Practical Loop Transformations for Tensor Contraction Expressions on Multi-level Memory Hierarchies

Wenjing Ma¹, Sriram Krishnamoorthy², and Gagan Agrawal¹

¹ The Ohio State University, Columbus, OH, 43210, USA
{mawe, agrawal}@cse.ohio-state.edu

² Pacific Northwest National Lab, Richland, WA, 99352, USA
sriram@pnl.gov

Abstract. Modern architectures are characterized by deeper levels of memory hierarchy, often explicitly addressable. Optimizing applications for such architectures requires careful management of the data movement across all these levels. In this paper, we focus on the problem of mapping tensor contractions to memory hierarchies with more than two levels, specifically addressing placement of memory allocation and data movement statements, choice of loop fusions, and tile size selection. Existing algorithms to find an integrated solution to this problem even for two-level memory hierarchies have been shown to be expensive. We improve upon this work by focusing on the first-order cost components, simplifying the analysis required and reducing the number of candidates to be evaluated. We have evaluated our framework on a cluster of GPUs. Using five candidate tensor contraction expressions, we show that fusion at multiple levels improves performance, and our framework is effective in determining profitable transformations.

1 Introduction

Starting from the last 4-5 years, chip designers cannot enable sequential code to be executed faster through increases in clock rates and/or instruction-level parallelism. This paradigm shift is impacting high performance computing as well, as clusters of uniprocessors are no longer the dominant HPC architectures. Instead, clusters of multi-core and/or many-core architectures are becoming extremely common, and overall, architectures are becoming increasingly heterogeneous and complex. One emerging trend in HPC is towards clusters of GPUs. In the list of top 500 supercomputers released in May 2010, two of the top seven fastest supercomputers are based on GPUs. Because of the overall popularity and cost-effectiveness of GPUs, this trend can be expected to continue.

Developing applications on these emerging systems involves complex optimizations. Among other issues, orchestrating data movement across multiple levels of hierarchy, and restructuring the computation to reduce the cost of such data movement, are both challenging problems. As a specific example, consider processing of a disk-resident dataset on a machine with one or more GPUs. If

the size of the main memory is smaller than the size of the dataset, there will be significant data movement costs between the disk and main memory. Similarly, it is likely that the GPU device memory is smaller than the main memory. Even though GPU memory capacities have increased rapidly, high-end GPUs today have 3-6 GB memory, and cheaper or older GPUs have much less memory. In comparison, servers today easily have 12 or more GB of main memory. Thus, data movements between main memory and device memory also need to be carefully planned. Moreover, when considering a cluster of GPUs and/or a single node connected with multiple GPUs, the data movement and optimization problems are even more challenging.

Optimizations such as tiling and loop fusion can enable better data reuse and reduce the cost of these data movements. However, what we need is an integrated framework to enable these transformations across more than 2 levels of memory hierarchy. There has been a considerable amount of work on compiler optimizations for out-of-core programs [6,18,17,5,36], but the solutions do not directly extend when data movements for another level need to be considered. Similarly, there has been some work on optimizing data movements from main memory to device memory [35,34,22]. As multi-level processor caches became very common in mid-nineties, several compiler efforts considered optimizations for them [25,32,30].

This paper presents an optimization framework for systems requiring data movement across multiple memory hierarchy levels. Our work is specifically in the context of tensor contractions [33,14], which can be viewed as generalized matrix products. Sequences of tensor contractions arise in several domains, particularly, in *ab initio* computational models in quantum chemistry. Such contractions are often performed over several large multi-dimensional arrays, making it very likely that input data does not fit in main memory. At the same time, these applications are compute-intensive, and can benefit from GPUs.

We present an extensive framework for optimizing tensor contractions on a system requiring explicit data movement across three memory hierarchy levels. We address the following challenges: 1) determining the loop structure, which comprises of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. While we specifically target machines with GPUs and problems with out-of-core data structures, our framework can be applied on any system with more than 2 explicitly managed memory hierarchy levels.

2 Background

The tensor contraction expressions we consider in this paper arise, for example, in the context of the Coupled Cluster (CC) theory [3], a widely used method for solving the electronic Schrödinger equation. A tensor contraction expression is comprised of a collection of multi-dimensional summations of product of several input tensors or arrays. Consider the following contraction:

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

This contraction is referred to as a *four-index transform*. Here, $A(p, q, r, s)$ is a four-dimensional input tensor and $B(a, b, c, d)$ is the transformed (output) tensor. $C1$, $C2$, $C3$, and $C4$ are the *transformation* tensors. The indices $a, b, \dots$ denote the dimensionality of the tensors and the relationship between them across tensors. Minimizing the operation count results in the following sequence of binary tensor contractions:

$$\begin{aligned} T3(a, q, r, s) &= \sum_p C4(p, a) \times A(p, q, r, s); T2(a, b, r, s) = \sum_q C3(q, b) \times T3(a, q, r, s) \\ T1(a, b, c, s) &= \sum_r C2(r, c) \times T2(a, b, r, s); B(a, b, c, d) = \sum_s C1(s, d) \times T1(a, b, c, s) \end{aligned}$$

where $T1$, $T2$, and $T3$ are *intermediate* tensors. In each contraction, a set of indices are summed over, or *contracted*, and are referred to as the *summation (or contracted) indices*. These indices occur in both inputs of a binary contraction and are also referred to as *common* indices. This is a generalization of matrix-matrix multiplication with individual indices replaced by sets of indices.

The size of each dimension (or loop index) varies from 100 to a few thousand. As a result, the input, output, and the intermediate tensors often do not fit into the main memory of a single node. This, together with the high floating-point intensity of these expressions, lends themselves to execution on a parallel system, and considerable effort has been put on parallelization of these methods [15,2].

Effective execution of tensor contraction expressions on parallel systems with multi-level memory hierarchies necessitates several transformations. In particular, the allocation and freeing of memories in the the multi-level hierarchy together with the associated data transfer needs to be carefully orchestrated. While the execution is dominated by the floating point operations, data movement can quickly overwhelm the total execution time if not effectively managed. Tiling directly benefits these calculations, analogous to its applicability to matrix-matrix multiplication implementations. While much work on matrix multiplication focuses on square matrices, tensor contractions often involve highly rectangular matrices, exacerbated by the higher dimensionality involved. In particular, it is not uncommon for one of the arrays involved in a contraction to be small enough to fit at some higher-level of the memory hierarchy, such as in the example above.

Loop fusion is an integral part of effective data locality management in such multi-level memory hierarchies. Much of the early work on loop fusion either focused on identifying the feasibility of fusing two loops, fusing the identified loops [23,24], possibly with other transformations that enable loop fusion [38], or a general transformation framework that encompasses loop permutation, fusion, and other transformations under an abstract cost model [21,1,19].

In this paper, we focus on identifying effective fusions for tensor contractions with as concrete a cost model as possible. Specifically, we consider data movement costs under memory constraints. For systems with only two levels of memory hierarchy, this problem has been addressed before by Sahoo *et al.* [33].

We note that, unlike more general application classes, tensor contractions, which consist of fully permutable loops, can be fused without requiring enabling loop transformations.

3 Problem Statement and Notation

Given a sequence of binary contractions, such as the one in Section 2, efficient code tailored to the specific memory hierarchy needs to be generated. We focus on machines comprising more than two levels in the memory hierarchies. The key decisions to be made are 1) determining the loop structure, comprising of fused and tiled loops, 2) the placement of memory allocation and data movement statements for each of the memory hierarchy levels, and 3) the tile sizes. It has been shown previously that the the space of possible choices is exponentially large [33]. Besides considering additional levels in the memory hierarchy in determining fusion structures, we focus on the prescriptive approaches to choosing candidate loop fusions without requiring expensive optimization procedures.

In our presentation, $N_i, N_j, \dots$ and $T_i, T_j, \dots$ denote sizes of full dimensions and individual tiles along dimensions $i, j, \dots$, respectively. Lower case letters denote loop and array reference indices, or sets of indices. $A[a, b]$ denotes a tensor indexed by a and b . In addition to identifying the dimensionality of the tensor, the labels (here a and b) identify the relationship between the indices of the tensors participating in a tensor contraction. The memory levels are denoted by $\alpha, \beta, \dots$, with α being the slowest memory hierarchy level. The size of memory available at each level is denoted as $M_\alpha, M_\beta, \dots$, and the time to move one element of the data between adjacent levels is $\Lambda_{\alpha\beta}, \Lambda_{\beta\gamma}, \dots$. We assume that data movement is always performed between adjacent levels of the memory hierarchy. The cost to perform one double precision floating point operation is denoted by C_{flop} .

4 Single Contraction Optimization

In this section, we determine the loop structure to map a single tensor contraction to a multi-level memory hierarchy. We first summarize existing results for the case when data movement is considered at only a single level (e.g. from disks to main memory). A detailed description is available from an earlier publication [33]. In a tensor contraction, the indices can be grouped into those that are contracted and the remaining indices in the two input tensors. Given that the loops are fully permutable and we reason about the total data movement volume, ignoring issues such as stride of access, the indices can be grouped into three composite indices. In the ensuing discussion, we refer to the contracted indices as N_k , and the remaining indices in the two inputs as N_i and N_j .

One Array Fits in Memory: Consider the three tensors $X[x, y]$, $Y[y, z]$, and $Z[x, z]$ in a tensor contraction, where X , Y , and Z correspond to the three tensors participating in a binary tensor contraction, one of them being the output tensor. Without loss of generality, assume that X fits in memory. Let the size along the

Fig. 1. Loop structures for a single contraction on a two-level memory hierarchy. (a) Loop structure if $N_y \leq N_x$ (b) Loop structure if $N_x \leq N_y$ (c) Simplified loop structure for discussion (d) Example code for the contraction $C = A \times B$.

dimensions x , y , and z be N_x , N_y , and N_z , respectively. It can be shown that to minimize the data movement costs, the smallest array is retained in memory, with the other two arrays being streamed. This results in a memory cost of [33]:

$$N_x \times N_y + \min(N_x, N_y) + 1 \leq M_\beta \quad (4.1)$$

We approximate Equation 4.1 by replacing the left hand side with $N_x \times N_y$. For values of M_β encountered in practice, the value of the min term in the worst case ($N_x = N_y = \sqrt{M_\beta}$) is much smaller than the memory size, and can be ignored.

The loop structures for the cases where one of the arrays fits in the memory are shown in Figure 1, where X, Y, Z represent any of the three tensors. Note that none of the tensors are surrounded by loops that do not index it. This ensures that there is no redundant data movement for any of the arrays, achieving the minimum possible data movement cost, i.e. we read each input array and write the output array exactly once [33]. The position of a tensor in the loop structure identifies both data movement and memory management steps. Viewing the loop structure as a loop nest tree, a tensor label, denoted by an allocate statement for a tensor, identifies its position in the loop nest tree. The memory allocation and read statements (if the array is read) are inserted into the structure before the code corresponding to the nested statements are generated. After the processing of all nested statements, the deallocation and write statement (if it is an output array) are generated. While the optimal choice of loop structure that minimizes the memory usage is given by Figure 1(a) or Figure 1(b), the difference in practice is a small reduction in memory overhead, not the dominant memory overhead or data movement cost. We, therefore, consider the simplified structure shown in Figure 1(c) for further analysis. Figure 1(d) illustrates the insertion of memory allocation and data movement statements for the abstract structure shown in Figure 1(c) for the contraction $C = A \times B$.

No Arrays Fit in Memory: For a two-level memory hierarchy, say the $\alpha \leftrightarrow \beta$ levels, it has been shown that the optimal loop structure involves the data

transfer for the output array being higher in the loop structure than that for the other two arrays. Recalling that tiles sizes along X and Y dimensions are T_i and T_j , respectively, the data movement and memory costs are now given by:

$$Data_{\alpha\beta} = A_{\alpha\beta} \times \left(N_i \times N_j + N_i \times N_j \times N_k \times \left(\frac{1}{T_i} + \frac{1}{T_j} \right) \right) \quad (4.2)$$

$$Memory_\beta = T_i \times T_j + T_i + T_j < M_\beta \quad (4.3)$$

Minimizing the data movement under the memory constraint results in $T_i = T_j = T \approx \sqrt{M_\beta}$. Intuitively, optimized mapping of a tensor contraction to a two-level memory hierarchy requires maximizing the tile size for the output array with a small amount of data for the input arrays brought in at a time.

4.1 Multi-level Memory Hierarchies

We now focus on extending the existing work to the systems where data movement is required across more than two levels in the memory hierarchy. Specifically, consider a memory hierarchy consisting of three levels, namely α , β , and γ . The total data movement cost is the sum of costs of the two $\alpha \leftrightarrow \beta$ and $\beta \leftrightarrow \gamma$ data transfers. We shall assume in this discussion that levels cannot be bypassed, with data movement always occurring between adjacent levels. We also assume that the memories get faster and smaller as we move along the memory hierarchies, with α being the slowest level with infinite capacity.

First, if one of the tensors fits into γ , minimal data movement can be achieved by placing the array in γ , or reading directly into γ , and streaming the other two arrays through all levels of memory hierarchy using the loop structure shown in Figure 1. Each tensor label in such a schedule corresponds to data transfer across all memory hierarchy levels.

Consider the data movement order with all $\alpha\beta$ transfers grouped to nest the $\beta\gamma$ transfers, with $C_{\alpha\beta}$ being the outermost data placement and the data transfer for the A and B arrays at the same loop nesting. As explained earlier, we ignore the nesting between A and B given the limited memory usage improvement with no change in data movement cost. The costs for this data placement are given by:

$$Data_{\alpha\beta} = A_{\alpha\beta} \times \left(N_i \times N_j + N_i \times N_j \times N_k \left(\frac{1}{T_i} + \frac{1}{T_j} \right) \right) \quad (4.4)$$

$$Data_{\beta\gamma} = A_{\beta\gamma} \times N_i \times N_j \times N_k \left(\frac{1}{T'_i} + \frac{1}{T'_j} + \frac{q_k}{T_k} \right) \quad (4.5)$$

$$Memory_\beta = T_i \times T_j + T_i \times T_k + T_j \times T_k \quad (4.6)$$

$$Memory_\gamma = T'_i \times T'_j + T'_i \times T'_k + T'_j \times T'_k \quad (4.7)$$

Here, T'_i , T'_j , and T'_k are the tile sizes at the γ level, and q_k is 1 if $T_k = N_k$, 2 otherwise. This is because when $T_k = N_k$, the values of C could be written to the output without having to be loaded and accumulated. The tile sizes for the data movement placement can be determined by solving this non-linear constrained optimization problem. However, the solution requires additional information on the problem size, which is not always available to a compiler. Thus, here our focus

is on deriving prescriptive solutions that do not require precise information on the problem size. Where the problem size is needed, we reduced the determination to a quick runtime decision. In the process, we demonstrate the tile sizes for optimal data movement in a single-level memory hierarchy need not match those of a multi-level memory hierarchy.

Note that the data transfer cost at the $\beta\gamma$ level depends on the tile size T_k determined for the $\alpha\beta$ data transfer. The optimal data transfer for the single-level memory hierarchy, when no array fits in β , involves $T_i = T_j = \sqrt{M_\beta}$ and $T_k = 1$. Even under the assumption that $A_{\alpha\beta}$ is much higher than $A_{\beta\gamma}$, for large values of N_k , the data transfer cost is dominated by $Data_{\alpha\beta} + A_{\beta\gamma}N_i \times N_j \times N_k$. We thus consider another alternative, with $T_i = T_j = T_k = \sqrt{M_\beta/3}$. This corresponds to equal amounts of all three arrays being stored in memory at any time, a less than optimal choice for a two-level memory hierarchy. The total data movement cost (with $T'_i = T'_j = \sqrt{M_\gamma}$) is given by:

$$N_i \times N_j \times \left[A_{\alpha\beta} \times \left(1 + N_k \times \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) + A_{\beta\gamma}N_k \times \left(\frac{2}{\sqrt{M_\gamma}} + \frac{2\sqrt{3}}{\sqrt{M_\beta}} \right) \right] \quad (4.8)$$

Comparing this cost with Equations 4.4 and 4.5, we observe that equal tile sizes are better when:

$$\frac{2}{\sqrt{M_\beta}}A_{\alpha\beta} + 2A_{\beta\gamma} \geq \frac{2\sqrt{3}}{\sqrt{M_\beta}}(A_{\alpha\beta} + A_{\beta\gamma}) \implies M_\beta \geq \left(\frac{(\sqrt{3}-1)A_{\alpha\beta}}{A_{\beta\gamma}} + \sqrt{3} \right) \quad (4.9)$$

This condition is evaluated with $q_k = 2$ in the above determination. It can be quickly evaluated at install time to choose the best tile size. Intuitively, the data transfer between the $\alpha\beta$ memory levels is increased to significantly reduce the cost of transfer between the $\beta\gamma$ levels, keeping it relatively small. While globally optimizing the objective function yields the best solution, the prescriptive solution is independent of the problem size while achieving similar ends.

When one of the arrays fits in β , the optimal single-level solution involves fitting the array in β and streaming the other arrays. When it fits within $(M_\beta/3)$, both schemes result in similar cost functions and the tile sizes chosen for the single-level optimal solution are chosen. However, for the smallest array size between $(M_\beta/3)$ and M_β , we have a choice between the two tiling alternatives. With the streaming scheme, without loss of generality, we assume the smallest matrix is A , then by keeping the smallest matrix in β , the tile sizes are $T_i = N_i$, $T_k = N_k$, and $T_j \leq \frac{M_\beta - N_i N_k}{N_i + N_k}$. The data movement at the $\alpha\beta$ level will be

$$A_{\alpha\beta}(N_i N_k + N_j N_k + N_i N_j) \quad (4.10)$$

This tiling decision at the $\alpha\beta$ level results in tensors of sizes T_j , $T_j N_k$, and $T_j N_i$ residing in β , which need to be scheduled for data movement across $\beta\gamma$. Since $N_i N_k$ is larger than $(M_\beta/3)$, we assume it does not fit in γ . However, because of possibly smaller T_j , $T_j N_k$, or $T_j N_i$ might fit γ . In this case, streaming is done in a way similar to β level. Suppose the smallest tile at β level is from B , which is of the size $T_j T_k$, then the total data movement at $\beta\gamma$ is

$$A_{\beta\gamma} \left(N_j N_k + N_i N_j + \frac{N_i N_k N_j}{T_j} \right) = A_{\beta\gamma} N_j (N_k + N_i) \left(1 + \frac{N_i N_k}{M_\beta - N_i N_k} \right) \quad (4.11)$$

When none of the data tiles fit in γ , a tiling solution analogous to the single-level tiling presented in Section 4 is employed. This results in the total data movement cost at $\beta\gamma$ being:

$$A_{\beta\gamma} \left(\frac{N_j N_k}{\sqrt{M_\gamma}} + \frac{N_i N_j}{\sqrt{M_\gamma}} + \frac{N_i N_k N_j}{T_j} \right) = A_{\beta\gamma} N_j (N_i + N_k) \left(\frac{2}{\sqrt{M_\gamma}} + \frac{N_i N_k}{M_\beta - N_i N_k} \right) \quad (4.12)$$

The total data movement cost is computed from the $\alpha\beta$ cost (given by Equation 4.12) together with the $\beta\gamma$ cost. The latter is determined using either of the Equations 4.11 and 4.12, based on the problem size. By comparing this overhead with the data movement using equal tile sizes, whose cost is given by the Equation 4.8, the tiling decision could be made at runtime. The decision could also be made at installation time for different matrix shapes, which impact the $(N_i + N_k)$ term.

5 Fusion for Tensor Contraction Sequences

In this section, we describe our approach to determining fused loop structures for multi-level memory hierarchies. We begin with an analytical approach to identify profitable loop fusions for tensor contractions. Intuitively, we observe that loop fusion can result in substantial performance gains only if the overall execution time is bound by data movement, and not computation. Consider a series of loops, which correspond to the following series of tensor contraction expressions.

$$\begin{aligned} I_1(d, c_2, \dots, c_n) &= I_0(d, c_1, \dots, c_n) \times B_0(d, c_1, \dots, c_n) \\ I_2(d, c_3, \dots, c_n) &= I_1(d, c_2, \dots, c_n) \times B_1(d, c_2, \dots, c_n) \\ &\dots \\ I_n(d) &= I_{n-1}(d, c_n) \times B_{n-1}(d, c_n) \end{aligned}$$

The tensors $B_0, \dots, B_{n-1}$ and I_0 are the *input* tensors and I_n is the *output* tensor. $I_1, \dots, I_{n-1}$ are the *intermediate tensors*. Data movement costs for such intermediate tensors can potentially be eliminated through fusion. c_j corresponds to the indices contracted in the production of I_j . The various d indices are never contracted out and contribute to the indices in I_n . $I_i(d)$ represents the set of indices I_i contributes to the final output. $I_i(c_j)$ denotes the indices in I_i that are a subset of the contracted indices in contraction j . I_i denotes both the tensor produced by contraction i and set of all indices that constitute that tensor. The reference shall be clear from the context. $|I_j(d)|$, and similar usage, denotes the total size of the indices in tensor I_j that contribute to the final output. The relationship between the indices is defined as:

$$\begin{aligned} 1 \leq i \leq n : I_i(d) &= I_{i-1}(d) \cup B_{i-1}(d) \\ 1 \leq i \leq n, 1 \leq j \leq i : I_i(c_j) &= \emptyset \\ 1 \leq i \leq n, i+1 \leq j \leq n : I_i(c_j) &= I_{i-1}(c_j) \cup B_{i-1}(c_j) \\ 0 \leq i \leq n-1 : I_i(c_{i+1}) &\equiv B_i(c_{i+1}) \end{aligned}$$

The fusion of the loops corresponding to the above tensor expressions is beneficial only if the data movement cost for the intermediate tensors (I_1 through I_{n-1}) dominates, or is at least comparable to, the data movement cost for the input and the output tensors and the computation cost.

Consider the fusion of the first two contractions. The computation cost of the first contraction is $C_{flop} \left(|I_0(d)| \times \prod_{j=1}^n |I_0(c_j)| \times |B_0(d)| \times \prod_{j=2}^n |B_0(c_j)| \right)$. The minimum data movement cost for I_1 that would be eliminated through loop fusion for the $\alpha\beta$ levels is given by: $2 \times \Lambda_{\alpha\beta} \times \left(|I_1(d)| \times \prod_{j=2}^n |I_1(c_j)| \right)$, the cost of reading and writing the tensor once for each of the contractions. Under the requirement that this is greater than the fraction, $0 < frac \leq 1$, of the computation cost of the two relevant contractions, we have:

$$2 \times \Lambda_{\alpha\beta} \geq frac \times C_{flop} \times |I_0(c_1)|$$

leading to

$$|I_0(c_1)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \quad (5.1)$$

For the second contraction, the computation cost is given by $C_{flop} \times \left(|I_1(d)| \times \prod_{j=2}^n |I_1(c_j)| \times |B_1(d)| \times \prod_{j=3}^n |B_1(c_j)| \right)$. If we want the execution to be dominated by the movement of I_1 , the condition now becomes

$$2\Lambda_{\alpha\beta} \geq frac \times C_{flop} \prod_{j=3}^n |B_1(c_j)| |B_1(d)| \implies \prod_{j=3}^n |B_1(c_j)| |B_1(d)| \leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \quad (5.2)$$

Therefore, for a contraction to be fused with both its previous and the next contraction, the input B_i has to satisfy both of the following requirements:

$$\begin{aligned} |I_i(c_{i+1})| &\leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \\ \prod_{j=i+2}^n |B_i(c_j)| \times |B_i(d)| &\leq \frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \end{aligned} \quad (5.3)$$

Combining the above two expressions, together with the fact that $I_i(c_{i+1}) = B_i(c_{i+1})$, the size of B_i should be less than $\left(\frac{2 \times \Lambda_{\alpha\beta}}{frac \times C_{flop}} \right)^2$. In current systems, this number is typically much smaller than the memory size. We therefore assume that the B tensors in all contractions in the sequence considered, except the first and last contraction, fit in memory.

The above evaluation does not preclude the B arrays in the first or last contraction in a fusible list, B_0 and B_{n-1} in the above example, from being too large to fit in memory. In these cases, fusion eliminates the data movement cost to read and write the intermediate tensor. On the other hand, fusion requires this contraction to share the available memory with other contractions, potentially increasing the cost incurred due to duplicated data movement.

Consider B_0 being large, with qM ($0 \leq q \leq 1$, M_β is the memory limit of level β) being the amount of memory required for fused execution of contractions

```

S1 = Ii ∩ Ii+1 ∩ Ii+2
S2 = Ii ∩ Ii+1, S3 = Ii+2
S4 = Ii
for sx ∈ S1 do
  {Allocate Ii+1[sx]};
  for sy ∈ S2 − S1 do
    {Allocate Ii[sy]};
    for sz ∈ S4 − S2 do
      {Produce Ii[sz]};
    end for
    {Update Ii+1[sy]};
  end for
  for sw ∈ S3 − S1 do
    {Allocate Ii+2[sw]};
    {Produce Ii+2[sw]};
  end for
end for

```

Algorithm 1. Sample loop structure with allocation for I_i , I_{i+1} , and I_{i+2}

producing tensors I_2 through I_n in the running example. Comparing the two alternatives discussed above, fusing the first contraction is beneficial when the condition below is true:

$$\begin{aligned}
\frac{2|I_1||I_0(c_1)|}{\sqrt{M_\beta}} + 2|I_1| &\geq \frac{2|I_1||I_0(c_1)|}{\sqrt{(1-q)M_\beta}} \implies \frac{|I_0(c_1)|}{\sqrt{M_\beta}} + 1 \geq \frac{|I_0(c_1)|}{\sqrt{M_\beta}\sqrt{1-q}} \\
&\implies \frac{\sqrt{1-q}}{1-\sqrt{1-q}} \geq \frac{|I_0(c_1)|}{\sqrt{M_\beta}} \quad (5.4)
\end{aligned}$$

As shown in equation 5.1, $|I_0(c_1)|$ is small, usually much smaller than the memory available. The expression on the left hand side of last inequality above is greater than 1 for values of $q \leq 0.7$. Thus fusion is beneficial despite duplicated data movement for one of the arrays even when up to 70% of the memory is consumed by the remaining contractions. Note that this condition can be quickly verified at runtime once the problem size is known.

5.1 Two-Level Memory Hierarchy

Based on the discussion above, we assume that for a given sequence of tensor contractions to be fused, the non-intermediate arrays in all contractions, except for the first and last contraction, must fit in the memory. For simplicity, we further assume that all such arrays together fit in the available memory. Based on this assumption, we will first present a solution to the problem of identifying a loop structure, with placement of data movement statements for all the intermediate arrays and one of the input arrays in the first contraction and the output array in the last contraction, assuming the remaining arrays fit in memory. This will later be extended to support the scenario in which the first or the last contraction require redundant data movement.

We simplify our presentation by assuming that all dimensions are similar in size. Consider a contraction list, referred to as a *fusable list*, in which all contractions are *fusable* according to the analysis presented in the previous subsection. We now determine the actual fusion chain and locations of the data allocation and movement statements. Let the fusable list be $I_0, \dots, I_n$. For any three tensors I_i, I_{i+1} , and I_{i+2} , a sample loop structure is as shown in Algorithm 1, where the intersection operation denotes the intersection of indices of two tensors. The key observation here is that the loop nest producing I_{i+1} must be enclosed by a memory allocation for I_i , since the loop nest consumes I_i . However, the allocation statements for tensors I_i and I_{i+2} need not enclose each other, since they do not have an immediate producer-consumer relationship. The data movement statements can be in arbitrary order as long as this condition is satisfied.

In this sample loop structure, the total memory cost is given by $mem(I_{i+1}) + \max(mem(I_i), mem(I_{i+2}))$. Since the unfused loop nest for each contraction, with one of the arrays fitting in memory, does not incur any redundant data movement, we consider only loop nests with no redundant data movement. The total memory cost can be determined through the recurrence relation shown below:

$$\begin{aligned} f(i, j) &= 0 \quad , \text{ if } j < i \\ &= \min_{k=i}^j \frac{|I_k|}{|Common_{i,j}|} + \max(f(i, k-1), f(k+1, j)) \quad , \text{ otherwise} \\ Common_{i,j} &= \cap_{k=i}^j I_k \end{aligned}$$

The recurrence is evaluated as $f(1, n)$ to compute the values of all $f(i, j)$ through dynamic programming. $Common_{i,j}$ denotes all the loops that are common to tensor contractions i through j . The procedure to determine the loop structure first involves computing the memory required by arbitrary sub-chains of the “fusable list” memoized in a matrix. The function computes the data movement cost, in terms of volume, of executing a given chain of tensor contractions, ignoring the data movement for the first input and the last output. If the entire chain can be fused with the memory available, the data movement cost is evaluated to be zero. If not, various choices of splitting the chain into sub-chains are considered, and the choice that minimizes the data movement cost is determined recursively. For each split the array at the split incurs the data movement cost equal to twice its size, to write it as output from the first sub-chain and read it back in the next sub-chain. Note that the algorithms shown only compute the memory and data movement costs. Determining the actual loop structure including the loop nesting to obtain the memory cost and splits to achieve the data movement cost can be determined by keeping track of the choices made while minimizing the cost function. The procedure is similar to that employed in the dynamic programming solution to matrix-chain multiplication [9].

When either the first or the last contraction involves large tensors necessitating duplicated data movement, we determine the optimal fusion structures for the fusable list excluding the contractions requiring such duplicated data movement. Let us consider the case with I_0, B_0 , and I_1 being too large to fit in

memory. As discussed in Section 5.1, the loops in the second contraction consuming the output of the first contraction should be enclosed by the memory allocation statement for the output tensor of the first contraction. In the running example, the allocation statement for I_1 must enclose the production of I_2 . Positions in the loop structure determined for contractions 2 through n that satisfy this requirement are determined. If the memory consumed by all live tensors at any of these positions is small enough to enable profitable fusion, as expressed by the condition 5.4, the first contraction is fused. This procedure is repeated for the last contraction in the list when it incurs duplicated data movement.

5.2 Multi-level Memory Hierarchies

In the previous section, we presented the steps to determine the fusion structure that minimizes the data movement cost given a memory constraint. Here we show how the procedure naturally extends to handle multi-level memory hierarchies. We have considered fusion as a transformation that combines the loop corresponding to the producer and consumer such that the data movement cost for the associated intermediate tensor is eliminated. Effective fusion of a sequence of contractions for data movement between the $\beta\gamma$ levels implies that all contractions in the chain can be executed without intervening $\beta\gamma$ data transfer, with sufficient space in γ to fit all necessary intermediates. A sub-chain can be fused at the γ level if the memory required by it, as computed by this expression, is less than M_γ . Given the memory cost for each sub-chain at γ , the memory cost at the β level is calculated using the following modified recurrence expression:

$$\begin{aligned}
 f(i, j) &= 0 \quad , \quad \text{if } j < i \\
 &= \frac{|I_i| + |I_j|}{|I_i \cap I_j|} \quad \text{if } \text{memory}_\gamma(i, j) \leq M_\gamma \\
 &= \min_{k=i}^j \frac{|I_k|}{|\text{Common}_{i,j}|} + \max(f(i, k-1), f(k+1, j)) \quad , \quad \text{otherwise} \\
 \text{Common}_{i,j} &= \cap_{k=i}^j I_k
 \end{aligned}$$

This expression takes into account the fact that when a sub-chain is fused at the γ level, only the first and last tensor in that sub-chain consume memory at the β level. The total data movement cost for fusion in the presence of a multi-level memory hierarchy is computed using a modified version of procedure employed for a two-level memory hierarchy. If the memory required at the β level to execute a sub-chain is less than M_β , the corresponding data movement cost is determined. The data movement cost for a sub-chain fusable at the γ level is given as the cost to transfer the first and last tensor in the chain between the $\beta\gamma$ levels. We assume that despite fusion of some of the loops, the dimensions of the remaining loops are large enough for tiling all dimensions at the γ level. When a sub-chain involves a single contraction, due to the need for redundant data movement, its total data movement cost is computed as discussed in Section 4.1.

Expression A (Small Input):

$$C[h1,h2,h3,h4,h5,h6] = A[h1,h2,h3,h7] \times B[h4,h5,h6,h7]$$

$$I[h1,h2,h3,h4,h5,h8] = B2[h8,h6] \times C[h1,h2,h3,h4,h5,h6]$$

Expression B (Large Common Index):

$$C[h6,h5,h4,h3,h2,h1] = A[h7,h5,h4,h3,h2,h1] \times B[h7,h6]$$

$$I[h8,h2,h1] = B2[h8,h6,h5,h4,h3] \times C[h6,h5,h4,h3,h2,h1]$$

Expression C (Large Input):

$$C[h8, h7, h6,h5,h4,h3,h2,h1] = A[h9, h4,h3,h2,h1] \times B[h9, h8, h7,h6,h5]$$

$$I[h10,h7,h6,h5,h4,h3,h2,h1] = B2[h10,h8] \times C[h8, h7, h6,h5,h4,h3,h2,h1]$$

Expression D (Long Chain):

$$M1[h1,h2,h3,h4,h5,h7] = M0[h1,h2,h3,h4,h5,h6] \times B1[h6,h7]$$

$$M2[h1,h2,h3,h4,h5,h8] = M1[h1,h2,h3,h4,h5,h7] \times B2[h5,h8]$$

$$M3[h1,h2,h3,h9,h7,h8] = M2[h1,h2,h3,h4,h5,h8] \times B3[h4,h9]$$

$$M4[h1,h2,h10,h9,h7,h8] = M3[h1,h2,h3,h9,h7,h8] \times B4[h3,h10]$$

$$M5[h1,h11,h10,h9,h7,h8] = M4[h1,h2,h10,h9,h7,h8] \times B5[h2,h11]$$

Expression E (Fewer Dimensions and Large Dimension Size):

$$C[h1,h2,h3,h4] = B[h3,h5] \times A[h1,h2,h4,h5]$$

$$I[h1,h2,h3,h6] = B2[h6,h4] \times C[h1,h2,h3,h4]$$

Fig. 2. Tensor contraction expressions used as benchmarks

6 Experimental Evaluation

We evaluate our methods by executing tensor contractions on a system with an explicitly managed, multi-level, memory hierarchy. Particularly, the system consisted of the following three levels in the memory hierarchy: disk, global memory, and the *local* or *device* memory. The global memory corresponds to the use of *global arrays* [27] on the cluster. The device memory is the memory on the GPU, which had a similar size as the local memory on each node. Since all processing is done on the GPU, processor cache is not an important component of the memory hierarchy. Within the GPU, the programmable cache or the *shared memory* reflects another level in the hierarchy, but did not turn out to be important for our target class of applications. In particular, the shared memory in the GPUs we used was too small to enable fusion. The impact of tiling on kernels such as matrix-matrix multiplication has been extensively evaluated in a variety of contexts [25,31,16,8]. While the tile sizes are derived as discussed earlier, we focus on demonstrating the impact of our approach to fusion.

The specific configuration used was a cluster environment where each computing node has two quad-core Intel Xeon X5560 CPUs, running at 2.80 GHz. Every pair of nodes shares a Tesla 1070 GPU box, which means every node has two GPUs available as accelerators. For simplicity, in our experiments, we launch two processes on each node, so that every process can use one GPU for acceleration. Our analytical models assume that there is a 32 GB global memory and 1 GB local memory for the application's data structures. In all the experiments, the input data and the final output are stored on the disk. Input files are

Fig. 3. Execution time with (left) and without (right) fusion at global memory level

Fig. 4. Disk I/O time with (left) and without (right) fusion at disk level

first loaded into the global memory, then moved to the local memory. Storage of output file follows the reverse order. Communication between disk and global memory is *collective*, whereas, the data movement between the global memory and the local memory is *one-sided* (each processor transfers data independently).

Our evaluation was based on five candidate sets of contractions, chosen to reflect the nature of the contractions in quantum chemistry. These benchmark expressions are listed in Figure 2. For each expression, we evaluate the benefit of fusion at both the global memory level and the disk level. Fusion at the global memory level implies that the intermediate data resides in the device memory, whereas fusion at the disk level implies that the intermediate data is kept in the global memory. As a baseline, we also created a non-fused version which refers to the case when all intermediate results are copied to the disk and need to be loaded for the next contraction. Our approach predicts that fusion is beneficial at the disk level for all five expressions, while fusion at the global memory level is beneficial for expressions *A*, *B*, *C*, and *D*.

6.1 Effectiveness on Different Expressions

We compare the fused and non-fused versions in terms of computation cost, the transfer time between global and local memory levels, and disk I/O cost. In all the charts, **LM-GM** implies data transfer time between the local memory and the global memory. In Figures 3, 5, and 7), the left bar in each *bar cluster* denotes the time with fusion, and the right bar denotes the time without fusion. **Computation** time reported here includes time spent in floating point operations and data movement between local memory and the GPU device memory. Therefore, in some cases, the computation time reported is shorter in the version with fusion. The disk I/O times, which could be much higher than the other components, are shown in Figures 4, 6, and 8.

Expression A – Fusion with small input matrices: The Expression A involves two contractions, where all input matrices fit in local memory. Data transfers for the matrices *C* and *I* are the dominating factors in this computation,

and fusion eliminates the overhead of moving C . The first cluster of bars in Figure 3 and Figure 4 show the benefit achieved by fusion at the disk and global memory levels, respectively. The experiments were done with the dimension size 40, on four computing nodes. We can see that the fusion at the GM level reduces the sum of computation and the local memory transfer time by about 40%. On the other hand, fusion at disk to global memory level is playing an even more important role, since the execution is dominated by this part. From Figure 4, we can see that fusion results in speedup of about 2.4.

Expression B – Fusion with large common index in the last contraction: In this expression, the input $B2$ in the second contraction is large, so tiling is done on the dimensions $h5$, $h4$, $h3$, and $h2$. Though it introduced replicated accumulation for I , the overhead is still much smaller than the overhead of communication for C and $B2$. Fusion turns out to be beneficial as it removes communication for C . The performance with and without fusion at the two levels is shown in the second clusters from the left in Figures 3 and 4. We have experimented with the size of each dimension being 40. For the sum of computation and local memory transfer times, fusion resulted in a 45% reduction. At the disk level, the speedup by fusion is about 3.7, because the large amount of time spent in writing C is avoided.

Expression C – Fusion for large inputs in the first contraction: As stated in Section 5.1, when the two input matrices in the first contraction are large enough to exceed the memory limit at a certain level of memory, fusion is still beneficial because the intermediate result is still the dominant part. Expression C represents this case, under the assumption that the matrix C is very large and does not fit in the local memory. Therefore, tiling has to be performed on the *non-common index* of the input matrices A and B .

We experimented with the dimension size being 16. From the third bar cluster in Figure 3, we can see that fusion saved 75% of the sum of computation and the local memory transfer times. Again, the benefit of fusion at the disk level is more significant. By fusion, the total disk I/O has a speedup of about 2.5 by removing matrix C , the dominating factor in the non-fused version.

Expression D – Fusion for a list of five contractions: The expression D was particularly chosen to test the fusion algorithm we had presented for a *fusion chain*. Every contraction here has one small input ($B1$, $B2$, $B3$, $B4$, and $B5$), which could reside in the local memory. Fusion of all five contractions requires memory size N^5 , where N is the size of one dimension. Therefore, in the case where the size of the local memory is 1 GB, we cannot fuse all the five contractions at global memory level when tile size is 40 or more. According to our fusion algorithm, in the fused version at the global memory level, if the tile size is smaller than 40, we fuse all the five contractions. However, when tile size is larger, we break the fusion chain and fuse the two shorter chains separately.

The performance for this expression with the dimension size 40 is shown as the fourth bar cluster in Figures 3 and 4. Moreover, as we have analyzed

Fig. 5. Execution time with (left) and without (right) fusion at global memory level for expression D - different dimension sizes

Fig. 6. Disk I/O time with and without fusion at disk level for expression D - different dimension sizes

above, this expression exhibits interesting behavior as problem size is changed. Therefore, we analyzed the performance with varying dimension sizes, and the results are shown in Figures 5 and 6. For dimension size smaller than 40, the speedup of fusion is about 2~2.5, and drops to 1.4~1.7 when $N \geq 40$. At the disk level, fusion of five contractions is always feasible with problem sizes we have experimented with, resulting in a speedup of 4~5 due to fusion.

Expression E – A case for which fusion does not have significant benefits: Our fusion condition predicts that fusion does not have significant benefit under certain conditions. The right most bar cluster in Figure 3 shows the performance with and without fusion at global memory level for the Expression E. We used a dimension size of 256. The execution time here is dominated by the computation time. Speedup by fusion is only about 1.02~1.09, which is consistent with our prediction. This shows that our models are capable of correctly predicting when fusion is not be beneficial, or result in small improvements.

6.2 Scalability Study

We studied the benefit of our optimizations as the number of nodes in the cluster increases. We only present detailed results from the Expression A – the results from other cases are similar. Figures 7 and 8 show the performance on a relatively large problem size ($N = 48$), with different numbers of nodes. For the sum of computation and local memory transfer times, the speedup is about 1.8~2.3. At disk level, the speedup due to fusion ranges between 2.3 and 2.5. It can be seen that the computation time decreases with increasing number of nodes, i.e., the computation is being parallelized effectively. However, the communication time becomes relatively large, as data may be retrieved from different parts of the global memory. Thus, fusion becomes increasingly important at larger node counts. At the disk level, the I/O time is relatively independent of the number of nodes, resulting in relatively stable performance improvement from fusion.

Fig. 7. Execution time with (left) and without (right) fusion at global memory for expression A on different numbers of nodes

Fig. 8. Disk I/O time with and without fusion for expression A on different numbers of nodes

7 Related Work

The previous work on optimizing tensor contractions focused on efficient exploration of the search space of memory hierarchy transformations [33,14]. Sahoo *et al.* pruned the search space while combining permutation, loop fusion, tiling, and placement of disk I/O operations [33]. Gao *et al.* extended this work further to minimize disk I/O costs [14]. This work only considered data movement across one level of memory hierarchy. In comparison, our work is the first in offering solutions with data movement across more than two levels. We have also presented more efficient algorithms for determining candidates for loop fusion.

Outside the specific context of tensor contractions, there is a very large volume of work on optimizing for memory hierarchies [23,24]. Most of this work has focused on improving the utilization of a processor cache, which is not explicitly programmer controlled. One key difference between the work in this area and our work is in the metrics used for transformations. Cache transformations are based on metrics like reuse distance [12] or stack distance [7]. In comparison, we target programmer controlled memory hierarchy levels, and applications where we primarily see capacity misses. Also, because we are focusing on a limited class of applications, it is possible to develop a more accurate cost model. Darté has studied the general complexity of loop fusion [10] and has presented an approach for optimized memory allocation [11].

Significant work has been done on optimizing out-of-core programs. Particularly, Brown *et al.* have used compiler analysis to optimize prefetching of data in out-of-core programs [6]. Kandemir *et al.* have considered several compiler optimizations for out-of-core parallel programs [18] and other I/O intensive applications [17], building on earlier work at Syracuse/Northwestern and Rice [5,36]. The key difference in our work is the consideration of data movement across more than two levels of programmer controlled memory hierarchy.

Prior to interest in accelerating computations using GPUs, several researchers have studied deeper memory hierarchies. Mitchell *et al.* considered *multi-level*

caches, and further included considerations for instruction-level parallelism and TLB in their work [25]. For multi-level caches, they observed that single-level tiling is sufficient. We have considered a different architecture, and our conclusions are different. Rivera and Tseng examined loop transformations for multi-level caches, and finding that all performance gains can be achieved by simply focusing on L1 cache [32]. Clearly, as we have considered architectures with a different type of multi-level memory hierarchy, our conclusions are different. More recent work has considered multi-level tiling, but specific to stencil computations [30]. We perform tiling for a different class of applications. Qaseem *et al.* have used complex modeling and a direct search to find transformation parameters [28]. Ren *et al.* presented an auto-tuning framework for software-managed memory hierarchies [29]. We, in comparison, employ an analytical approach.

With recent interest in GPUs, there has been some work on optimizing data movements from main memory to device memory [35,34,22]. However, we are not aware of any extensive code restructuring framework to optimize these costs. Within the context of GPUs, another research direction involves optimization of shared memory use in GPUs, which are also a form of application-controlled cache. In this area, Baskaran *et al.* have provided an approach for automatically arranging shared memory on NVIDIA GPU by using the polyhedral model for affine loops [4]. Moazeni *et al.* have adapted approaches for register allocation, particularly those based on graph coloring, to manage shared memory on GPU [26]. A very similar problem is optimization for *scratch-pad* memory in embedded systems. Diouf *et al.* used integer linear programming to solve the problem [13]. Li *et al.* proposed interval coloring approach for arranging scratch-pad memory, utilizing the observation that live ranges of two arrays should not interfere or one should not contain the other [20]. Udayakumaran *et al.* address the scratch-pad memory allocation in a dynamic way, using a greedy heuristic as the cost model [37]. None of these efforts have considered multi-level data movement. Most of these efforts are also not suitable for applications like tensor contractions, where only a fraction of a single array can fit in the memory.

8 Conclusions

Emerging high-end architectures are bringing new challenges for compilers and code generation systems. Particularly, deeper, explicitly controlled memory hierarchies are becoming common and need to be optimized for. This paper has considered tiling, loop fusion, and placement of data movement operations for tensor contractions, which is an important class of scientific applications, for systems with more than two levels in the memory hierarchy. We have developed practical techniques focusing on the dominant cost factors. Experimental evaluation has shown that loop fusion for multiple levels improves performance and our methods can correctly predict cases where loop fusion is not advantageous. In addition to being effective, we believe these low complexity approaches serve to reduce the search space and provide effective starting points for empirical search in the “neighborhood” of the candidate loop structures chosen.

References

1. Ahmed, N., Mateev, N., Pingali, K.: Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming* 29(5), 493–544 (2001)
2. Aprà, E., Rendell, A.P., Harrison, R.J., Tipparaju, V., deJong, W.A., Xantheas, S.S.: Liquid water: obtaining the right answer for the right reasons. In: SC (2009)
3. Bartlett, R.J., Musiał, M.: Coupled-cluster Theory in Quantum Chemistry. *Rev. Mod. Phys.* 79(1), 291–352 (2007)
4. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: PPOPP, pp. 1–10 (2008)
5. Bordawekar, R., Choudhary, A., Kennedy, K., Koelbel, C., Paleczny, M.: A model and compilation strategy for out-of-core data parallel programs. In: PPOPP, pp. 1–10 (July 1995)
6. Brown, A.D., Mowry, T.C., Krieger, O.: Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.* 19(2), 111–170 (2001)
7. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: ICS, pp. 150–159 (2003)
8. Coleman, S., McKinley, K.S.: Tile size selection using cache organization and data layout. In: PLDI, pp. 279–290 (1995)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to algorithms*. MIT Press, Cambridge (2001)
10. Darte, A.: On the complexity of loop fusion. *Parallel Computing* 26(9), 1175–1193 (2000)
11. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. *IEEE Trans. Computers* 54(10), 1242–1257 (2005)
12. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: PLDI, pp. 245–257. ACM, New York (2003)
13. Diouf, B., Ozturk, O., Cohen, A.: Optimizing local memory allocation and assignment through a decoupled approach. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 408–415. Springer, Heidelberg (2010)
14. Gao, X., Krishnamoorthy, S., Sahoo, S.K., Lam, C.-C., Baumgartner, G., Ramanujam, J., Sadayappan, P.: Efficient search-space pruning for integrated fusion and tiling transformations. *Concurrency and Computation: Practice and Experience* 19(18), 2425–2443 (2007)
15. Hirata, S.: Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A* 107(46), 9887–9897 (2003)
16. Hsu, C.-h., Kremer, U.: A quantitative analysis of tile size selection algorithms. *J. Supercomput.* 27(3), 279–294 (2004)
17. Kandemir, M., Choudhary, A., Choudhary, A.: Compiler optimizations for i/o intensive computations. In: *Proceedings of International Conference on Parallel Processing* (September 1999)
18. Kandemir, M., Choudhary, A., Ramanujam, J., Bordawekar, R.: Compilation techniques for out-of-core parallel computations. *Parallel Computing* 24(3-4), 597–628 (1998)
19. Kelly, W., Pugh, W.: Finding legal reordering transformations using mappings. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) LCPC 1994. LNCS, vol. 892. Springer, Heidelberg (1995)

20. Li, L., Nguyen, Q.H., Xue, J.: Scratchpad allocation for data aggregates in super-perfect graphs. In: LCTES 2007: Proceedings of Conference on Languages, Compilers, and Tools for Embedded Systems, pp. 207–216 (2007)
21. Lim, A.W., Cheong, G.I., Lam, M.S.: An affine partitioning algorithm to maximize parallelism and minimize communication. In: International Conference on Supercomputing, pp. 228–237 (1999)
22. Ma, W., Agrawal, G.: A Translation System for Enabling Data Mining Applications on GPUs. In: Proceedings of International Conference on Supercomputing (ICS) (June 2009)
23. McKinley, K.S., Carr, S., Tseng, C.-W.: Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18(4), 424–453 (1996)
24. McKinley, K.S., Temam, O.: Quantifying loop nest locality using spec'95 and the perfect benchmarks. *ACM Trans. Comput. Syst.* 17(4), 288–336 (1999)
25. Mitchell, N., Högstedt, K., Carter, L., Ferrante, J.: Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming* 26(6), 641–670 (1998)
26. Moazeni, M., Bui, A., Sarrafzadeh, M.: A Memory Optimization Technique for Software-Managed Scratchpad Memory in GPUs (July 2009), <http://www.sasp-conference.org/index.html>
27. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing* 10(2), 169–189 (1996)
28. Qasem, A., Kennedy, K., Mellor-Crummey, J.M.: Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing* 36(2), 183–196 (2006)
29. Ren, M., Park, J.Y., Houston, M., Aiken, A., Dally, W.J.: A tuning framework for software-managed memory hierarchies. In: PACT, pp. 280–291 (2008)
30. Renganarayana, L., Harthikote-matha, M., Dewri, R., Rajopadhye, S.: Towards optimal multi-level tiling for stencil computations. In: IPDPS (2007)
31. Renganarayana, L., Rajopadhye, S.: Positivity, posynomials and tile size selection. In: SC, pp. 1–12 (2008)
32. Rivera, G., wen Tseng, C.: Locality Optimizations for Multi-level Caches. In: Proceedings of the SC 1999 (November 1999)
33. Sahoo, S.K., Krishnamoorthy, S., Panuganti, R., Sadayappan, P.: Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In: SC, p. 13. IEEE Computer Society, Los Alamitos (2005)
34. Sundaram, N., Raghunathan, A., Chakradhar, S.: A framework for efficient and scalable execution of domain-specific templates on GPUs. In: IPDPS (2009)
35. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program gpus for general-purpose uses. In: ASPLOS, pp. 325–335 (2006)
36. Thakur, R., Bordawekar, R., Choudhary, A.: Compilation of out-of-core data parallel programs for distributed memory machines. In: Second Annual Workshop on Input/Output in Parallel Computer Systems (IPPS), pp. 54–72 (April 1994)
37. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5(2), 472–511 (2006)
38. Yi, Q., Kennedy, K., Adve, V.: Transforming complex loop nests for locality. *J. Supercomput.* 27(3), 219–264 (2004)