊑ h′:   // test if h is more/equally constraining than h′
    r′ = 1
    x′ = ∅
    for r = 1 to |h|
      if x′ = ∅ then
        if r′ > |h′| then
          return false
        x′ = h′[r′++]
      if h[r] ⊈ x′ then
        return false
      x′ = x′ − h[r]
    return true

InitMemUsage (s):
    s.mem = 0

AddMemUsage (v, f, size, s1, s):
    s.mem = s1.mem + size

MergeMemUsage (s1, s2, s):
    s.mem = s1.mem + s2.mem

Inferior (s, s′) ≡
    s.nesting ⊑ s′.nesting and s.mem ≥ s′.mem

FusedSize (v, f) ≡
    ∏_(i ∈ v.indices − {v.sumindex} − f) N_i

ExtNesting (f, u) ≡ f + (u.indices − Set (f))

MaxFusion (h, v) ≡
    max{f | prefix(f, h) and Set (f) ⊆ v.parent.indices}

Set (f) ≡ ⋃_(1 ≤ r ≤ |f|) f[r]
```

Fig. 3. Algorithm for static memory allocation.

of array v is added to memory usage by **AddMemUsage**. Inferior solutions are also removed.

Although the complexity of the algorithm is exponential in the number of index variables and the number of solutions could in theory grow exponentially with the size of the expression tree, the number of index variables in practical applications is usually small and there is indication that the pruning is effective in keeping the size of the solution set in each node small.

The algorithm assumes the leaf nodes may be freely fused with their parents and the root node array must be available in its entirety at the end of the computation. If these assumptions do not hold, the **InitFusible** procedure can be easily modified to restrict or expand the allowable fusions for those nodes.

v	line	src	nesting	fusion	ext-nest	memory usage	opt
A	1		$\langle ij \rangle$	$\langle ij \rangle$	$\langle ij \rangle$	1	$\surd$
B	2		$\langle jkl \rangle$	$\langle jkl \rangle$	$\langle jkl \rangle$	1	$\surd$
C	3		$\langle kl \rangle$	$\langle kl \rangle$	$\langle kl, j \rangle$	1	
	4		$\langle kl \rangle$	$\langle k \rangle$	$\langle k, jl \rangle$	15	$\surd$
	5		$\langle kl \rangle$	$\langle l \rangle$	$\langle l, jk \rangle$	40	
	6		$\langle kl \rangle$	$\langle \rangle$	$\langle jkl \rangle$	600	
f_1	7	1	$\langle ij \rangle$	$\langle j \rangle$	$\langle j, k \rangle$	1+1=2	
	8	1	$\langle ij \rangle$	$\langle \rangle$	$\langle jk \rangle$	1+100=101	$\surd$
f_2	9	2,3	$\langle kl, j \rangle$	$\langle kl, j \rangle$	$\langle kl, j \rangle$	(1+1)+1=3	
	10	2,4	$\langle k, jl \rangle$	$\langle k, jl \rangle$	$\langle k, jl \rangle$	(1+15)+1=17	$\surd$
	11	2,5	$\langle l, jk \rangle$	$\langle l, jk \rangle$	$\langle l, jk \rangle$	(1+40)+1=42	
	12	2,6	$\langle jkl \rangle$	$\langle jkl \rangle$	$\langle jkl \rangle$	(1+600)+1=602	
f_3	13	10	$\langle k, jl \rangle$	$\langle k, j \rangle$	$\langle k, j \rangle$	17+1=18	$\surd$
	14	12	$\langle jkl \rangle$	$\langle jk \rangle$	$\langle jk \rangle$	602+1=603	
f_4	15	7,14	$\langle j, k \rangle$	$\langle j, k \rangle$	$\langle j, k \rangle$	(2+603)+1=606	
	16	8,13	$\langle k, j \rangle$	$\langle k, j \rangle$	$\langle k, j \rangle$	(101+18)+1=120	$\surd$
	17	8,14	$\langle jk \rangle$	$\langle jk \rangle$	$\langle jk \rangle$	(101+603)+1=705	
f_5	18	16	$\langle k, j \rangle$	$\langle \rangle$	$\langle \rangle$	120+40=160	$\surd$

Fig. 4. Solution sets for the subtrees in the example.

To illustrate how the algorithm works, consider again the empty fusion graph in Figure 2(d) for the expression tree in Figure 1(c). Let $N_i = 500$, $N_j = 100$, $N_k = 40$, and $N_l = 15$. There are $2^3 = 8$ different fusions between B and f_2. Among them, only the full fusion $\langle jkl \rangle$ is in B.*solns* because all other fusions result in more constraining nestings and use more memory than the full fusion and are pruned. However, this does not happen to the fusions between C and f_2 since the resulting nesting $\langle kl, j \rangle$ of the full fusion $\langle kl \rangle$ is not less constraining than those of the other 3 possible fusions. Then, solutions from B and C are merged together to form solutions for f_2. For example, when the two full-fusion solutions from B and C are merged, the merged nesting for f_2 is $\langle kl, j \rangle$, which can then be extended by full fusion (between f_2 and f_3) to form a

full-fusion solution for f_2 that has a memory usage of only 3 scalars. Again, since this solution is not the least constraining one, other solutions cannot be pruned. Although this solution is optimal for the subtree rooted at f_2, it turns out to be non-optimal for the entire expression tree. Figure 4 shows the solution sets for all of the nodes. The "src" column contains the line numbers of the corresponding solutions for the children. The "ext-nest" column shows the resulting nesting for the parent. A $\surd$ mark indicates the solution forms a part of an optimal solution for the entire expression tree. The fusion graph for the optimal solution is shown in Figure 5(a).

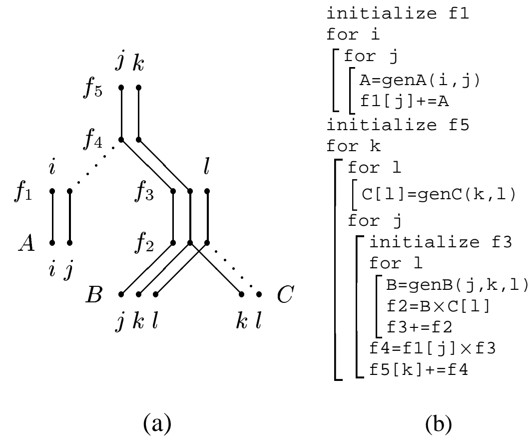


Fig. 5. An optimal solution for the example.

Once an optimal solution is obtained, we can generate the corresponding fused loop structure from it. The following procedure determines an evaluation order of the nodes:

1. Initialize set P to contain a single node T.*root* and list L to an empty list.
2. While P is not empty, remove from P a node v where v.*optfusion* is maximal among all nodes in P, insert v at the beginning of L, and add the children of v (if any) to P.
3. When P is empty, L contains the evaluation order.

Putting the loops around the array evaluation statements is trivial. The initialization of an array can be placed inside the innermost loop that contains both the evaluation and use of the array. The optimal loop fusion configuration for the example expression tree is shown in Figure 5(b).

3.4 An Example

We illustrate the practical application of the memory usage minimization algorithm on the following example formula sequence which can be used to determine self-energy in electronic structure of solids. It is optimal in operation count and has a cost of $1.89 \times$

10^{15} operations. The ranges of the indices are $N_k = 10$, $N_t = 100$, $N_{RL} = N_{RL1} = N_{RL2} = N_G = N_{G1} = 1000$, and $N_r = N_{r1} = 100000$.

```
f1[r,RL,RL1,t] = Y[r,RL] * G[RL1,RL,t]
f2[r,RL1,t] = sum RL f1[r,RL,RL1,t]
f5[r,RL2,r1,t] = Y[r,RL2] * f2[r1,RL2,t]
f6[r,r1,t] = sum RL2 f5[r,RL2,r1,t]
f7[k,r,r1] = exp[k,r] * exp[k,r1]
f10[r,r1,t] = f6[r,r1,t] * f6[r1,r,t]
f11[k,r,r1,t] = f7[k,r,r1] * f10[r,r1,t]
f13[k,r1,t,G] = fft r f11[k,r,r1,t] * exp[G,r]
f15[k,t,G,G1] = fft r1 f13[k,r1,t,G] * exp[G1,r1]
```

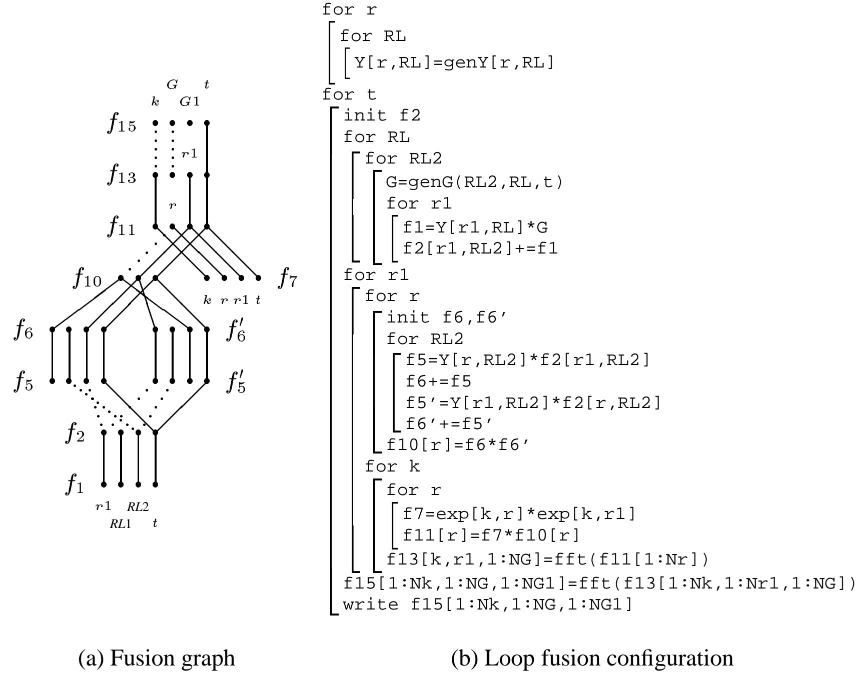
In this example, array Y is sparse and has only 10% non-zero elements. Notice that the common sub-expressions Y, exp, and f_6 appear at the right hand side of more than one formula. Also, f_{13} and f_{15} are fast Fourier transform formulae. Discussions on how to handle sparse arrays, common sub-expressions, and fast Fourier transforms can be found in [12, 13].

Without any loop fusion, the total size of the arrays is 1.13×10^{14} elements. If each array is to be computed only once, the presence of the common sub-expressions and FFTs would prevent the fusion of some loops, such as the r and $r1$ loops between f_6 and f_{10}. Under the operation-count restriction, the optimal loop fusion configuration obtained by the memory usage minimization algorithm for static memory allocation requires memory storage for 1.10×10^{11} array elements, which is 1000 times better than without any loop fusion. But this translates to about 1,000 gigabytes and probably still exceeds the amount of memory available in any computer today. Thus, relaxation of the operation-count restriction is necessary to further reduce to memory usage to reasonable values. Discussions on heuristics for trading arithmetic operations for memory can be found in [13].

We perform the following simple transformations to the DAG and the corresponding fusion graph.

- Two additional vertices are added: one for a k-loop around f_{10} and the other for a t-loop around f_7. These additional vertices are then connected to the corresponding vertices in f_{11} with additional potential fusion edges to allow more loop fusion opportunities between f_{11} and its two children.
- The common sub-expressions Y, exp, and f_6 are split into multiple nodes. Also, two copies of the sub-DAG rooted at f_6 are made. This will overcome some constraints on legal fusion graphs for DAGs.

The memory usage minimization algorithm for static memory allocation is then applied on the transformed fusion graph. The fusion graph and the loop fusion configuration for the optimal solution found are shown in Figure 6. For clarity, the input arrays are not included in the fusion graph. The memory usage of the optimal solution after relaxing the operation-count restriction is significantly reduced by a factor of about 100 to 1.12×10^9 array elements. The operation count is increased by only around 10% to 2.10×10^{15}. The best hand-optimized loop fusion configuration produced by domain experts also has some manually-applied transformations to reduce memory usage to



(a) Fusion graph (b) Loop fusion configuration

Fig. 6. Optimal loop fusions for the example formula sequence.

1.12×10^9 array elements and has 5.08×10^{15} operations. In comparison, the optimal loop fusion configuration obtained by the algorithm shows a factor of 2.5 improvement in operation count while using the same amount of memory.

4 Conclusion

In this paper, we have considered an optimization problem motivated by some computational physics applications. The computations are essentially multi-dimensional integrals of the product of several arrays. In practice, the input arrays and intermediate arrays could be too large to fit into the available amount of memory. It becomes necessary to fuse the loops to eliminate some dimensions of the arrays and reduce memory usage. The problem of finding a loop fusion configuration that minimizes memory usage was addressed in this paper. Based on a framework that models loop fusions and memory usage, we have presented an algorithm that solves the memory optimization problem.

Work is in progress on the optimization and the implementation of the algorithms in this paper. We also plan on developing an automatic code generator that takes array partitioning and loop fusion information as input and produces the source code of a parallel program that computes the desired multi-dimensional integral.

References

1. W. Aulbur, *Parallel implementation of quasiparticle calculations of semiconductors and insulators*, Ph.D. Dissertation, Ohio State University, Columbus, October 1996.
2. S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng, *Automatic array alignment in data-parallel programs*, 20th Annual ACM SIGACTS/SIGPLAN Symposium on Principles of Programming Languages, New York, pp. 16–28, 1993.
3. S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng, *Optimal evaluation of array expressions on massively parallel machines*, ACM TOPLAS, 17 (1), pp. 123–156, Jan. 1995.
4. C. N. Fischer and R. J. LeBlanc Jr, *Crafting a compiler*, Menlo Park, CA:Benjamin/Cummings, 1991.
5. L. J. Guibas and D. K. Wyatt, *Compilation and Delayed Evaluation in APL*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, pp. 1–8, Jan. 1978.
6. G. Gao, R. Olsen, V. Sarkar, and R. Thekkath, *Collective loop fusion for array contraction*, Languages and Compilers for Parallel Processing, New Haven, CT, August 1992.
7. M. S. Hybertsen and S. G. Louie, *Electronic correlation in semiconductors and insulators: band gaps and quasiparticle energies*, Phys. Rev. B, 34 (1986), pp. 5390.
8. K. Kennedy and K. S. McKinley, *Maximizing loop parallelism and improving data locality via loop fusion and distribution*, Languages and Compilers for Parallel Computing, Portland, OR, pp. 301–320, August 1993.
9. C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, *Memory-optimal evaluation of expression trees involving large objects*, Technical report no. OSU-CISRC-5/99-TR13, Dept. of Computer and Information Science, The Ohio State University, May 1999.
10. C. Lam, P. Sadayappan, and R. Wenger, *On optimizing a class of multi-dimensional loops with reductions for parallel execution*, Parallel Processing Letters, Vol. 7 No. 2, pp. 157–168, 1997.
11. C. Lam, P. Sadayappan, and R. Wenger, *Optimization of a class of multi-dimensional integrals on parallel machines*, Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997.
12. C. Lam, P. Sadayappan, D. Cociorva, M. Alouani, and J. Wilkins, *Performance optimization of a class of loops involving sums of products of sparse arrays*, Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, March 1999.
13. C. Lam, *Performance optimization of a class of loops implementing multi-dimensional integrals*, Technical report no. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University, Columbus, August 1999.
14. N. Manjikian and T. S. Abdelrahman, *Fusion of Loops for Parallelism and Locality*, International Conference on Parallel Processing, pp. II:19–28, Oconomowoc, WI, August 1995.
15. H. N. Rojas, R. W. Godby, and R. J. Needs, *Space-time method for Ab-initio calculations of self-energies and dielectric response functions of solids*, Phys. Rev. Lett., 74 (1995), pp. 1827.
16. S. Singhai and K. MacKinley, *Loop Fusion for Data Locality and Parallelism*, Mid-Atlantic Student Workshop on Programming Languages and Systems, SUNY at New Paltz, April 1996.