Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions☆

Qingda Lu[a,1], Xiaoyang Gao[a,2], Sriram Krishnamoorthy[a,3], Gerald Baumgartner[b,*], J. Ramanujam[c], P. Sadayappan[a]

[a]*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA*
[b]*Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*
[c]*Dept. of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA*

Abstract

Empirical optimizers like ATLAS have been very effective in optimizing computational kernels in libraries. The best choice of parameters such as tile size and degree of loop unrolling is determined in ATLAS by executing different versions of the computation. In contrast, optimizing compilers use a model-driven approach to program transformation. While the model-driven approach of optimizing compilers is generally orders of magnitude faster than ATLAS-like library generators, its effectiveness can be limited by the accuracy of the performance models used. In this paper, we describe an approach where a class of computations is modeled in terms of constituent operations that are empirically measured, thereby allowing modeling of the overall execution time. The performance model with empirically determined cost components is used to select library calls and choose data layout transformations in the context of the Tensor Contraction Engine, a compiler for a high-level domain-specific language for expressing computational models in quantum chemistry. The effectiveness of the approach is demonstrated through experimental measurements on representative computations from quantum chemistry.

Keywords: data layout optimization, library call selection, compiler optimization, tensor contractions

1. Introduction

Optimizing compilers use high-level program transformations to generate efficient code. The computation is modeled in some form and its cost is derived in terms of metrics such as reuse distance. Program transformations are then applied in order to reduce the cost. The large number of parameters and the variety of programs to be handled limits optimizing compilers to employ model-driven optimization with relatively simple cost models. As a result, there has been much recent interest in developing generalized tuning systems that can similarly tune and optimize codes input by users or library developers [12, 60, 57]. Approaches to empirically optimize a computation, such as ATLAS [59] (for linear algebra) and FFTW [19] generate solutions for different structures of the optimized code and determine the parameters that optimize the execution time by running different versions of the code for a given target architecture and choosing the optimal one. But empirical optimization of large complex applications can be prohibitively expensive.

In this paper, we decompose a class of computations into its constituent operations and model the execution time of the computation in terms of an empirical characterization of its constituent operations. The empirical measurements allow modeling of the overall execution time of the computation, while decomposition enables off-line determination of the cost model and efficient global optimization across multiple constituent operations. This approach combines

☆This work was supported in part by the National Science Foundation under grants CHE-0121676, CHE-0121706, CNS-0509467, CCF-0541409, CCF-1059417, CCF-0073800, EIA-9986052, and EPS-1003897.

*Corresponding author. Tel.: +1 225 578 2191; fax: +1 225 578 1465.

[1]Current address: Software and Services Group, Intel Corporation, 2111 NE 25th Ave., Hillsboro, OR 97124, USA

[2]Current address: IBM Silicon Valley Lab., 555 Bailey Ave., San Jose, CA 95141, USA

[3]Current address: Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA 99352, USA

the best features of empirical optimizations, namely, the incorporation of complex behavior of modern architectures, and a model-driven approach that enables efficient exploration of the search space.

Our domain of interest is the calculation of electronic structure properties using *ab initio* quantum chemistry models, such as the coupled cluster models [48]. We have developed an automatic synthesis system called the Tensor Contraction Engine (TCE) that generates efficient parallel programs from high-level expressions for a class of computations expressible as tensor contractions [6, 26, 16, 15]. Tensor contractions are essentially matrix multiplications generalized to higher-dimensional arrays (i.e., tensors). The computation is represented by an expression tree, in which each node represents the contraction of two tensors to produce a result tensor. The order of indices of the intermediate tensors is not constrained. In contrast to our other papers on the TCE [6, 26, 16, 15], in this paper, we address the problem of effective code generation for tensor contractions (products of multi-dimensional arrays) in terms of calls to linear algebra libraries that are optimized for two-dimensional arrays for a variety of target architectures.

Computational kernels such as Basic Linear Algebra Subroutines (BLAS) [18] have been tuned to achieve very high performance. These hand-tuned or empirically optimized kernels generally achieve better performance than conventional general-purpose compilers [61]. Significant improvements in execution time can be obtained if program transformations can identify these computational kernels and use these libraries [3]. If General Matrix Multiplication (GEMM) routines available in BLAS libraries are used to perform tensor contractions, the multi-dimensional intermediate arrays that arise in tensor contractions must be transformed to group the indices to allow a two-dimensional view of the arrays, as required by GEMM. We observe that the performance of the GEMM routines is significantly influenced by the choice of parameters used in their invocation. We determine the layouts of the intermediate arrays and the choice of parameters to the GEMM invocations that minimize the overall execution time. The overall execution time is estimated from the GEMM and index permutation times. Empirically-derived costs for these constituent operations are used to determine the GEMM parameters and array layouts.

The approach presented in this paper may be viewed as an instance of the telescoping languages approach [33, 34, 11, 10]. The telescoping languages approach aims at facilitating a high-level *scripting* interface for a domain-specific computation to the user, while achieving high performance that is portable across machine architectures, and compilation time that only grows linearly with the size of the user script. In this paper, we evaluate the performance of the relevant libraries empirically. On distributed-memory machines, parallel code is generated using the Global Arrays (GA) library [54, 24, 53]. Parallel matrix multiplication is performed using the Cannon matrix multiplication algorithm [9, 25], extended to handle non-square distributions of matrices amongst the processors [21]. The matrix multiplication within each node is performed using GEMM. The parallel matrix multiplication and parallel index transformation costs are estimated from the local GEMM and transformation costs and the communication cost. We then use the empirical results to construct a performance model that enables the code generator to determine the appropriate choice of array layouts and distributions and usage modalities for library calls.

The paper is organized as follows. In Section 2, we elaborate on the computational context, demonstrate potential optimization opportunities, and then define our problem. Section 3 discusses the constituent operations in the computation and the parameters to be determined to generate optimal parallel code. Section 4 describes the determination of the constituent operation costs. Section 5 discusses the determination of the parameters of the generated code from the constituent operation costs. Results are presented in Section 6. Section 7 discusses related work. Section 8 concludes the paper.

2. The computational context

The Tensor Contraction Engine (TCE) [6, 26, 16, 15] is a domain-specific compiler for developing accurate ab initio models in quantum chemistry. The TCE takes as input a high-level specification of a computation expressed as a set of tensor contractions and transforms it into efficient parallel code. In the class of computations considered, the final result to be computed can be expressed as multi-dimensional summations of the product of several input arrays.

The TCE incorporates several compile-time optimizations, including algebraic transformations [45, 46] and common subexpression elimination [26] for minimizing operation counts, finding the optimal evaluation order [44] and loop fusion [43, 42] for reducing memory requirements, space-time trade-off optimization [15], data locality optimization, which combines loop fusion and tiling for reducing disc-to-memory traffic [22, 7, 40], and communication minimization [23, 16]. Regardless, of whether outer loops are fused or whether loops are tiled as a result of other

optimizations, the inner-most perfectly-nested loop nests can be implemented as a combination of generalized matrix multiplication and index permutation (multi-dimensional transposition) library calls. In this paper, we discuss the optimal selection of these library calls together with determining the layout of intermediate arrays.

Consider the following tensor contraction expression, a sub-expression of the coupled cluster singles and doubles (CCSD) equation [48],

$$S[a,b,i,j] = \sum_{c,d,k,l} A[a,b,k,l] \times B[c,d,k,l] \times C[c,i] \times D[d,j]$$

where all indices range over N. The direct computation of this expression using eight perfectly nested loops would require $O(N^8)$ arithmetic operations. Instead, by computing the following intermediate partial results, the number of operations can be reduced to $O(N^6)$.

$$\begin{aligned} T1[d,i,k,l] &= \sum_{c} B[c,d,k,l] \times C[c,i] \\ T2[i,j,k,l] &= \sum_{d} T1[d,i,k,l] \times D[d,j] \\ S[a,b,i,j] &= \sum_{k,l} A[a,b,k,l] \times T2[i,j,k,l] \end{aligned}$$

It is possible for different (non-equivalent) operation-minimal codes to have different execution times, or even for a non-operation-minimal code to out-perform the operation-minimal ones. However, because of the computational complexity of operation minimization, we currently only use a single operation-minimal form as a starting point for later optimizations.

Each of the summation expressions above is the contraction of a pair of tensors. As in matrix multiplication, the summation indices in a tensor contraction occur twice while non-summation indices occur once. Tensor contraction can, therefore, be viewed as a generalization of matrix multiplication.

Since efficient tuned library Generalized Matrix Multiplication (GEMM) routines exist, it is attractive to translate the computation for each tensor contraction node into a GEMM call. However, that may require restructuring the layout (or permuting the indices) of the tensor, so that its layout can be viewed as a two-dimensional matrix.

The layout of an array is the order in which the elements are stored in adjacent locations of memory. If the adjacent elements in memory correspond to differences in the left-most index followed by the indices in the right, the array is said to be laid out in column-major order. Variation in indices from right-to-left is referred to as row-major order. Following Fortran convention, the argument and result matrices in a GEMM call are assumed to be laid out in column-major order. E.g., a matrix $M[i,j]$ with n rows and m columns is laid out in memory as follows:

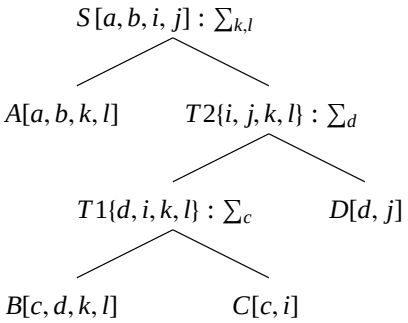
$$M[1,1], M[2,1], \ldots, M[n,1], M[1,2], \ldots, M[n,2], \ldots, M[1,m], M[2,m], \ldots, M[n,m]$$

The four-dimensional tensor $S[a,b,i,j]$ has the following layout in memory:

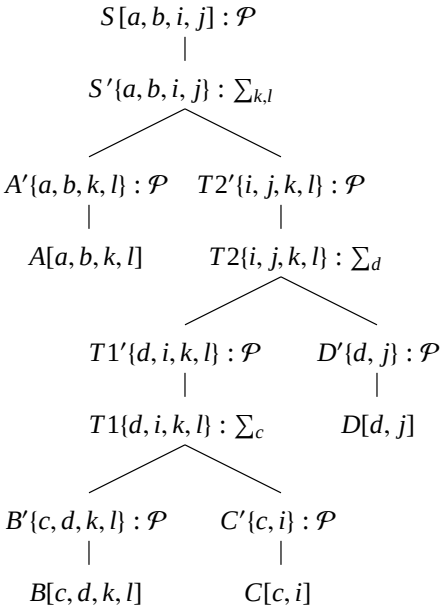
$$S[1,1,1,1], \ldots, S[N,1,1,1], S[1,2,1,1], \ldots, S[N,2,1,1], \ldots, S[1,N,1,1], \ldots, S[N,N,1,1], S[1,1,2,1], \ldots, S[N,N,N,N]$$

Since the elements for the first two dimensions are consecutive in memory for fixed values of i and j, this four-dimensional tensor can be viewed as a three-dimensional $N^2 \times N \times N$ tensor $S[ab,i,j]$, where the meta index ab is computed as $a + N(b-1)$. In general, any consecutive indices can be grouped into a meta index. This tensor, therefore, can be viewed as either of the matrices $S[abi,j]$, $S[ab,ij]$, or $S[a,bij]$.

Each of the matrix arguments in a GEMM call has one summation and one non-summation index. In a normal invocation, the summation index is the right index of the left argument and the left index of the right argument, but GEMM allows one or both arguments to be supplied in transposed form. For a multi-dimensional tensor to be viewed as a two-dimensional matrix, the summation and non-summation indices in the contraction must be grouped into two contiguous sets of indices. This may involve changing the layout of the tensor to meet the requirements of the GEMM call. For example, the tensor $A[a,b,k,l]$ with summation indices k and l can be viewed as the two-dimensional $N^2 \times N^2$ matrix $A[ab,kl]$ with non-summation meta index ab and summation meta index kl. In this case, no layout transformation is necessary.



(a) Expression tree



(b) Expanded expression tree

Figure 1: Expression tree for a sub-expression in the CCSD equation. (a) Original expression tree (b) The expanded expression tree used for layout optimization

When the tensor expressions are executed in parallel, the arrays need to be distributed amongst the processors. Optimized implementations of parallel matrix-matrix multiplication, such as ScaLAPACK [13], employ a blocked Cartesian distribution of the two-dimensional arrays onto a square processor grid.

For the above 3-contraction example, the first contraction can be implemented directly as a call to GEMM with B viewed as an $N \times N^3$ rectangular matrix $B[c, dkl]$ and C as an $N \times N$ matrix. For the GEMM call, B is provided as a transposed operand and C as a normal operand. The intermediate $T1$ resulting from this GEMM call is generated as the two-dimensional matrix $T1[dkl, i]$, which corresponds to the tensor $T1[d,k,l,i]$. For the second GEMM call, $T1[d,k,l,i]$ can be viewed as an $N \times N^3$ rectangular matrix $T1[d, kli]$ and D as an $N \times N$ matrix. The resulting intermediate $T2$ is generated as the two-dimensional matrix $T2[kli, j]$, which corresponds to the tensor $T2[k,l,i,j]$, which is then viewed as an $N^2 \times N^2$ matrix $T2[kl, ij]$ for the third GEMM call. By swapping the left and right operands to the third GEMM call, the result tensor S can be generated in the desired layout as $S[a,b,i,j]$.

However, suppose that the input tensor B is provided as $B[d,c,l,k]$. Since the summation index c is neither the left-most nor the right-most index, the layout of the tensor must be restructured. E.g., the indices could be permuted to result in $B'[c,d,l,k]$, which can then be used as a transposed operand in the GEMM call as above. Alternatively, the indices could be permuted to result in $B'[d,l,k,c]$, which is then used as a normal operand in the GEMM call. As we will see in the next section, the execution times for these two combinations of index permutation and matrix multiplication can be significantly different. In either case, however, $T2$ will eventually be generated as $T2[l,k,i,j]$, which requires an additional index permutation. If the index permutation of B is chosen to result in either $B'[c,d,k,l]$ or $B'[d,k,l,c]$, the index permutation of $T2$ can be avoided.

We represent this computation as an expression tree. As an example, Fig. 1(a) shows the expression tree for the sub-expression from the CCSD model discussed above. The root of the tree is the output array; input arrays form the leaf nodes. Interior nodes correspond to contractions. For contraction nodes, we indicate the indices that are summed over with a summation symbol. The layouts of leaf nodes and of the root of the tree are specified by the user and are fixed in column major order of the indices. The array reference labels shown for interior nodes are for convenience in referring to the result of the contraction, but do not constrain the layout, which we indicate with curly braces.

An intermediate array can benefit from different layouts when it is produced as the output of one GEMM call and consumed as input in another GEMM call. This translates into the possibility of an index permutation for each intermediate node. We represent this possibility in an expanded expression tree that is derived from the original

expression tree. Each intermediate node in the original expression tree is duplicated in the expanded expression tree, with the edge between the original and duplicate node corresponding to an an array reshaping or index permutation operation, labeled $\mathcal{P}$, on top of each of the original nodes. For example, Fig. 1(b) shows the expanded expression tree derived from the expression tree in Fig. 1(a). These duplicated nodes will be referred to as array reshape nodes.

If the computation does not fit into the available memory, we employ loop fusion to minimize the memory requirements. By fusing a loop that is in common between the producer and consumer of an intermediate array, the corresponding dimension can be removed from the intermediate. E.g., by fusing the l loop between the production and consumption of $T1$ above, $T1$ can be reduced to a three-dimensional tensor $T1[d, i, k]$. A fully fused computation, however, can be significantly slower because of poor cache behavior. E.g., if $T1$ is fused to a scalar, it is no longer possible to use a GEMM call to compute it. After loop fusion, we tile the fused loops and expand fused dimensions to tile size. By moving all intra-tile loops inside the fused tiling loops, we are left with perfectly nested loop nests inside the tiling loops that can then be replaced by combinations of GEMM and index permutation calls. Since the layout optimization problem presented in this paper is independent of whether loop fusion and tiling are employed or not, we restrict our attention to expanded expression trees as shown in Fig. 1(b). Layout optimization has an effect on memory usage as well, since index permutations are not performed in place. However, since buffers can be allocated and deallocated dynamically, this effect is minimal and can be ignored.

Instead of implementing the computation represented by such an expression tree as a sequence of GEMM calls interspersed with array reshaping operations, it is also possible to implement it directly as a collection of loop nests, one for each node of the expression tree. Optimizing the cache behavior of the resulting collection of a large number of loop nests directly, however, is a difficult challenge. In general, the best performance can be obtained by implementing a sequence of multi-dimensional tensor contractions using GEMM and array reshaping operations.

The problem addressed in this paper is the following:

> Given a sequence of tensor contractions (expressed as an expanded expression tree), determine the layouts (i.e., dimension order) of the intermediate tensors, the distributions (among multiple processors) of all tensors, and the modes of invocation of GEMM so that the specified computation is executed in minimal time.

To facilitate the data exchange between the TCE-generated code and a quantum chemistry package, the layouts of the input and output tensors are determined by their declarations in the TCE input specification. Since in a distributed implementation inputs and outputs are stored on disk because of their size, their optimal distributions can be determined by our algorithm.

In a sequential implementation, a tensor contraction node is implemented as a GEMM call and an array reshaping operation is an index permutation (or a no-op), as shown in the example above. In a distributed implementation, a tensor contraction node is implemented as a parallel matrix multiplication and an array reshaping operation may involve a redistribution among the processors in addition to the index permutation. The details of the optimization parameters are explained in the next section.

Since for an n-dimensional tensor there are $n!$ possible index permutations and since there are four possible choices for each GEMM invocations, and in the parallel case multiple possible distributions, the search space is very large. Simple heuristics do not work, because a wrong layout decision can cause the need for additional index permutations for ancestor nodes. E.g., choosing the wrong index permutation for B above resulted in an additional index permutation for $T2$. However, as we will show in Section 5, the constraints imposed on the layouts by GEMM reduce the size of the search space drastically and result in an algorithm that is linear in the size of the tree.

3. Constituent operations

In this section, we discuss the various operations within the computation and their influence on the execution time. The parameters that influence these costs, and hence the overall execution time, are detailed.

3.1. General Matrix Multiplication (GEMM)

Matrix multiplication is one of the most important computational kernels. It has been tuned, sometimes at the assembly language level, to achieve close-to-peak performance by machine and library vendors. These libraries obtain

Table 1: Configuration of the Itanium 2 cluster at the Ohio Supercomputer Center

Node	Memory	OS	Compilers	TLB	Network Latency	Interconnect	Comm. Library
Dual 900MHz Itanium 2	4GB	Linux 2.4.21smp	g77, ifc	128 entry	17.8μs	Myrinet 2000	ARMCI

significantly higher performance than code generated for these kernels by general-purpose compilers. Recognizing matrix multiplication operations and using these libraries can, therefore, significantly improve the generated code.

General Matrix Multiplication (GEMM) is a set of matrix multiplication subroutines in the BLAS library. It is used to compute

$$C = alpha * op(A) * op(B) + beta * C$$

In this paper, we use the double precision version of the GEMM routine of the form

$$\text{DGEMM}(ta, tb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)$$

where *ta* (*tb*) specifies whether A (B) is in transposed form. When *ta* is 'n' or 'N', $op(A) = A$; when *ta* equals 't' or 'T', $op(A) = A^T$; *alpha* and *beta* are scalars; C is an $M \times N$ matrix; $op(A)$ and $op(B)$ are matrices of dimensions $M \times K$ and $K \times N$, respectively. The matrices are assumed to have a column-major memory layout. The parameters *lda*, *ldb*, and *ldc* specify the sizes of the leading dimensions of matrices A, B, and C, respectively.

To illustrate the performance characteristics of DGEMM, we measured its performance with varying input parameters on the Itanium 2 Cluster at the Ohio Supercomputer Center (Dual 900MHz processors with 4GB memory, interconnected by Myrinet 2000 network). The cluster's configuration is shown in Table 1. The latency measurement of the interconnect was obtained from the ARMCI Web page [1, 55].

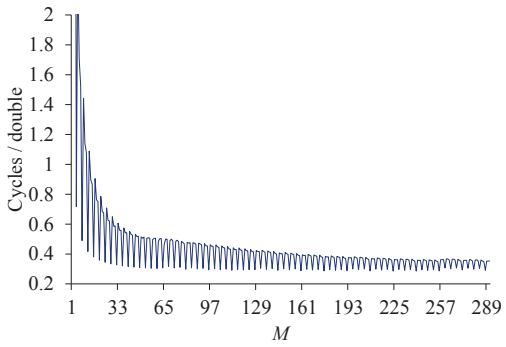
Matrix multiplications of the form $A*B$ were performed, where B was a 4000×4000 matrix and A was an $M\times4000$ matrix, with M varied from 1 to 300. Matrix multiplications involving such oblong matrices are quite typical in quantum chemistry computations, where for a multi-dimensional tensor only one dimension might be contracted. Two BLAS libraries were evaluated on the Itanium 2 Cluster, the Intel Math Kernel Library (MKL) 9.0 [28] and ATLAS 3.6.0 [59]. The *tb* argument was specified as 't' for the results shown in Fig. 2(a) and Fig. 3(a). Fig. 2(b) and Fig. 3(b) show the results for *tb* being 'n'. The x-axis shows the value of M and the y-axis shows the matrix multiplication execution time per double word in clock cycles. The measurements were averages over a large number of runs. The steady-state performance after the caches were warmed up showed very little variation and is representative of quantum chemistry computations with large tensors.

We observe that the performance characteristics of the DGEMM operation differs between different libraries and between the transposed and untransposed versions, and that the cost of the transposed version cannot be interpreted as the sum of the cost of transposition and the cost of the untransposed version. For example, in some of the experiments with the ATLAS library, the transposed version performs better. Furthermore, the performance can vary quite drastically for small changes in dimension size. It is, therefore, not feasible to develop an accurate analytical cost model that allows the compiler to predict the performance for given dimension sizes and invocation mode.

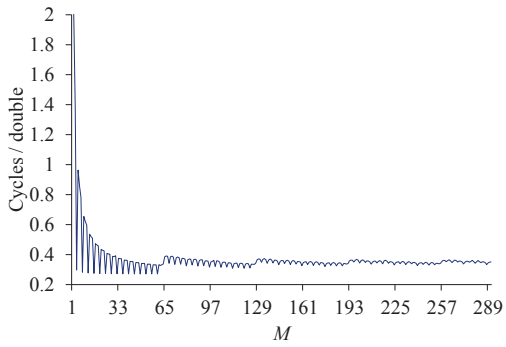
The cost of GEMM is influenced by the choice of its parameters. For any given tensor contraction, the summation indices are fixed and cannot be modified. The stride parameters (*lda*, *ldb*, and *ldc*) as well as the scalars are not beneficial for optimization purposes. The input arrays, however, may or may not be transposed, and their relative order could be changed. Thus, these three parameters need to be determined for each GEMM invocation for optimizing the overall execution time. Since an analytical cost model would not be accurate enough, these parameters are determined based on performance measurements.

Note that the layout of input and output arrays for a DGEMM invocation uniquely determines its parameters. Thus the problem of determining the DGEMM parameters is equivalent to determining the layouts of all intermediates.

For certain input sizes either BLAS implementation may outperform any others. It would, therefore, also be possible to let the compiler determine which GEMM implementation to use for a given invocation. However, since the vendor libraries typically outperforms ATLAS on average, especially for newer versions of the vendor libraries, we chose not to make this choice an optimization parameter.

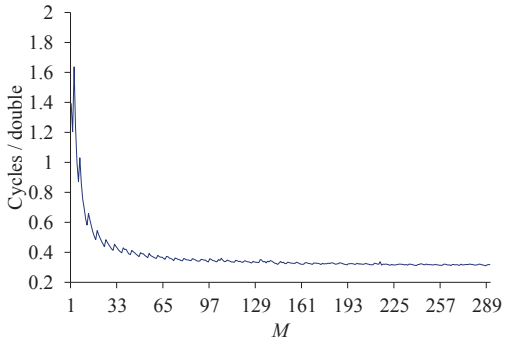


(a) MKL (‘n’, ‘t’)

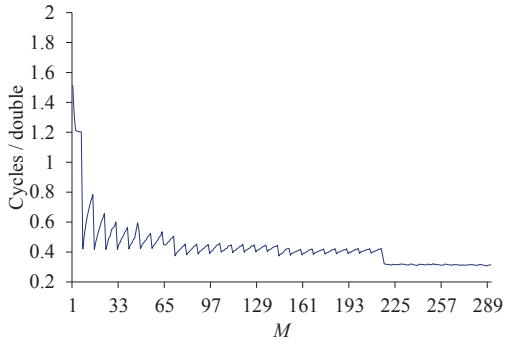


(b) MKL (‘n’, ‘n’)

Figure 2: The matrix multiplication times using DGEMM from the MKL library for $C = AB$ where C is of size $M \times N$, A is of size $M \times K$, B is of size $K \times N$, and $K = N = 4000$. (a) *tb*=’t’ (b) *tb*=‘n’ in input argument to DGEMM. M is varied along the x-axis. The y-axis shows the execution time per double word in clock cycles.



(a) ATLAS (‘n’, ‘t’)



(b) ATLAS (‘n’, ‘n’)

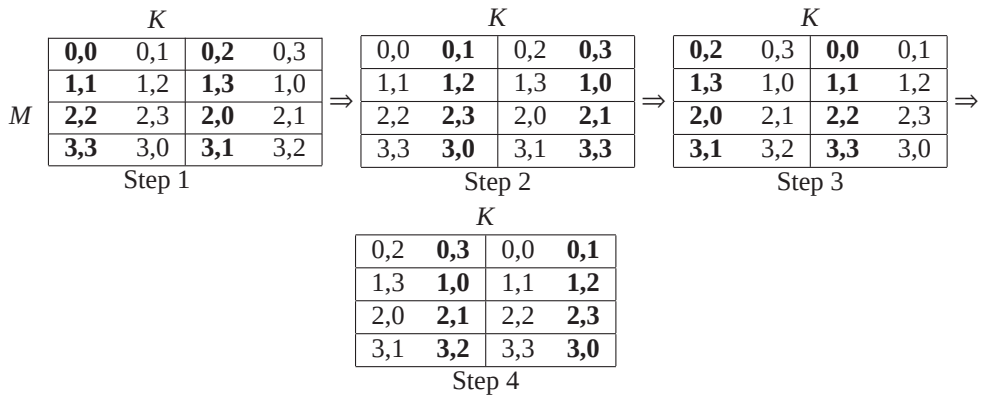
Figure 3: The matrix multiplication times using DGEMM from the ATLAS library for $C = AB$ where C is of size $M \times N$, A is of size $M \times K$, B is of size $K \times N$, and $K = N = 4000$. (a) *tb*=‘t’ (b) *tb*=‘n’ in input argument to DGEMM. M is varied along the x-axis. The y-axis shows the execution time per double word in clock cycles.

Note that on computers with symmetric multiprocessing (SMP), where GEMM routines are often parallelized using OpenMP, the above observations about the behavior of GEMM apply as well.

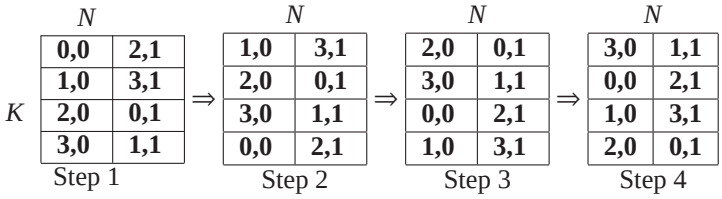
3.2. Cannon’s matrix multiplication algorithm

On computer clusters, i.e., in the case of distributed memory, several approaches have been proposed for implementing parallel matrix multiplication [58, 18]. In this paper, we consider an extension to Cannon’s algorithm for parallel matrix multiplication [21, 9, 25] that removes the restriction of using a square grid of processors for array distribution. The local matrix multiplications on each processor are performed using DGEMM. While our layout optimization approach can be applied to any parallel matrix multiplication algorithm whose execution time can be empirically modeled, we chose Cannon’s algorithm because it is very efficient with memory usage and communication, allows computation and communication to be overlapped, and provides flexibility for optimization.

Cannon’s algorithm assumes a logical view of a group of processors in which the processors form a two-dimensional square grid. Let $C(M, N) \mathrel{+}= A(M, K) * B(K, N)$ be the multiplication being performed on a $\sqrt{P} \times \sqrt{P}$ processor grid. The input arrays are distributed among the processors in an identical fashion with each processor holding one block each of A, B, and C. The algorithm proceeds in $\sqrt{P}$ steps. At the beginning of the first step, the blocks of A in the



(a) Array A



(b) Array B

Figure 4: The processing steps in the extended Cannon Algorithm. Initially, processor P_{ij} holds blocks labeled B_{ij} and $A_{i(j:j+1)}$. The portion of data accessed in each step is shown in bold.

i-th row of the processor grid are rotated left i positions, and those of B are rotated up i positions. At the beginning of each subsequent step, every block of A is rotated left one position, and every block of B is rotated up one position. In each step, the local blocks of A and B are multiplied and added to the local block of C.

Since Cannon's algorithm does not replicate portions of the input matrices on multiple processors, it uses memory very efficiently. Individual matrix blocks can be as large as one fifth of the storage available on a processor, with one block from each of the three matrices to perform the computation and storage for two more blocks to allow communication to be overlapped with computation. Our extension generalizes the algorithm to allow for logical views of a group of processors as rectangular processor grids and, unlike Lee et al.'s extension [47], it allows any of the three indices to be the rotation index and limits the communication time to $O(P + Q)$ for a processor grid of size $P * Q$. Further details about our extension of Cannon's algorithm can be found in [21].

The extended Cannon algorithm for a 4×2 processor grid is illustrated for the matrix multiplication $C(M, N)$ += $A(M, K) * B(K, N)$ in Fig. 4. The processors form a logical rectangular grid. All the arrays are distributed amongst the processors in the grid. Each processor holds two blocks of A and one block of each of the arrays B and C. The algorithm divides the common dimension (K in this illustration) to have the same number of sub-blocks. Each step operates on a sub-block and not on the entire data local to each processor. In each step, if the sub-block required is local to the processor, no communication is required. Fig. 4 shows in bold the sub-blocks of arrays A and B accessed in each step. It shows that the entire array B is accessed in each step.

Given a processor grid, the number of steps is given by the number of sub-blocks along the common dimension (K). The number of blocks of A that are needed by one processor corresponds to the number of processors along the common dimension, and that of B correspond to the other dimension. Table 2 illustrates the number of steps and the number of remote blocks required per processor for the possible distributions with sixteen processors. The number of steps and the number of remote blocks required per processor depend on the distribution of the arrays amongst the processors. The block size for communication is independent of the dimensions. It can be seen that different distributions have different costs for each of the components.

The relative sizes of the arrays A and B determine the optimal distribution. When one array is much larger than

Table 2: Extended Cannon algorithm's per-processor costs for different distributions of a 16-processor grid

Distribution	#steps	#blocks communicated	
		Array A	Array B
1 * 16	16	15	0
2 * 8	8	7	1
4 * 4	4	3	3
8 * 2	8	1	7
16 * 1	16	0	15

the other, the cost can be reduced by skewing the distribution to reduce the number of remote blocks accessed for that array. The shape of the array that is local to each processor affects the local DGEMM cost. Thus, the array distribution influences the communication and computation costs and is an important parameter to be determined.

The optimization parameters for a parallel matrix multiplication using our extended version of Cannon's algorithm, therefore, are the shape of the processor grid and the selection of distribution index for each processor grid dimension (the remaining index is the rotation index) in addition to the optimization parameters for the local DGEMM call.

3.3. Index permutation

DGEMM requires a two-dimensional matrix view of the inputs. This means that the summation and non-summation indices of a tensor must be grouped into two contiguous sets of indices. The layout of a multi-dimensional tensor, therefore, might have to be transformed to be used as input to DGEMM. Further, an additional index permutation cost might be worth paying if it can reduce the DGEMM cost through the use of a transposed (or non-transposed) argument form.

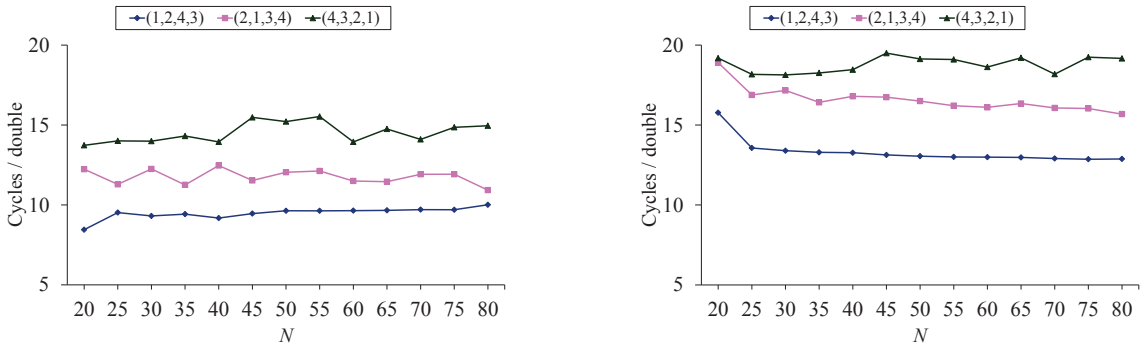
For each architecture of interest, we implemented a collection of index permutation routines, one each for a given number of dimensions. On architectures without SIMD (single instruction, multiple data) support such as Itanium 2, the routines were tiled in the fastest varying indices in the source and target arrays. We observed that performing the computation such that the target arrays are traversed in the order of their storage resulted in better performance than biasing the access to the source array. This can be explained by the write-back L2 cache in the system, which causes write-back of dirty blocks when they are replaced. Replacement of read blocks does not pay this penalty. Thus fully writing a cache line of the target array reduces the cost due to these write-backs. The execution times for different tile sizes was determined and the best tile size was chosen. The performance of the routines was evaluated on a number of permutations to determine the tile sizes.

On the Itanium 2 platform, we measured the execution times of these routines for some index permutations on four-dimensional arrays of size $N \times N \times N \times N$, with N varying from 15 to 85. The measurements were averages over a large number of runs. The steady-state performance after the caches were warmed up showed very little variation. The results are shown in Fig. 5.

Unlike the cost of DGEMM, which for larger tensors is less than half a clock cycle per double word, the cost of index permutation is in the range of 10–20 cycles per double word. GEMM implementations can achieve such a high performance, since they use tiling for all levels in the memory hierarchy to optimize temporal locality. Since tensor elements only get copied but not reused during an index permutation, there cannot be any temporal locality. Even though index permutation is only an $O(N^2)$ operation while GEMM is an $O(N^3)$ operation, for the dimension sizes of interest the cost of index permutation can be a significant part of the total cost.

Different permutations are observed to incur different costs. These differences are due to variations in spatial locality because of differences in the memory access patterns of different permutations. We also notice that the use of different compilers leads to differences in performance.

On architectures such as recent Intel x86 processors, where SIMD instructions are available, we generate index permutation routines following an automatic approach described in [51, 26, 50]. The basic idea is to apply loop tiling at different cache/TLB levels and then to search automatically for the optimal loop order, tile sizes, and SIMD code sequence for index permutation.



(a) Intel Fortran compiler

(b) g77 compiler

Figure 5: Index permutation times on Itanium 2 for three different permutations for an $N \times N \times N \times N$ matrix using (a) Intel Fortran compiler (b) g77 compiler. N is varied along the x-axis. The y-axis shows the execution time per double word in clock cycles.

The layout of the arrays influences the index permutation costs and is the parameter to be determined to evaluate the index permutation cost. Parallel index permutation can be viewed as a combination of local index permutation and array redistribution. The extended Cannon algorithm requires that the summation and non-summation index groups be distributed along the slowest varying index in that group. The number of processors along the dimension in the processor grid corresponding to a group can also be varied to determine the shape/size of arrays used in the local DGEMM calls. Thus, in addition to the layout of the arrays, their distribution needs to be determined as well.

4. Empirical measurement of constituent operations

4.1. GEMM cost

For determining the precise cost of GEMM, it must be executed with a range of parameter choices on the target machine. For the purpose of this paper, we execute it at compile time for any parameter combinations considered by our layout optimization algorithm. For small examples, such as the expression tree for the CCSD sub-expression in Fig. 1(b), this is clearly prohibitively expensive, since it can result in compilation times that are larger than the execution time by testing multiple GEMM parameter choices for each contraction node in the tree. Realistic tensor contraction expressions in quantum chemistry, however, consist of dozens to hundreds of sub-expressions of this form with repeated occurrences of the same tensor sizes. Caching the results of any GEMM measurements, therefore, will be very effective. Furthermore, quantum chemistry computations often require out-of-core treatment [7, 40, 41], where tiles of multi-dimensional arrays are brought into memory and operated upon. These loops are in turn enclosed in an outermost loop in iterative chemical methods. Thus, each contraction node in the expression tree will correspond to multiple invocations of GEMM. Even with performing GEMM measurements at compile time, the compilation time will still be much less than the execution time.

Alternatively, the GEMM performance measurements could be performed offline at TCE installation time. Since the range of tensor dimension sizes is limited and since quantum chemistry models typically use no more than eight dimensions and no more than two different dimension sizes per tensor, this would not lead to a combinatorial explosion. Furthermore, for larger dimension sizes it would be possible to interpolate between measurements. The spikes observed in our MKL measurements in Fig. 2 occurred when M was a multiple of four. Using special treatment for multiples of four and a careful selection of measured values can, therefore, make interpolation very precise. While even with interpolation a few thousand measurements might be needed, this would be an acceptable one-time cost.

4.2. Cannon's matrix multiplication

The cost of parallel matrix multiplication using Cannon's algorithm is the sum of the computation and the communication costs. Since the local computation is performed using DGEMM, the computation cost can be derived from

the DGEMM cost. The communication cost is the sum of the communication costs at the beginning of each step. We summarize the model employed for the Cannon's matrix multiplication algorithm [21]. A latency-bandwidth model is used to determine the communication cost. Consider the matrix multiplication $C(M,N) \mathrel{+}= A(M,K) * B(K,N)$ and assume that K is the rotation index. Let P_M, P_K, P_N be the number of processors into which the array is distributed along the M, N, and K dimensions, respectively. The total communication cost is given by

$$\begin{aligned} CommnCost_A &= \left(T_s + \frac{M * K}{BW * P_M * P_K}\right) * (P_K - P_K/P_M) \\ CommnCost_B &= \left(T_s + \frac{K * N}{BW * P_K * P_N}\right) * (P_K - P_K/P_N) \\ CommnCost &= CommnCost_A + CommnCost_B, \end{aligned}$$

where $CommnCost_A$ and $CommnCost_B$ are the initialization and shift costs for matrices A and B, respectively. T_s is the latency of the interconnect shown in Table 1. BW, the bandwidth, is estimated from a table constructed from the bandwidth curve on the ARMCI Web page [1, 55]. Similar formulas are used if another rotation index is chosen.

4.3. Index permutation

Fig. 5 shows the performance of our index permutation routines for some permutations. The performance of the implementation appears to be relatively independent of the array dimensions, but is influenced by the permutation being performed.

An analysis of the implementation revealed that the variation in the per-element permutation cost was primarily influenced by the variation in the TLB misses for different permutations and the capability of compilers to perform efficient register tiling.

We estimated the index permutation cost to consist of two components. The first component is the basic copy cost, the minimum cost required to copy a multi-dimensional array, together with the index calculation. We determined two types of basic copy costs. The first, referred to as c_0, is the one in which both the source and target arrays are traversed with sufficient locality. The other basic copy cost, referred to as c_1, is one in which there is only locality in traversing the target array. Depending on the permutation and the size of the arrays, one of these basic copy costs is chosen. The basic costs c_0 and c_1 were found to be compiler dependent. On the Itanium 2 system, they were determined to be 9.5 and 11.3 cycles, respectively, per double word with the Intel Fortran Compiler and 12.9 and 15.9 cycles, respectively, per double word with g77. The second component is the TLB miss cost. Each processor on the Itanium 2 cluster had an 128 entry fully-associative TLB with a miss penalty of 25 cycles. Different permutations can lead to different blocks of data being contiguously accessed and at different strides. The permutation to be performed and the array size are used to determine the TLB cost.

In the parallel version of the algorithm, index permutation is coupled with array redistribution. Transformation from one layout and distribution configuration to another is accomplished in two steps, a local index permutation followed by array redistribution.

A combination of index permutation and redistribution can result in each processor communicating its data to more than one processor. The communication cost is estimated differently for different cases. When the target patch written to is local to a processor no communication is required. When the layout transformation is such that each processor needs to communicate its data to exactly one other processor, the cost is uniform across all the processors and is estimated as the cost of communicating that block. In all other cases, we estimate the communication cost to be the cost incurred by the processor whose data is scattered among the most number of processors.

5. Composite performance model

In this section, we discuss how the empirical measurements of the constituent operations are used to determine the parameters that optimize the overall execution time.

5.1. Constraints on array layouts and distributions

Each of the input and output arrays is constrained to have one specified layout. Each array corresponding to a contraction node is produced as the output of a GEMM call. As explained earlier, GEMM operates on a two-dimensional view of the input arrays producing as output a two-dimensional array represented by an index order with the non-summation indices of the first input array followed by the non-summation indices of the second input. For the set of non-summation indices for each input array, all possible permutations are valid.

The array reshape (or index permutation) nodes act as inputs to the GEMM calls. Thus, the layout of arrays represented by the array reshape nodes is constrained by the restrictions imposed on the inputs to GEMM calls. All summation indices are grouped together and laid out contiguously, with some choices for the remaining indices.

In the expanded expression tree, each intermediate node is either a contraction node or an array reshape node. A contraction node corresponds to an array that is the output of the contraction and does not serve as the input to another GEMM call. An array reshape node corresponds to an array that is the input of a GEMM call and is produced as the result of an array reshape operation. In particular, it is not produced as the output of another GEMM call. Thus, each interior node is *associated* with exactly one contraction, with each contraction node corresponding to the output of the associated contraction and each array reshape node corresponding to input of the associated contraction.

We now formally define the set of possible permutations to be evaluated for each intermediate node. The summation (contraction) and non-summation indices in each contraction are identified as explained in Section 2. For each node n, we define the summation indices and non-summation indices as those in the associated contraction:

`SI(n)` Set of summation indices in contraction associated with node n

`NSI(n)` Set of non-summation indices contributed by node n to its associated contraction

Let $\mathbb{P}(s)$ denote the set of all permutations of the indices in index set s. The indices in each interior node can be partitioned into two groups. For a contraction node, these correspond to the set of indices from each of the input arrays. For an array reshape node, these correspond to the set of summation and non-summation indices. The two groups themselves can be permuted with one another, with either group appearing on the left.

Let i_1 and i_2 represent the input arrays contributing to contraction node n. The set of all valid permutations for each interior node is given by:

$$\mathcal{S}(n) = \begin{cases} \{[l,r],[r,l]\}\, \forall l \in \mathbb{P}(\texttt{NSI(}i_1\texttt{)}),\ r \in \mathbb{P}(\texttt{NSI(}i_2\texttt{)}) & \text{if } n \text{ is contraction node} \\ \{[l,r],[r,l]\}\, \forall l \in \mathbb{P}(\texttt{SI(n)}),\ \ r \in \mathbb{P}(\texttt{NSI(n)}) & \text{if } n \text{ is an index permutation node} \end{cases}$$

where $[l,r]$ is the concatenation of the ordered lists of indices l and r to produce a complete index list (and similarly for $[r,l]$). The size of the resulting set is given by:

$$|\mathcal{S}(n)| = \begin{cases} 2 \times |\mathbb{P}(\texttt{NSI(}i_1\texttt{)})| \times |\mathbb{P}(\texttt{NSI(}i_2\texttt{)})| & \text{if } n \text{ is contraction node} \\ 2 \times |\mathbb{P}(\texttt{SI(n)})| \times |\mathbb{P}(\texttt{NSI(n)})| & \text{if } n \text{ is an index permutation node} \end{cases}$$

where the size of permutation set $\mathbb{P}$, $|\mathbb{P}(s)|$ is $|s|!$. For example, consider the contraction node $T2$ in Fig. 1(b). The indices in $T2$ are grouped into $\{i,k,l\}$ and $\{j\}$ based on whether they are provided by the input array $T1'$ or by D'. The number of possible layouts is thus $2 * 3! * 1! = 12$. Similarly, the indices in the contraction node $T2'$ can be grouped into the non-summation indices $\{i, j\}$ and the summation indices $\{k, l\}$. The number of possible layouts is thus $2 * 2! * 2! = 8$.

The extended Cannon algorithm requires the processor grid for each tensor contraction to be Cartesian in nature. In addition, the children of each contraction node in the expression tree are required to have the same processor grid as that node. Thus, for each distribution of a contraction node, there is a corresponding distribution for its children. There is no restriction on the distribution of the contraction nodes themselves.

The indices in an intermediate tensor, corresponding to an interior node, are divided into two groups as explained above. Each of these groups of tensor indices form one meta index for the matrix view of the tensor. We assume a two-dimensional processor grid with blocked data distribution to match the needs of the Cannon algorithm. The choice of distribution of the tensor onto a processor grid involves picking one index from each group and partitioning it amongst the processors. These indices are referred to as the distribution indices. A choice of distribution is uniquely defined by the processor grid employed in the distribution (e.g., 2×2, 1×4, etc.) and the distribution indices. For the

discussion of the algorithm, a distribution is defined to be the pair of processor grid and distribution indices; in other contexts, it is often synonymous with processor grid.

The block of data owned by a process can be laid out in any one of the possible permutations enumerated above. Note that the choice of data distribution amongst the processors is orthogonal to the choice of data layout within each process, with neither constraining the other. Both choices are only constrained by the role of the tensor in the GEMM operation it participates in.

5.2. Determination of optimal parameters

For the specified layouts of the root and leaves of the expanded expression tree, we determine the distributions of the root and leaves and the layouts and distributions of the intermediate arrays. The layouts of the root and leaves are fixed to facilitate the data exchange with a quantum chemistry package. For each layout of an array produced by GEMM, the arrays corresponding to its children nodes are required to have a compatible layout, i.e., the order in which the summation and non-summation indices are grouped is required to be identical in the produced and consumed arrays. This is because GEMM does not perform any index permutation within a group. Such layouts of the input arrays are said to be compatible with the given layout of the array output by the contraction node. Similarly, the distribution (processor grid and distribution indices) of the children must be compatible with the parent as explained above.

The algorithm proceeds by recursively computing the cost associated with a node for each layout and distribution as the sum of the costs associated with producing its inputs (children in the expanded expression tree) in all compatible layouts and distributions and transforming the layout and distribution to produce that node. This assumes that the performance of the individual operations is independent from one another. While this may not hold for small arrays due to cache effects, it is a valid assumption for the large tensors used in our computational domain. Note that the cost to compute a node for a given layout and distribution is independent of its use in another contraction or index permutation operation. Such a decoupled cost function enables a dynamic programming solution in which the costs are propagated in a bottom-up fashion with any given alternative configuration (layout and distribution) for each node evaluated at most once.

The configuration of an array n is represented by a distribution-layout pair (d, l). The cost of a node is determined as the least cost to compute its children and subsequently compute it from its children. The cost to evaluate node n, together with all intermediates in the subtree rooted at n, with distribution d and layout l is computed as follows:

$$C_t(n,d,l) = \begin{cases} \min_{\forall d' \in \mathcal{D}(n), \forall l' \in \mathcal{L}(n)} C_t(i_1,d',l') + C_{ip}((i_1,d',l') \rightarrow (n,d,l)) \\ \qquad\qquad \text{if } n \text{ is a index permutation node} \\ \min_{\forall l', l'' \in \mathcal{L}(n)} C_t(i_1,d,l') + C_t(i_2,d,l'') + C_{dg}((i_1,d,l') \times (i_2,d,l'') \rightarrow (n,d,l)) \\ \qquad\qquad \text{if } n \text{ is a contraction node} \end{cases}$$

where

$$\begin{aligned} C_t &\equiv \text{Total cost of computing a node with given distribution and layout} \\ C_{ip} &\equiv \text{Cost of the required index permutation} \\ C_{dg} &\equiv \text{Cost of the required DGEMM invocation} \\ \mathcal{D}/\mathcal{L} &\equiv \text{All feasible distributions/layouts of a node} \\ i_1/i_2 &\equiv \text{Left/right child of } n \end{aligned}$$

As shown by the above expressions, the total cost of computation of each non-leaf node, for different configurations, can be determined from the cost of computing its children from the leaf nodes and the cost of the basic operation, index permutation or GEMM, to compute the node from its children. The algorithm first determines the feasible layouts for each of the nodes in the expanded expression tree. The optimal cost of the root node is subsequently computed using the dynamic programming formulation.

A traversal of the tree together with memoization of the intermediate results for all valid layouts and distributions ensures that each node in the expanded expression tree is traversed exactly once. Thus, the cost incurred by this algorithm is linear in the number of nodes in the tree. For each contraction node, the layout chosen for the output array constrains the layouts of its inputs to two choices — whether each is transposed or not. Thus the evaluation of

Table 3: Configuration of the Intel Xeon workstation

Processor	Num. of Processors	Memory	OS	Compiler	BLAS Library
Intel Xeon E5530 2.4GHz quad-core	1	4GB	Linux 2.6.31	Intel Compiler 11.1	Intel Math Kernel Library 10.2

Table 4: Configuration of the AMD Opteron workstation

Processor	Num. of Processors	Memory	OS	Compiler	BLAS Library
AMD Opteron 8218 2.6GHz dual-core	4	16GB	Linux 2.6.18	Intel Compiler 10	AMD Core Math Library 14.3.0

a GEMM node considers four alternatives ((‘n’, ‘n’), (‘n’, ‘t’), (‘t’, ‘n’), (‘t’, ‘t’)) for each layout. The evaluation of the cost of an index permutation node needs to consider more alternatives. In particular, the number of alternatives is given by the cross product of all possible layouts of the input and output arrays. Given that the number of possible layouts in itself is non-linear, this operation could be expensive to compute. We overcome this limitation using the insights gleaned from Section 3.3. The similarity in the costs per element for a variety of permutations is used to categorize the permutations into a small number of choices, each of which is evaluated in terms of the basic copy cost and the TLB miss cost.

Thus, the only non-linear component in the cost associated with the approach presented is the factor corresponding to the number of possible layouts of an output array to be evaluated in each contraction node. Even this factor is eliminated when we consider the performance profile associated with a GEMM call. Recall that the GEMM call does not recognize the different layouts of indices within the two groups of indices — one each corresponding to the non-summation indices from the two input arrays. In particular, all such layouts result in array dimensions of the same size when viewed as a two-dimensional array. Therefore, all such layouts incur the same GEMM cost, and only the four choices corresponding to possible transpositions of inputs need to be evaluated.

The cost incurred by the algorithm is, therefore, linear in the number of nodes. The cost associated with each node is a small constant, corresponding to the few alternatives to be evaluated.

6. Experimental results

We evaluated our approach on three different systems. The only distributed-memory computer is the Itanium 2 cluster at the Ohio Supercomputer Center, whose configuration is shown in Table 1. The other two systems that we used are a single-processor four-core Intel Xeon E5530 workstation and a four-processor dual-core AMD Opteron 8218 workstation. The configuration details of these two systems are listed in Tables 3 and 4, respectively. We believe these two workstation systems represent the typical environments where small- to mid-scale tensor contraction computations are carried out while large computations are often performed on clusters. All the experiment programs were compiled with the Intel Fortran Compiler for its better performance. Our extension of Cannon’s algorithm was only used on the Itanium 2 cluster. We used the natively optimized BLAS library for each system. With Intel processors it was the Intel Math Kernel Library (MKL) [28] and with AMD processors it was the AMD Core Math Library (ACML) [4]. We did not use ATLAS to report our results as recent CPU vendor-provided libraries often perform better than ATLAS and most large-scale computational environments have CPU vendor-provided libraries pre-installed and maintained. We measured the constituent operations on all three platforms and performed benchmarks using the following two computations in our domain:

CCSD. We used a typical sub-expression from the CCSD theory for determining electronic structures. It is the same example as in Fig. 1, except that the layouts of the input and result tensors have been reversed to allow interfacing

Table 5: Layouts and distributions for the CCSD computation for the unoptimized and optimized versions of the code

Array	Unoptimized				Optimized			
	Distribution/ Proc. Grid	Dist./ Indices	Layout	GEMM Parameters	Distribution/ Proc. Grid	Dist./ Indices	Layout	GEMM Parameters
A	$(2,2)$	(k,a)	(l,k,b,a)	–	$(1,4)$	(k,a)	(l,k,b,a)	–
A'	$(2,2)$	(a,k)	(b,a,l,k)	–	–	–	–	–
B	$(2,2)$	(c,k)	(d,c,l,k)	–	$(1,4)$	(c,k)	(d,c,l,k)	–
B'	$(2,2)$	(k,c)	(d,l,k,c)	–	$(1,4)$	(c,k)	(c,d,l,k)	–
C	$(2,2)$	(i,c)	(i,c)	–	$(1,4)$	(i,c)	(i,c)	–
C'	$(2,2)$	(c,i)	(c,i)	–	–	–	–	–
D	$(2,2)$	(j,d)	(j,d)	–	$(1,4)$	(j,d)	(j,d)	–
D'	$(2,2)$	(d,j)	(d,j)	–	–	–	–	–
$T1$	$(2,2)$	(k,i)	(d,l,k,i)	B',C',(‘n’, ‘n’)	$(1,4)$	(i,k)	(i,d,l,k)	C,B',(‘n’, ‘n’)
$T1'$	$(2,2)$	(i,d)	(l,k,i,d)	–	$(1,4)$	(d,k)	(d,i,l,k)	–
$T2$	$(2,2)$	(i,j)	(l,k,i,j)	$T1'$,D',(‘n’, ‘n’)	$(1,4)$	(j,k)	(j,i,l,k)	D,$T1'$,(‘n’, ‘n’)
$T2'$	$(2,2)$	(k,j)	(l,k,i,j)	–	–	–	–	–
S'	$(2,2)$	(a,j)	(b,a,i,j)	A',$T2$,(‘n’, ‘n’)	$(4,1)$	(a,j)	(b,a,j,i)	A,$T2$, (‘t’, ‘t’)
S	$(2,2)$	(i,a)	(j,i,b,a)	–	$(1,4)$	(i,a)	(j,i,b,a)	–

with Fortran code.

$$S[j,i,b,a] = \sum_{l,k} A[l,k,b,a] \times \sum_{d} \left(\sum_{c} B[d,c,l,k] \times C[i,c] \right) \times D[j,d]$$

On the Itanium 2 cluster, all the array dimensions were 64 for the sequential experiments and 96 for the parallel experiments. On the two workstation systems, due to the smaller memory sizes, we used 64 and 80 for the sequential and parallel experiments, respectively.

AO-to-MO transform. This expression, henceforth referred to as the 4-index transform, is commonly used to transform two-electron integrals from atomic orbital (AO) basis to molecular orbital (MO) basis.

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

On the Itanium 2 cluster, the array dimensions were 80 and 96 for the sequential and parallel experiments, respectively. On the two workstation systems, we used 64 and 80 for the sequential and parallel experiments, respectively.

The chosen dimension sizes are fairly large. A four-dimensional tensor with dimension size 96 and double-precision floating point numbers as elements requires 648MB of storage. With dimension size 64, such a tensor requires 128MB of storage. Since the computations involve multiple tensors, any significant increase would require more processors or out-of-core computation.

We compared our approach with the baseline implementation in which an initial layout for the arrays is provided and, in case of a cluster with P processors, a fixed $\sqrt{P} \times \sqrt{P}$ array distribution is required throughout the computation. The order of parameters in the GEMM call is the same as the order of subtrees in the expression tree, with an (‘n’, ‘n’) invocation mode. This approach was, in fact, used in our early implementations. The optimized version is allowed flexibility in the distribution (but not the layout) of the input and output arrays.

Table 5 shows the configurations chosen for each array in the parallel experiment on the Itanium 2 system for the unoptimized and optimized cases. A first look reveals that the number of intermediate arrays is reduced by effective choice of layouts and distributions. The GEMM parameters for all three GEMM invocations are different, either in the order chosen for the input arrays or in the transposition of the input parameters. The distribution chosen for all the arrays is different from those for the unoptimized version of the computation.

As explained in Section 2, both B' and $T1'$ are viewed as $N \times N^3$ rectangular matrices for the purpose of GEMM, while $T2$ is viewed as an $N^2 \times N^2$ matrix. Because of the rectangular shapes of B' and $T1'$, the algorithm chose rectangular processor grids for all matrices. While a square $(2,2)$ processor grid would have been more efficient

Table 6: Sequential performance results for CCSD and 4index-transform on Itanium 2

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	55.28	1.41	56.69	45.58	0.78	46.36
4index	10.06	2.58	12.64	10.58	0.0	10.58

Table 7: Parallel performance results on 4 processors for CCSD and 4index-transform on Itanium 2

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	157.93	7.21	165.14	136.41	2.86	139.27
4index	12.23	7.74	19.97	7.57	3.64	11.21

Table 8: Sequential performance results for CCSD and 4index-transform on Intel Xeon

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	15.71	0.30	16.01	15.66	0.07	15.73
4index	1.19	0.22	1.41	1.30	0.0	1.30

for the GEMM call involving the square matrix view of $T2$, the elimination of the redistribution operation for $T2$ outweighed the benefit of a more efficient GEMM call.

The final array reshape operation from S' to S involves only a layout transformation and no redistribution, since the processor dimensions for indices i and j, respectively, contain only one processor. (Alternatively, the resulting distribution indices could have been chosen to be (j, a).)

Table 6 and Table 7 show the sequential and parallel results on Itanium 2, respectively. In the parallel CCSD experiments, the GEMM and index permutation times reported subsume the communication costs. The optimized version has close to 20% improvement over the unoptimized version in almost all cases. The parallel 4-index transform has an improvement of more than 75% over the unoptimized version. The effective choice of GEMM parameters results in a noticeable improvement in the GEMM cost for most cases. The index permutation cost is either improved or completely eliminated. The trade-off between the GEMM and the index permutation costs can be observed in the sequential 4-index transform experiment. In this experiment, the optimization process chooses an inferior configuration for the GEMM computation, so as to eliminate the index permutation cost completely, and hence reduce the overall execution time.

Table 8 and Table 9 show the sequential and parallel results on the Intel Xeon workstation, respectively. The parallel version has four threads each running on its own processor core. As index permutation is highly optimized on this processor, the sequential and parallel improvements with CCSD are 1.8% and 2.1%, respectively. However, the parallel 4-index transform still has an improvement of 32.1% over the unoptimized version. Table 10 and Table 11 show the sequential and parallel results on the AMD Opteron workstation, respectively. The parallel version has eight threads each running on its own processor core. Compared to the Intel Xeon workstation, on the AMD platform index permutation is not as highly optimized due to its multi-processor structure and other micro-architectural factors. Hence, we achieved a 147.7% improvement with the parallel 4-index code.

Fig. 6 and Fig. 7 show the relative improvements of the optimized over the unoptimized versions of CCSD and the 4-index transform, respectively, for all configurations. As these figures show, our algorithm in all cases reduces

Table 9: Parallel performance results on 4 processors for CCSD and 4index-transform on Intel Xeon

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	15.01	0.67	15.68	15.18	0.18	15.36
4index	0.99	0.42	1.43	1.07	0.0	1.07

Table 10: Sequential performance results for CCSD and 4index-transform on AMD Opteron

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	30.21	2.14	32.35	30.08	0.25	30.34
4index	2.14	0.56	2.70	2.52	0.0	2.52

Table 11: Parallel performance results on 4 processors for CCSD and 4index-transform on AMD Opteron

	Unoptimized (secs)			Optimized (secs)		
	GEMM	Index Permutation	Exec. Time	GEMM	Index Permutation	Exec. Time
CCSD	14.53	2.66	17.19	14.69	0.54	15.23
4index	2.70	1.24	3.94	1.59	0.0	1.59

or eliminates the index permutation cost. In some cases, it chooses less efficient DGEMM modes in exchange for eliminating the index permutation cost, while in several instances it is also able to improve the DGEMM performance.

Our measurements show that our layout optimization is beneficial for any architecture, multi-core, SMP, or cluster. Since DGEMM performs best on large arrays, and since even for computations that are near the memory limit we see significant improvements, we can conclude that we would see improvements on larger clusters as well. With larger dimension sizes and a larger cluster, the $O(N^3)$ cost of DGEMM would dominate more over the $O(N^2)$ cost of index permutation. On the other hand, the cost of array redistributions would increase wherever all-to-all communication is needed. Since in many cases our optimization was able to reduce the DGEMM cost, we would expect similar reductions to the local DGEMM computation on a larger cluster. Similarly, the ability of our algorithm to eliminate some redistributions would be of benefit on a larger cluster, and the improvement obtained by adapting the geometry of the processor grid to the geometry of the input tensors is expected to carry over to any cluster size.

7. Related work

There has been prior work that has attempted to use data layout optimizations to improve spatial locality in programs, either in addition to or instead of loop transformations. Leung and Zahorjan [49] were the first to demonstrate cases where loop transformations fail (for a variety of reasons) for which data transformations are useful. The data transformations they consider correspond to non-singular linear transformations of the data space. O'Boyle and Knijnenburg [56] present techniques for generating efficient code for several layout optimizations such as linear transformations of memory layouts, alignment of arrays to page boundaries, and page replication. Several authors [5, 29] discuss the use of data transformations to improve locality on shared memory machines. Kandemir et al. [32] present a hyperplane representation of memory layouts of multi-dimensional arrays and show how to use this representation to derive very general data transformations for a single perfectly-nested loop. In the absence of dynamic data layouts, the layout of an array has an impact on the spatial locality characteristic of all the loop nests in the program that access

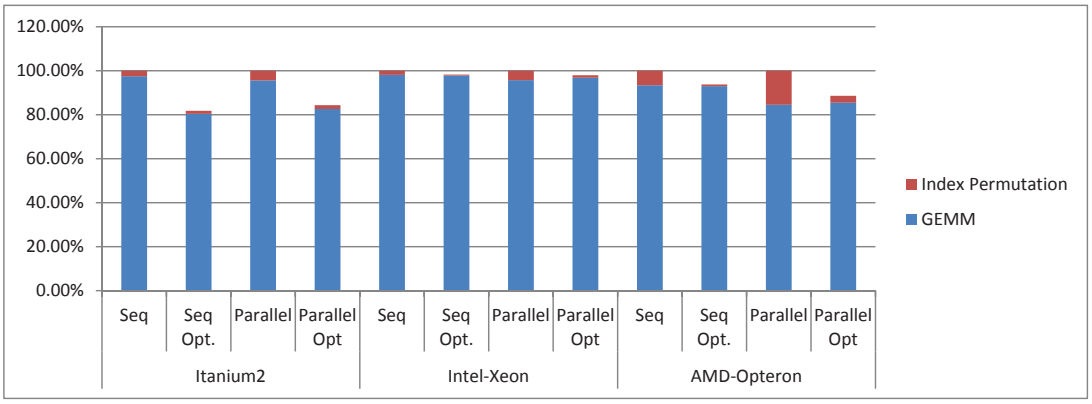


Figure 6: CCSD performance relative to unoptimized code.

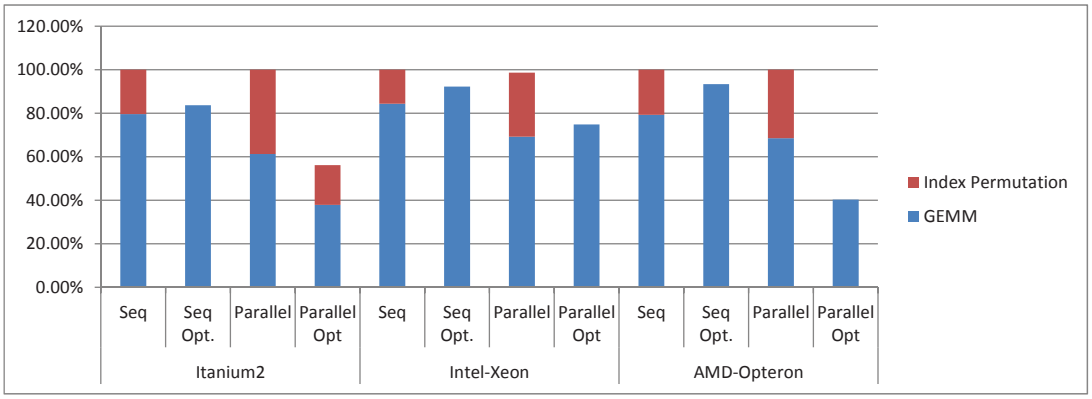


Figure 7: 4-index performance relative to unoptimized code.

the array. As a result, Kandemir et al. [30, 31, 32] and Leung and Zahorjan [49] present a global approach to this problem; of these, [30] considers dynamic layouts. Some authors have addressed unifying loop and data transformations into a single framework. These works [14, 31] use loop permutations and array dimension permutations in an exhaustive search to determine the appropriate loop and data transformations for a single nest and then extend it to handle multiple nests.

Researchers have explored the use of a performance-model driven approach in combination with empirical search [52, 12]; of these, Mitchell et al. [52] have explored the use of offline exploration. Iterative compilation [37, 36, 20] has received a lot of attention but it is time consuming. Motivated by this, Knijnenburg et al. [38, 39] have explored the use of static models in the context of caches along with empirical search to reduce the time needed for iterative compilation by as much as 50%. More recently, Yuki et al. [62] have explored the automatic creation of performance models for tile selection for use in machine learning through the use of simple program features, running synthesized kernels.

We are not aware of any work that addresses the kind of data layout optimization problem considered in this paper. Moreover, our approach is driven by empirically derived cost models and is constrained by the data layout requirements of the library calls used.

FFTW [19], PHiPAC [8] and ATLAS [59] produce high performance libraries for specific computation kernels, by executing different versions of the computation and choosing the parameters that optimize the overall execution time. Our approach is similar to these in that we perform empirical evaluation of the constituent operations for various possible parameters. However, our work focuses on a more general class of computations than a single kernel. This

forbids an exhaustive search strategy. The Sparsity system [27] also uses offline benchmarking to get parameters for a run-time tuning model, specifically for run-time data structure tuning in sparse linear algebra kernels.

As mentioned earlier, the approach presented in this paper may be viewed as an instance of the telescoping languages approach [33, 34, 11, 10]. The telescoping languages approach provides a high-level scripting interface for a computation to the user, while achieving high performance that is portable across machine architectures. It focuses on mechanisms to pre-optimize libraries and expose their performance trade-offs to allow the code generator to make effective use of the libraries.

Templates for algorithm recognition were presented by Alias et al. [2]. Building on this, Alias and Barthou [3] proposed a method that helps a user locate all fragments of code that can be replaced by library calls. Their approach does not include data layout transformations, which we address. Djoudi et al. [17] have explored the use of code (i.e., loop) specialization for different inputs through run-time switching among pre-generated versions. Khan et al. [35] have developed an approach for improving the performance of computational kernels through fast instantiations of templates done mostly at compile-time and occasionally, with negligible overhead, at run time. None of these works address layout transformations.

8. Conclusions

We have described an approach to the synthesis of efficient parallel code for tensor contractions that reduces the overall execution time. The approach has been developed for a program synthesis system targeted at the quantum chemistry domain. The code is generated as a sequence of DGEMM calls interspersed with index permutation and redistribution to enable the use of the BLAS libraries and to improve overall performance. The costs of the constituent operations in the computation were empirically measured and were used to model the cost of the computation. This computational model has been used to determine layouts and distributions that minimize the overall execution time. Experimental results on three different architectures have been provided that show the effectiveness of our optimization approach.

This approach combines the best features of empirical optimizations, namely, the incorporation of complex behavior of modern architectures, and a model-driven approach that enables efficient exploration of the search space. The regularity of the constituent operations encountered in the target application has been used to empirically measure the constituent operations. We have presented a dynamic programming solution to choose the data layouts and calls to optimized GEMM kernels that is linear in the number of tensor expressions to be optimized.

Since layout optimization applies to any GEMM or index permutation library, we plan to generalize the approach to select between different GEMM implementations. E.g., it would be straightforward to let our algorithm decide whether to use the vendor library or ATLAS for GEMM calls if for certain dimension sizes ATLAS is found to outperform the vendor library. Similarly, on a cluster, the algorithm might choose among our extended version of Cannon's algorithm, Global Arrays DGEMM [54, 24, 53], and ScaLAPACK [13]. We are currently working on a new software infrastructure for the Tensor Contraction Engine that would facilitate such experiments as well as additional measurements using larger equations on a wider variety of architectures.

We are planning to conduct experiments on larger clusters and clusters of multi-core processors that would let us better understand the trade-offs between GEMM computation time and redistribution cost in order to further optimize tensor computations on large machines. For our extension of Cannon's algorithm, we are planning to explore the relative impacts on the performance due to the shape of the processor grid, the choice of distribution indices, and redistribution. Finally, we intend to further explore the trade-offs between empirical measurements and estimation of the cost of constituent operations, so that it can be tuned by the user to achieve the level of accuracy desired.

The empirical data-layout optimization described in this paper can be viewed as an instance of telescoping languages, which understands and optimizes the library components as if they were primitive operations in the base language. We believe that this approach can be generalized and applied to other computational problems that consist of a sequence of basic operations with data layouts as optimization parameters.

Role of the funding source

This work was supported in part by the National Science Foundation under grants CHE-0121676, CHE-0121706, CNS-0509467, CCF-0541409, CCF-1059417, CCF-0073800, EIA-9986052, and EPS-1003897. The National Sci-

ence Foundation had no involvement in any aspect of the research.

References

[1] Aggregate Remote Memory Copy Interface (ARMCI) webpage. http://www.emsl.pnl.gov/docs/parsoft/armci/.

[2] C. Alias and D. Barthou. On the recognition of algorithm templates. *Electronic Notes in Theoretical Computer Science*, 82(2), 2003.

[3] C. Alias and D. Barthou. Deciding where to call performance libraries. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648 of *Lecture Notes in Computer Science*, pages 336–345. Springer, 2005.

[4] AMD Core Math Library (ACML) webpage. http://developer.amd.com/cpu/libraries/acml/.

[5] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, pages 166–178, Santa Barbara, CA, July 1995. ACM.

[6] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb. 2005.

[7] A. Bibireata, S. Krishnan, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, and V. Choppella. Memory-constrained data locality optimization for tensor contractions. In L. Rauchwerger, editor, *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 93–108, College Station, TX, Oct. 2003. Springer-Verlag.

[8] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. of ACM International Conference on Supercomputing*, pages 340–347, 1997.

[9] L. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.

[10] A. Chauhan and K. Kennedy. Reducing and vectorizing procedures for Telescoping Languages. *International Journal of Parallel Programming*, 30(4):291–315, Aug. 2002.

[11] A. Chauhan, C. McCosh, and K. Kennedy. Automatic type-driven library generation for Telescoping Languages. In *Proceedings of SC: High-performance Computing and Networking Conference*, Nov. 2003.

[12] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO'05*, 2005.

[13] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: a portable linear algebra library for distributed memory computers – design issues and performance. *Computer Physics Communications*, 97(1-2):1 – 15, 1996. High-Performance Computing in Science.

[14] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Languages Design and Implementation*, pages 205–217, La Jolla, CA, June 1995. ACM.

[15] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 177–186, Berlin, Germany, June 2002. ACM.

[16] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam. Global communication optimization for tensor contraction expressions under memory constraints. In *Proceedings of Seventeenth International Parallel and Distributed Processing Symposium (IPDPS '03)*, page 37b, Nice, France, Apr. 2003. IEEE Computer Society Press.

[17] L. Djoudi, J.-T. Acquaviva, and D. Barthou. Compositional approach applied to loop specialization. *Concurrency and Computation: Practice and Experience*, 21(1):71–84, 2009.

[18] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[19] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of 1998 IEEE International Conference on Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998. IEEE.

[20] G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg. Evaluating iterative compilation. In W. Pugh and C.-W. Tseng, editors, *LCPC*, volume 2481 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2002.

[21] X. Gao. *Integrated Compiler Optimizations for Tensor Contractions*. PhD thesis, The Ohio State University, Columbus, OH, 2008.

[22] X. Gao, S. Krishnamoorthy, S. Sahoo, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Efficient search-space pruning for integrated fusion and tiling transformations. *Concurrency and Computation: Practice and Experience*, 19(18):2425–2443, Dec. 2007.

[23] X. Gao, S. Sahoo, Q. Lu, G. Baumgartner, C. Lam, J. Ramanujam, and P. Sadayappan. Performance modeling and optimization of parallel out-of-core tensor contractions. In *Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*, pages 266–276, Chicago, Illinois, June 2005. ACM.

[24] Global Arrays webpage. http://www.emsl.pnl.gov/docs/global/.

[25] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing (2nd Edition)*. Addison-Wesley, 2003.

[26] A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. M. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry*, 113(45):12715–12723, 2009.

[27] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18:135–158, February 2004.

[28] Intel Math Kernel Library (Intel MKL) webpage. http://software.intel.com/en-us/intel-mkl/.

[29] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proceedings of the Fourth International Workshop on Language and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 344–358, Santa Clara, CA, Aug. 1991. Springer-Verlag.

[30] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. Static and dynamic locality optimizations using integer linear programming. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):922–941, 2001.

[31] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings of 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 285–296, Dallas, TX, 30 November - 2 December 1998. IEEE.

[32] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, 1999.

[33] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping Languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, Mar. 2005.

[34] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping Languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, Dec. 2001.

[35] M. A. Khan, H.-P. Charles, and D. Barthou. Improving performance of optimized kernels through fast instantiations of templates. *Concurrency and Computation: Practice and Experience*, 21(1):59–70, 2009.

[36] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 237–, Washington, DC, USA, 2000. IEEE Computer Society.

[37] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In C. D. Polychronopoulos, K. Joe, A. Fukuda, and S. Tomita, editors, *ISHPC*, volume 1615 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 1999.

[38] P. Knijnenburg, T. Kisuki, and K. Gallivan. Cache models for iterative compilation. In R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 254–261. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44681-8_37.

[39] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling: Research articles. *Concurr. Comput. : Pract. Exper.*, 16:247–270, January 2004.

[40] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, and V. Choppella. Data locality optimization for synthesis of efficient out-of-core algorithms. In *Proceedings of the the International Conference on High-Performance Computing*, volume 2913 of *Lecture Notes in Computer Science*, pages 406–417, Hyderabad, India, Dec. 2003. Springer-Verlag.

[41] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *Journal of Parallel and Distributed Computing*, 66(5):659–673, May 2006.

[42] C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, Aug. 1999. Also available as Technical Report No. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University.

[43] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage requirement for a class of loops implementing multi-dimensional integrals. In *Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, pages 350–364, San Diego, CA, Aug. 1999. Springer-Verlag.

[44] C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, July 2011.

[45] C. Lam, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

[46] C. Lam, P. Sadayappan, and R. Wenger. Optimization of a class of multi-dimensional integrals on parallel machines. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, Mar. 1997. Society for Industrial and Applied Mathematics.

[47] H.-J. Lee, J. Robertson, and J. Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 44–51, Vienna, Austria, July 1997. ACM.

[48] T. Lee and G. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. Langhoff, editor, *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pages 47–109. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[49] S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, Dept. Computer Science, University of Washington, Seattle, WA, 1995.

[50] Q. Lu. *Data Layout Optimization Techniques for Modern and Emerging Architectures*. PhD thesis, The Ohio State University, Columbus, OH, 2008.

[51] Q. Lu, S. Krishnamoorthy, and P. Sadayappan. Combining analytical and empirical approaches in tuning matrix transposition. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 233–242, New York, NY, USA, 2006. ACM.

[52] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In V. N. Alexandrov, J. Dongarra, B. A. Juliano, R. S. Renner, and C. J. K. Tan, editors, *International Conference on Computational Science (1)*, volume 2073 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2001.

[53] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[54] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–213, 2006.

[55] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda. High performance remote memory access communications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20(2):233–253, 2006.

[56] M. F. P. O'Boyle and P. M. W. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. Parallel Program.*, 27(3):131–159, 1999.

[57] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *IPDPS '09*, May

2009.
[58] R. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.
[59] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–25, 2001.
[60] R. Whaley and D. Whalley. Tuning high performance kernels through empirical compilation. In *ICPP*, pages 89–98, 2005.
[61] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. J. Garzarán, D. A. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *PLDI*, pages 63–76. ACM, 2003.
[62] T. Yuki, L. Renganarayanan, S. V. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, editors, *CGO*, pages 190–199. ACM, 2010.

Qingda Lu received the B.E. degree in 1999 from Beijing Institute of Technology, Beijing, China, his M.S. degree in 2002 from Peking University, Beijing, China and his Ph.D. degree in 2008 from The Ohio State University, all in Computer Science. He is currently working at Intel Corporation as a senior software engineer. His research interests are in optimizing compilers and operating systems.

Xiaoyang Gao received the B.S. degree in computer science from Peking University, Beijing, China, in 1997 and her Ph.D. degree from The Ohio State University in 2007. She is currently working at IBM Corporation as a staff software engineer. Her research interests are in distributed systems, compilers for high-performance computer systems, and software optimizations.

Sriram Krishnamoorthy received his B.E. degree from College of Engineering–Guindy, Anna University, Chennai, his M.S. and Ph.D. degrees from The Ohio State University, Columbus, Ohio. He is currently a research scientist at Pacific Northwest National Laboratory. His research interests include programming models for accelerators and inter-process communication, fault tolerance, and performance tools. He has received best paper awards for his publications at the International Conference on High Performance Computing (HiPC'03) and the International Parallel and Distributed Processing Symposium (IPDPS'04).

Gerald Baumgartner received the Dipl.-Ing. degree from the University of Linz, Austria, and M.S. and Ph.D. degrees from Purdue University, all in computer science. He is currently an Associate Professor at the Department of Computer Science at Louisiana State University. Previously he was at The Ohio State University. His research interests include compiler optimizations, the design and implementation of domain-specific and object-oriented languages, desktop grids, and testing tools for object-oriented and embedded systems programming.

J. (Ram) Ramanujam received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Madras, India in 1983, and his M.S. and Ph.D. degrees in Computer Science from The Ohio State University in 1987 and 1990 respectively. He is currently the John E. & Beatrice L. Ritter Distinguished Professor in the Department of Electrical and Computer Engineering at Louisiana State University. His research interests are in compilers and runtime systems for high-performance computing, domain-specific languages and compilers for parallel computing, and embedded systems. He has worked on several NSF-funded projects including the Tensor Contraction Engine and the Pluto project for automatic parallelization and locality optimization.

P. (Saday) Sadayappan received the B.Tech. degree from the Indian Institute of Technology, Madras, India, and an M.S. and a Ph.D. from the State University of New York at Stony Brook, all in Electrical Engineering. He is currently a Professor in the Department of Computer Science and Engineering at The Ohio State University. His research interests include compile-time/run-time optimization and scheduling and resource management for parallel/distributed systems.