Empirical Performance-Model Driven Data Layout Optimization*

Qingda Lu[1], Xiaoyang Gao[1], Sriram Krishnamoorthy[1], Gerald Baumgartner[2], J. Ramanujam[3], and P. Sadayappan[1]

[1] Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA
{luq,gaox,krishnsr,saday}@cse.ohio-state.edu
[2] Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803, USA
gb@csc.lsu.edu
[3] Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803, USA
jxr@ece.lsu.edu

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 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 perform data layout optimization 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 some representative computations from quantum chemistry.

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 to the computational model to minimize its cost. The large number of parameters and the variety of programs to be handled limits optimizing compilers to model-driven optimization with relatively simple cost models. Approaches to empirically optimize a computation, such as ATLAS [**?**], generate solutions for different structures of the optimized code and determine the

* Supported in part by the National Science Foundation through the Information Technology Research program (CHE-0121676 and CHE-0121706), by NSF grant CCF-0073800 and by a grant from the Environmental Protection Agency

parameters that optimize the execution time by running different versions of the code 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 empirical characterization of its constituent operations. The empirical measurements allow modeling of the overall execution time of the computation while decomposition enables offline determination of the cost model and efficient global optimization across multiple constituent operations.

Our domain of interest is the calculation of electronic structure properties using ab initio quantum chemistry models such as the coupled cluster models [**?**]. We are developing an automatic synthesis system called the Tensor Contraction Engine (TCE), to generate efficient parallel programs from high-level expressions, for a class of computations expressible as tensor contractions [**?**,**?**,**?**]. These calculations employ multi-dimensional tensors in contractions, which are essentially generalized matrix multiplications. The computation is represented by an operator tree, in which each node represents the contraction of two tensors to produce a result tensor. The order of indices of the intermediate tensors (multidimensional arrays) is not constrained.

Computational kernels such as Basic Linear Algebra Subroutines (BLAS) [**?**] have been tuned to achieve very high performance. These hand-tuned or empirically optimized kernels generally achieve better performance than conventional general-purpose compilers [**?**]. 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 described in [**?**]. 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. Parallel code is generated using the Global Arrays (GA) library [**?**]. Parallel matrix multiplication is performed using the Cannon matrix multiplication algorithm [**?**], extended to handle non-square distribution of matrices amongst the processors. 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) [**?**,**?**,**?**] 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 contraction expressions 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.

Consider the following tensor contraction expression.

$$E[i,j,k] = \text{Sum}\{a,b,c\}A[a,b,c]B[a,i]C[b,j]D[c,k]$$

where all indices range over N. a, b, c are the summation indices. The direct way to compute this would require $O(N^6)$ arithmetic operations. Instead, by computing the following intermediate partial results, the number of operations can be reduced to $O(N^4)$.

$$\begin{aligned} T1[a,b,k] &= \text{Sum}\{c\}A[a,b,c]D[c,k] \\ T2[a,j,k] &= \text{Sum}\{b\}T1[a,b,k]C[b,j] \\ E[i,j,k] &= \text{Sum}\{a\}T2[a,j,k]B[a,i] \end{aligned}$$

This form of the computation is represented as an operator tree. For example, Fig. 1(a) shows the operator tree for a sub-expression from the CCSD (Coupled Cluster Singles and Doubles) model [**?**]. The curly braces around the indices indicate the fact that there is no implied ordering between the indices. The computation represented by such an operator tree could be implemented as a collection of nested loops, one per node of the operator tree. However, optimizing the resulting collection of a large number of nested loops to minimize execution time is a difficult challenge. But each contraction is essentially a generalized matrix multiplication, for which efficient tuned library Generalized Matrix Multiplication (GEMM) routines exist. Hence it is attractive to translate the computation for each tensor contraction node into a call to GEMM. For the above 3-contraction example, the first contraction can be implemented directly as a call to GEMM with A viewed as an $N^2 \times N$ rectangular matrix and D as an $N \times N$ matrix. The second contraction, however, cannot be directly implemented as a GEMM call because the summation index b is the middle index of $T1$. GEMM requires the summation indices and non-summation indices in the contraction to be collected into two separate contiguous groups. In order to use GEMM, $T1$ needs to be "reshaped", e.g. $T1[a,b,k] \rightarrow T1r[a,k,b]$. Then GEMM can be invoked with the first operand $T1r$ viewed as an $N^2 \times N$ array and the second input operand C as an $N \times N$ array. The result, which has the index order $[a,k,j]$, would have to be reshaped to give $T2[a,j,k]$.

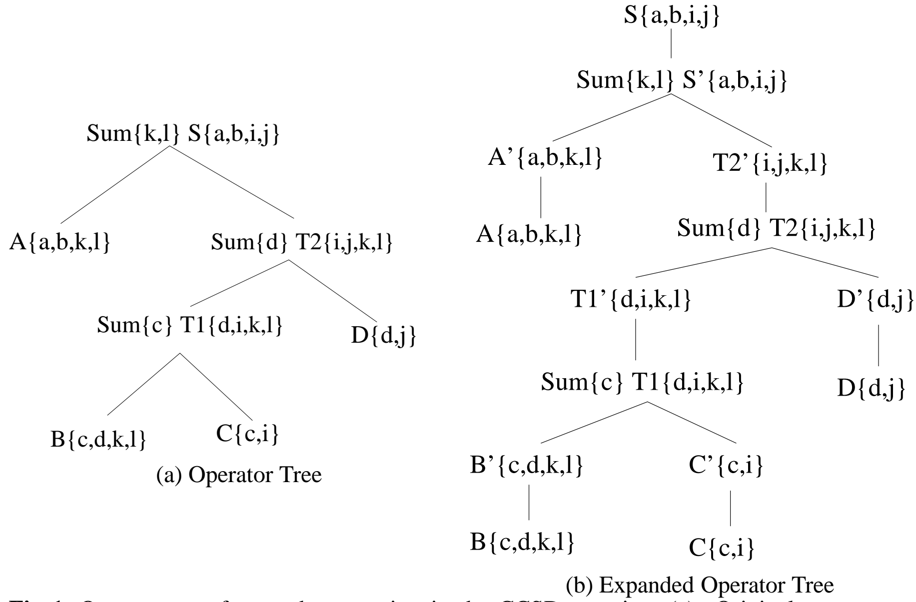


Fig. 1. Operator tree for a sub-expression in the CCSD equation. (a) Original operator tree (b) The expanded operator tree used for optimal code generation

Since $T2$ is only a temporary intermediate array, the order of its dimensions could be chosen to be $[a, k, j]$ instead of $[a, j, k]$, which avoids the need to reshape the output array produced by GEMM. Considering the last contraction, it might seem that some reshaping would be necessary in order to use GEMM. However, GEMM allows one or both of its input operands to be transposed. Thus, the contraction can be achieved by invoking GEMM with B as the first operand in transposed form, and $T2[a, j, k]$ as the second operand, with shape $N \times N^2$. But, as will be shown later, the performance of GEMM for transposed and non-transposed input operands could differ significantly.

In general, a sequence of multi-dimensional tensor contractions can be implemented using a sequence of GEMM calls, possibly with some additional array reordering operations interspersed. We represent this computation as an expanded operator tree. For example, Fig. 1(b) shows the expanded operator tree derived from the operator tree in Fig. 1(a). Each node in the operator tree is replicated to represent a possible array reordering. The problem addressed in this paper is: given a sequence of tensor contractions (expressed as an expanded operator tree), determine the layout (i.e., dimension order) and distribution (among multiple processors) of the tensors, and the modes of invocation of GEMM so that the specified computation is executed in minimal time.

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 Generalized Matrix Multiplication (GEMM)

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

$$dgemm(transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc),$$

where $transa$ ($transb$) specifies whether A (B) is in the transposed form. When $transa$ is $'n'$ or $'N'$, $op(A) = A$; when $transa$ equals to $'t'$ 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.

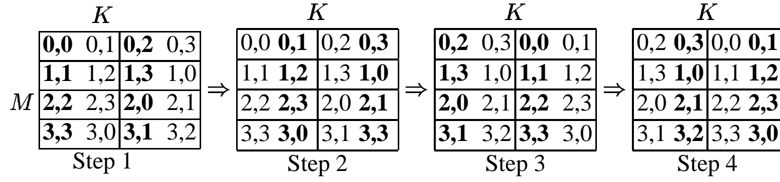
We measured the variation in the performance of GEMM with variation in its input parameters on the Itanium 2 Cluster at the Ohio Supercomputer Center (Dual 900 MHz processors with 4 GB memory, interconnected by Myrinet 2000 network). The cluster's configuration is shown in Table 1. The latency and bandwidth measurements of the interconnect were obtained from [?]. Matrix multiplications of the form $A * B$ were performed, where B was a 4000×4000 matrix and A was an $M \times 4000$ matrix, with M varied from 1 to 300. Matrix multiplications involving such oblong matrices is quite typical in quantum chemistry computations. Two BLAS libraries available in the Itanium 2 Cluster, ATLAS [?] and the Intel Math Kernel Library (MKL) [?] were evaluated. The $transb$ argument was specified as $'t'$ for the results shown in Fig. 3(a) and Fig. 4(a). Fig. 3(b) and Fig. 4(b) show the results for $transb$ being $'n'$. The x-axis shows the value of M and the y-axis represents the performance of matrix multiplication in GFLOPS. We observe that the performance of the GEMM operation for the transposed and untransposed versions cannot be interpreted as the cost of transposition at the beginning of the computation for the experiments with transposed B. For example, in some of the experiments with the ATLAS library, the transposed version performs better. Thus the parameters of the DGEMM invocations need to be determined so as to optimize the overall execution time.

Cannon's Matrix Multiplication Algorithm Several approaches have been proposed for implementing parallel matrix multiplication [?,?]. In this paper, we consider an extension to Cannon's algorithm [?], which removes the restriction of using a square grid of processors for array distribution.

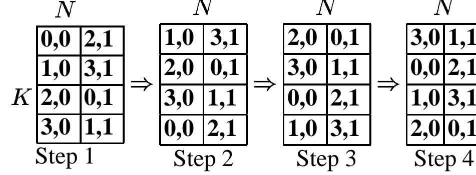
The extended Cannon algorithm for a 4×2 processors grid is illustrated for the matrix multiplication $C(M, N) \mathrel{+}= A(M, K) * B(K, N)$ in Fig. 2. The processors form

Table 1. Configuration of the Itanium 2 cluster at OSC

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



(a) Array A



(b) Array B

Fig. 2. 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

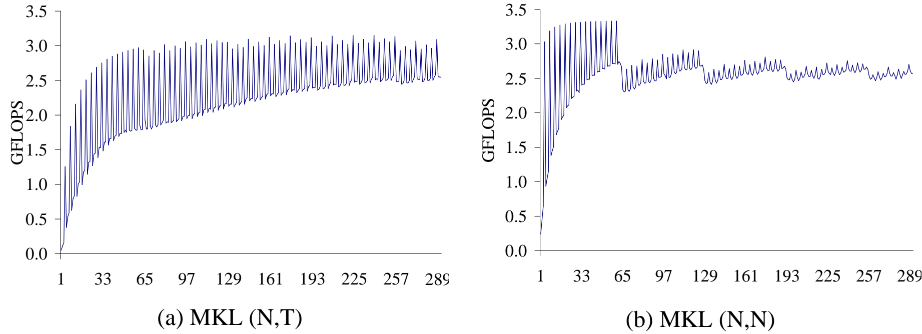
a logical rectangular grid. All the arrays are distributed amongst the processors in the gird in an identical fashion. Each processor holds a block of arrays A, B and C. The algorithm divides the common dimension (K in this illustration) to have the same number of sub-blocks. Each step operates on a 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. 2 shows in bold the sub-blocks of arrays A and B accessed in each step. It shows that the entire B array 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. 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 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.

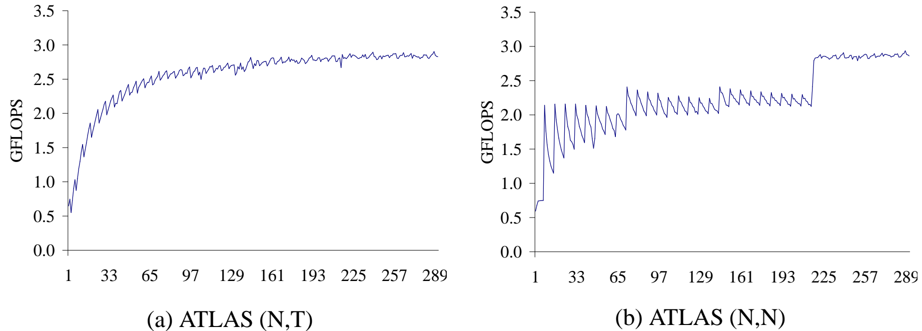
3.2 Index Permutation

DGEMM requires a two-dimensional view of the input matrices. This means that the summation and non-summation indices of a tensor contraction must be grouped into two contiguous sets of indices. Thus the layout of a multi-dimensional array might have to be transformed to be used as input to DGEMM. Further, additional index permutation



(a) MKL (N,T) (b) MKL (N,N)

Fig. 3. The matrix multiplication times using the MKL library for $C(M, N)$ += $A(M, K) * B(K, N)$ where $K = N = 4000$. M is varied along the x-axis. The performance obtained in shown on the y-axis in GFLOPS. (a) transb='t' (b) transb='n' in input argument to dgemm



(a) ATLAS (N,T) (b) ATLAS (N,N)

Fig. 4. The matrix multiplication times using the ATLAS library for $C(M, N)$ += $A(M, K) * B(K, N)$ where $K = N = 4000$. M is varied along the x-axis. The performance obtained in shown on the y-axis in GFLOPS. (a) transb='t' (b) transb='n' in input argument to dgemm

cost might be worth paying if it can reduce the DGEMM cost through the use of a transposed (or non-transposed) argument form.

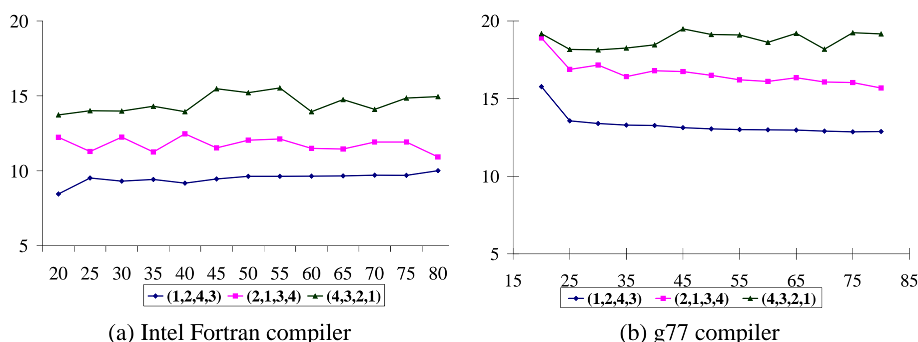
We implemented a collection of index permutation routines, one each for a different number of dimensions. The routines were tiled in the fastest varying indices in the source and target arrays. We observed that performing the computation such that the target arrays are traversed in the order of their storage resulted in better performance than biasing the access to the source array. 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.

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 results are shown in Fig. 5. Different permutations are observed to incur different costs. We also notice that the use of different compilers leads to different performances.

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's 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.

Note that the layout of input and output arrays for a DGEMM invocation uniquely determines its parameters. Thus the problem of determination of the DGEMM parameters can be reduced to the layout optimization problem. The variation in the cost of DGEMM with its parameters has the effect of increasing the search space to be explored.



(a) Intel Fortran compiler (b) g77 compiler

Fig. 5. Index permutation times 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

4 Empirical measurement of constituent operations

GEMM cost

The DGEMM cost can be determined by executing the DGEMM routine with the specified parameters on the target machine. Alternatively, the cost of DGEMM routines in a library on a particular system can be modeled by sampling it offline. The DGEMM cost for the relevant parameters can then be estimated from the sampled data set. Executing the DGEMM at runtime increases the code generation time, while estimating it leads to potential inaccuracies in the cost model.

In this paper, we determine the cost of DGEMM by executing it with the specified parameters. In the operator tree model considered, in which each non-leaf node requires exactly one invocation of the DGEMM, this could result in compilation times that are as long as the execution times. But in real quantum chemistry computations, which require out-of-core treatment [**?**], 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 node in the operator tree requires multiple invocations of DGEMM. Thus the compilation time is much less than the execution time.

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. A latency-bandwidth model is used to determine the communication cost. Consider the matrix multiplication $C(M,N) \mathrel{+}= A(M,K) * B(K,N)$. 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

$$CommnCost = CommnCost_A + CommnCost_B$$
$$CommnCost_A = (T_s + \frac{M * K}{BW * P_M * P_K}) * (P_K - P_K/P_M)$$
$$CommnCost_B = (T_s + \frac{K * N}{BW * P_K * P_N}) * (P_K - P_K/P_N)$$

where 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 in [**?**].

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 to have sufficient locality. The other basic copy cost, referred to as c_1, is one in which only the target array is traversed to have locality. Depending on the permutation and the size of the arrays, one of these basic copy costs is chosen. Note that with multi-level tiling of the routines there would be only one basic copy cost. The basic costs c_0 and c_1 were found to be compiler dependent. 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.

Constraints and Array Layouts and Distributions

The input and output arrays are constrained to have one layout each. The feasible layouts for the other nodes is given by the following equation.

$$\mathcal{S}(n) = \begin{cases} \bigcup(\forall l \in \mathcal{P}(NSI(n')))(\forall r \in \mathcal{P}(NSI(n'')))\{(l,r) \cup (r,l)\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n \text{ is contraction node} \\ \bigcup(\forall l \in \mathcal{P}(SI(n)))(\forall r \in \mathcal{P}(NSI(n)))\{(l,r) \cup (r,l)\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } n \text{ is an index permutation node} \end{cases}$$

where $\mathcal{S}(n)$ is the set of possible layouts, $SI(n)$ the set of summation indices and $NSI(n)$ the set of non-summation indices in node n. $\mathcal{P}$ is the set of all permutations of its argument. n' and n'' are the left and right child of node n, respectively. (l, r) denotes the concatenation of the sequences l and r.

A tree node C corresponding to a DGEMM computation of the form $C(M, N)$ += $A(M, K) * B(K, N)$ can have layouts corresponding to the cross-product of the permutations of the non-summation indices of its children. The remaining nodes are index permutation nodes and are constrained by the layouts acceptable by their parent (i.e., the contraction node to which they act as input). They have layouts corresponding to the cross-product of the permutations of their summation and non-summation indices.

For example, if A and B contain 3 non-summation and 2 summation indices (as determined by the C array) each, A and B have $3! * 2! * 2 = 24$ possible layouts each and C has $3! * 3! * 2 = 72$ possible layouts.

The extended Cannon algorithm requires all distributions to be rectangular in nature. In addition, the children of each contraction node in the operator tree are required to have the same distribution 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.

Determination of Optimal Parameters

For the specified layout and distribution of the root and leaves of the operator tree, we determine the layouts and distributions of the intermediate arrays. For each layout of an

array produced by DGEMM, 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 the DGEMM does not perform any index permutation within a group. This restriction is used to prune candidate layouts.

The configuration of an array is represented by a layout-distribution pair. Dynamic programming is used to determine the optimal configuration for the intermediate arrays. The cost of a node is determined as the least cost to compute its children and subsequently compute it from its children. It is as follows.

$$\mathcal{C}_t(n,d,l) = \begin{cases} \min_{\forall d' \in \mathcal{D}, \forall l' \in \mathcal{L}} \mathcal{C}_t(n',d',l') + \mathcal{C}_{ip}((n',d',l') \rightarrow (n,d,l)) \\ \qquad\qquad\qquad\qquad \text{if } n \text{ is a index permute node} \\ \min_{\forall l', l'' \in \mathcal{L}} \mathcal{C}_t(n',d,l') + \mathcal{C}_t(n'',d,l'') + \\ \quad \mathcal{C}_{dg}((n',d,l') \times (n'',d,l'') \rightarrow (n,d,l)) \\ \qquad\qquad\qquad\qquad \text{if } n \text{ is a contraction node} \end{cases}$$

where

$$\begin{aligned} \mathcal{C}_t &\equiv \text{Total cost of computing a node with relevant } (d,l) \\ \mathcal{C}_{ip} &\equiv \text{Cost of the required index permutation} \\ \mathcal{C}_{dg} &\equiv \text{Cost of the required DGEMM invocation} \\ \mathcal{D}(\mathcal{L}) &\equiv \text{All feasible distributions (layouts) of relevant node} \\ n'(n') &\equiv \text{Left(Right) child of } n \end{aligned}$$

The expanded operator tree for the example in Fig. 1(a) is shown in Fig. 1(b). The replicated nodes correspond to the index permutations. The original nodes correspond to the contractions. Thus, in the expanded operator tree, each non-leaf node is computed from its children by either index permutation or contraction. Therefore, the total cost of computation of each non-leaf, 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 operator tree. The optimal cost of the root node is subsequently computed using the dynamic programming formulation described above.

6 Experimental Results

We evaluated our approach on the OSC Itanium-2 cluster whose configuration is shown in Table 1. All the experiment programs were compiled with the Intel Itanium Fortran Compiler for Linux. We considered two computations in our domain.

1. CCSD: We used a typical sub-expression from the CCSD theory used to determine electronic structures.

$$\begin{aligned} S(j,i,b,a) = \text{Sum}\{l,k\}\, (A\{l,k,b,a\} \\ \times (\text{Sum}\{d\}\, (\text{Sum}\{c\}(B\{d,c,l,k\} \times C\{i,c\}) \times D\{j,d\})) \end{aligned}$$

All the array dimensions were 64 for the sequential experiments and 96 for the parallel experiments.

Table 2. Layouts and distributions for the CCSD computation for the unoptimized and optimized versions of the code

Array	Unoptimized				Optimized			
	Distribution	Dist. Index	Layout	GEMM. Parameters	Distribution	Dist. Index	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,i)	(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)	–

2. 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.

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

The array dimensions were 80 and 96 for the sequential and parallel experiments.

We compared our approach with the baseline implementation in which an initial layout for the arrays is provided. A fixed $\sqrt{P} \times \sqrt{P}$ array distribution is required throughout the computation. This approach was, in fact, used in our early implementations. The optimized version is allowed flexibility in the distribution of the input and output arrays.

Table 2 shows the configurations chosen for each array in the parallel experiment 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 is 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.

Table 3 and Table 4 show the sequential and parallel results respectively. In the parallel 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 totally eliminated. The trade-off between the GEMM and the index permutation costs can be observed in the sequential 4-index

Table 3. Sequential performance results for ccsd and 4index-transform

	Unoptimized (secs)				Optimized(secs)			
	GEMM	Index Permutation	Exec. Time	GFLOPS	GEMM	Index Permutation	Exec. Time	GFLOPS
ccsd	55.28	1.41	56.69	2.50	45.58	0.78	46.36	3.06
4index	10.06	2.58	12.64	2.07	10.58	0.0	10.58	2.48

Table 4. Parallel performance results on 4 processors for ccsd and 4index-transform

	Unoptimized(secs)				Optimized(secs)			
	GEMM	Index Permutation	Exec. Time	GFLOPS	GEMM	Index Permutation	Exec. Time	GFLOPS
ccsd	157.93	7.21	165.14	9.68	136.41	2.86	139.27	11.71
4index	12.23	7.74	19.97	3.27	7.57	3.64	11.21	5.83

Table 5. Per-processor costs incurred by message-passing placements

Placement	CommCost	DiskCost	Fusion Limit.
Outside	$\frac{A.size}{P} + \frac{B.size}{P}$	$\frac{A.size \times Jt}{P \times P} + \frac{B.size \times It}{P \times P} + \frac{2 \times C.size}{P} + \frac{A.size}{P} + \frac{B.size}{P}$	Only suitble for a prefectly nested loop st
Inside	$\frac{A.size \times Jt}{P \times P} + \frac{B.size \times It}{P \times P}$	$\frac{A.size \times Jt}{P \times P \times P} + \frac{B.size \times It}{P \times P \times P} + \frac{2 \times C.size}{P \times P}$	No constraint for fusion structure

	$I \geq \sqrt{MP}, J \geq \sqrt{MP}, K \geq \sqrt{MP}$	$I < \sqrt{MP}, J \geq \sqrt{MP}, K \geq \sqrt{MP}$	$I \geq \sqrt{MP}, J < \sqrt{MP}, K \geq \sqrt{MP}$	$I \geq \sqrt{M}$
ABC	$MD = \frac{A}{P} + 3\sqrt{\frac{ABC}{MP^3}}$ $MC = 2\sqrt{\frac{ABC}{MP^2}}$	Lower bound is same as (3)	Lower bound is higher than (6)	

Table 6. xxx

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.

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 [?] 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 [?] present techniques for generating efficient code for several layout optimizations such as linear transformations memory layouts, alignment of arrays to page boundaries, and page replication. Several authors [?,?] discuss the use of data transformations to improve locality on shared memory machines. Kandemir et al. [?] 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 which access the array. As a result, Kandemir et al. [?,?,?] and Leung and Zahorjan [?] present a global approach to this problem; of these, [?] considers dynamic layouts.

Some authors have addressed unifying loop and data transformations into a single framework. These works [?,?] 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.

FFTW [?] and ATLAS [?] 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. But our work focuses on a more general class of computations than a single kernel. This forbids an exhaustive search strategy.

8 Conclusions

We have described an approach to the synthesis of efficient parallel code that minimizes the overall execution time. The approach was developed for a program synthesis system targeted at the quantum chemistry domain. The code was generated as a sequence of DGEMM calls interspersed with index permutation and redistribution to enable to 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 was used to determine layouts and distributions that minimize the overall execution time. Experimental results were provided that showed the effectiveness of our approach.

In future, we intend to further explore the trade-offs between empirical measurement and estimation of the cost of constituent operations, so that it can be tuned by the user to achieve the level of accuracy desired. We also plan to evaluate our approach with other parallel matrix multiplication algorithms.