Compiler Support for Optimizing Tensor Contraction Expressions in Quantum Chemistry Computations *

Gerald Baumgartner
Dept. of Computer and Information Science
Ohio State University
gb@cis.ohio-state.edu

Daniel Cociorva
Dept. of Computer and Information Science
Ohio State University
cociorva@cis.ohio-state.edu

Chi-Chung Lam
Dept. of Computer and Information Science
Ohio State University
clam@cis.ohio-state.edu

J. Ramanujam
Dept. of Electrical and Computer Engineering
Louisiana State University
jxr@ee.lsu.edu

P. Sadayappan
Dept. of Computer and Information Science
Ohio State University
saday@cis.ohio-state.edu

Abstract

This paper provides an overview of compile-time optimizations developed in the context of a program synthesis system for a class of quantum chemistry computations. These computations are expressible as a set of tensor contractions and arise in electronic structure modeling. The synthesis system will take as input a high-level specification of the computation and generate high-performance parallel code for a number of target architectures. Several components of the synthesis system are described, focusing on compile-time optimization issues that they address.

1. Introduction

The development of high-performance parallel programs for scientific applications is often very complicated. The effect of the algorithm choice on memory access costs and communication overhead are often very complex. Currently, available tools for software development and performance optimization do not provide adequate support to the developers of high-performance scientific applications. Often, the time to develop an efficient parallel program for a computational model is the limiting factor in the rate of progress of the science. Therefore, approaches to automated synthesis of high-performance parallel programs from high-level specifications are very attractive. In general, automatic program synthesis is an intractable problem. However, for specific domains, this is feasible, as is being demonstrated by the SPIRAL project [21] for the domain of signal processing.

In this paper, we provide an overview of a project that is developing program synthesis tools to facilitate the development of high-performance parallel programs for a class of scientific computations encountered in chemistry and physics. We focus on electronic structure calculations where many computationally intensive components are expressible as a set of tensor contractions.

These computational structures arise in some computational physics codes modeling electronic properties of semiconductors and metals, and in computational chemistry codes such as ACES II, GAMESS, Gaussian, NWChem[8], PSI, and MOLPRO. In particular, they comprise the bulk of the computation with the coupled cluster approach to the accurate description of the electronic structure of atoms and molecules [16, 17]. Computational approaches to modeling the structure and interactions of molecules, the electronic and optical properties of molecules, the heats and rates of chemical reactions, etc., are crucial to the understanding of chemical processes in real-world systems. Examples of applications include combustion and atmospheric chemistry, chemical vapor deposition, protein structure and enzymatic chemistry, and industrial chemical processing. Com-

* Supported in part by the National Science Foundation through awards CHE-0121676 and EIA-9986052, and the U. S. Department of Energy through award DE-AC05-00OR22725.

putational chemistry and materials science account for significant fractions of supercomputer usage at national centers (for example, approximately 85% of total usage at Pacific Northwest National Laboratories, 30% at NERSC, and about 50% of SDSC's IBM SP/128 system).

2. The computational context

In the class of computations considered, the final result to be computed can be expressed in terms of tensor contractions, essentially a collection of multi-dimensional summations of the product of several input arrays. Due to commutativity, associativity, and distributivity, there are many different ways to compute the final result, and they could differ widely in the number of floating point operations required. Consider the following expression:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}$$

If this expression is directly translated to code (with ten nested loops, for indices $a - l$), the total number of arithmetic operations required will be $4 \times N^{10}$ if the range of each index $a - l$ is N. Instead, the same expression can be rewritten by use of associative and distributive laws:

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

This corresponds to the formula sequence shown in Fig. 1(a) and can be directly translated into code as shown in Fig. 1(b). This form only requires $6 \times N^6$ operations. However, additional space is required to store temporary arrays $T1$ and $T2$. Often, the space requirements for the temporary arrays poses a serious problem. For this example, abstracted from a quantum chemistry model, the array extents along indices $a - d$ are the largest, while the extents along indices $i - l$ are the smallest. Therefore, the size of temporary array $T1$ would dominate the total memory requirement.

The operation minimization problem encountered here is a generalization of the well known matrix-chain multiplication problem, where a linear chain of matrices to be multiplied is given, e.g., ABCD, and the optimal order of pair-wise multiplications is sought, i.e., ((AB)C)D versus (AB)(CD), etc. In contrast, for computations expressed as sets of matrix contractions, there is additional freedom in choosing the pair-wise products. For the above example, instead of forcing a single chain order, e.g., ABCD, other orders are possible, such as the BDCA order shown for the operation-reduced form above.

We have shown that the problem of determining the operator tree with minimal operation count is NP-complete, and have developed a pruning search procedure [14, 15] that is very efficient in practice. For the above example, although the latter form is far more economical in terms of the number of operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions. Sometimes, the sizes of intermediate arrays needed for the "operation-minimal" form are too large to even fit on disk.

A systematic way to explore ways of reducing the memory requirement for the computation is to view it in terms of potential loop fusions. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus reduces the memory requirements. For the example considered, the application of fusion is illustrated in Fig. 1(c). This way, $T1$ can be reduced to a scalar and $T2$ to a 2-dimensional array, without changing the number of operations.

For a computation comprising a number of nested loops, there are a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. In prior work, we addressed the problem of finding the choice of fusions for a given operator tree that minimized the total space required for all arrays [11, 12, 13].

3. An example

One of the most computationally intensive components of many quantum chemistry packages is the CCSD(T) scheme. It is a coupled cluster approximation that includes single and double excitations from the Hartree-Fock wavefunction plus a perturbative estimate for the *connected* triple excitations. For molecules well described by a Hartree-Fock wave function, this method predicts bond energies, ionization potentials, and electron affinities to an accuracy of ± 0.5 kcal/mol, bond lengths accurate to ± 0.0005 Å, and vibrational frequencies accurate to $\pm 5 cm^{-1}$. This level of accuracy is adequate to answer many of the questions that arise in studies of chemical systems.

As a motivating example for the problem addressed, we discuss a component of the CCSD(T) calculation. The following representative equation arises in the Laplace factorized expression for linear triples perturbation correction:

$$\begin{aligned} A3A &= X_{ce,af} Y_{ae,cf} + X_{a\bar{e},c\bar{f}} Y_{c\bar{e},a\bar{f}} + X_{a\bar{e},\bar{c}f} Y_{\bar{c}\bar{e},af} \\ &+ X_{\bar{a}e,c\bar{f}} Y_{ce,\bar{a}\bar{f}} + X_{\bar{a}e,\bar{c}f} Y_{\bar{c}e,\bar{a}f} + X_{\bar{a}\bar{e},\bar{c}\bar{f}} Y_{\bar{c}\bar{e},\bar{a}\bar{f}}, \end{aligned}$$

where X and Y are of the form $X_{ae,cf} = t_{ij}^{ae} t_{ij}^{cf}$ and $Y_{ce,af} = \langle cb \parallel ek \rangle \langle ab \parallel fk \rangle$, respectively.

Integrals with two vertical bars have been antisymmetrized and may be expressed as: $(\langle pq \parallel rs \rangle = \langle pq \mid rs \rangle - \langle pq \mid sr \rangle)$, where integrals with one vertical bar are of the form $\langle \mu\nu \mid \omega\lambda \rangle = \int\int dr^3 ds^3 \phi_\mu(\underline{\mathrm{r}}) \phi_\nu(\underline{\mathrm{s}}) |\underline{\mathrm{r}} -$

$$
\begin{aligned}
T1_{bcdf} &= \sum_{el} B_{befl} \times D_{cdel} \\
T2_{bcjk} &= \sum_{df} T1_{bcdf} \times C_{dfjk} \\
S_{abij} &= \sum_{ck} T2_{bcjk} \times A_{acik}
\end{aligned}
$$

(a) Formula sequence

```
T1=0; T2=0; S=0;
for b, c, d, e, f, l
[ T1bcdf += Bbefl Dcdel
for b, c, d, f, j, k
[ T2bcjk += T1bcdf Cdfjk
for a, b, c, i, j, k
[ Sabij += T2bcjk Aacik
```

(b) Direct implementation (unfused code)

```
S = 0;
for b, c
  T1f = 0; T2f = 0;
  for d, f
    for e, l
      [ T1f += Befl Dcdel
    for j, k
      [ T2fjk += T1f Cdfjk
  for a, i, j, k
    [ Sabij += T2fjk Aacik
```

(c) Memory-reduced implementation (fused)

Figure 1. Example illustrating use of loop fusion for memory reduction.

$\underline{\mathrm{s}}|^{-1}\phi_\omega(\underline{\mathrm{r}})\phi_\lambda(\underline{\mathrm{s}})$ and are quite expensive to compute (requiring on the order of 1000 arithmetic operations). Electrons may have either up or down (or alpha/beta) spin. Down spin is denoted here with an over-bar. The indices i, j, k, l, m, n refer to occupied orbitals, of number O between 30 and 100. The indices a, b, c, d, e, f refer to unoccupied orbitals of number V between 1000 and 3000. The integrals are written in the molecular orbital basis, but must be computed in the underlying atom-centered Gaussian basis, and transformed to the molecular orbital basis. We omit these details in our discussion here.

A3A is one of many contributions to the energy, and among the most expensive, scaling as $O(OV^5)$. Here, we assume that we have already computed the amplitudes t_{ij}^{ae}, and they must be read as necessary, and contracted to form a block of X. The integrals $\langle cb \parallel ek\rangle$ must be recomputed as necessary, contracted to form a block of Y corresponding to X, and the two contracted to form the scalar contribution to the energy.

Fig. 2 shows pseudo-code for the computation of one of the energy components E for $A3A$. Temporary arrays $T1$ and $T2$ are used to store the integrals of form $\langle ab \parallel ek\rangle$, where the functions $f1$ and $f2$ represent the integral calculations. The intermediate quantities X_{aecf} are computed by contracting over (i.e., summing over products of) input array T, while the intermediate quantities Y_{ceaf} are obtained by contracting over $T1$ and $T2$. The final result is a single scalar quantity E, that is obtained by adding together the $O(OV^3)$ pair-wise products $X_{aecf}Y_{ceaf}$.

The cost of computing each integral $f1$, $f2$ is represented by C_i, and in practice is of the order of hundreds or a few thousand arithmetic operations. The pseudo-code form shown in Fig. 2 is computationally very efficient in minimizing the number of expensive integral function evaluations $f1$ and $f2$, and maximizing the reuse of the stored integrals in $T1$ and $T2$ (each element of $T1$ and $T2$ is used $O(V^2)$ times). However, it is impractical due to the huge memory requirement. With $O = 100$ and $V = 5000$, the size of $T1$, $T2$ is $O(10^{14})$ bytes and the size of X, Y is $O(10^{15})$ bytes. By fusing together pairs of producer-consumer loops in the computation, reductions in the needed array sizes may be sought, since the fusion of a loop with common index in the pair of loops allows the elimination of that dimension of the intermediate array. It can be seen that the loop that produces X (with indices a, e, c, f), the loop that produces Y (with indices c, e, a, f) and the loop that consumes X and Y to produce E (with indices c, e, a, f) can all be fully fused together, permitting the elimination of all explicit indices in X and Y to reduce them to scalars. However, the loops producing $T1$ (with indices c, e, b, k) and $T2$ (with indices a, f, b, k) cannot also be directly fused with the other three loops because their indices do not match.

Fig. 3 shows how a reduction of space for $T1$ and $T2$ can be achieved by introducing redundant loops around their producer loops — add loops with the missing indices a, f for $T1$ and c, e for $T2$. Now all five loops have common indices a, e, c, f that can be fused, permitting elimination of those indices from all temporaries. Further, by fusing the producer loops for $T1$ and $T2$ with their consumer loop, which produces Y, the b, k indices can also be eliminated from $T1$ and $T2$. A dramatic reduction of memory space is achieved, reducing all temporaries $T1, T2, X$ and Y to scalars, but the space savings come at the price of a significant increase in computation. No reuse is achieved of the quantities derived from the expensive integral calculations $f1$ and $f2$. Since C_i is of the order of 1000 in practice, the integral calculations now dominate the total compute time, increasing the operation count by three orders of magnitude over the unfused form in Fig. 2.

A desirable solution would be somewhere in between the unfused structure of Fig. 2 (with maximal memory requirement and maximal reuse) and the fully fused structure of Fig. 3 (with minimal memory requirement and minimal reuse). This is shown in Fig. 4, where tiling and partial fusion of the loops is employed. The loops with indices a, e, c, f are tiled by splitting each of those indices into a pair of indices. The indices with a superscript t represent the tiling loops and the unsuperscripted indices now stand for intra-tile loops with a range of B, the block size used for tiling. For each tile (a^t, e^t, c^t, f^t), blocks of $T1$ and $T2$ of size B^2 are computed and used to form B^4 product con-

```
for a, e, c, f
  for i, j
    X_aecf += T_ijae T_ijcf
for c, e, b, k
  T1_cebk = f1(c, e, b, k)
for a, f, b, k
  T2_afbk = f2(a, f, b, k)
for c, e, a, f
  for b, k
    Y_ceaf += T1_cebk T2_afbk
for c, e, a, f
  E += X_aecf Y_ceaf
```

array	space	time
X	V^4	V^4O^2
T1	V^3O	C_iV^3O
T2	V^3O	C_iV^3O
Y	V^4	V^5O
E	1	V^4

Figure 2. Unfused operation-minimal form.

```
for a, e, c, f
  for i, j
    X_aecf += T_ijae T_ijcf
for a, f
  for c, e, b, k
    T1_cebk = f1(c, e, b, k)
for c, e
  for a, f, b, k
    T2_afbk = f2(a, f, b, k)
for c, e, a, f
  for b, k
    Y_ceaf += T1_cebk T2_afbk
for c, e, a, f
  E += X_aecf Y_ceaf
```

⇒

```
for a, e, c, f
  for i, j
    X += T_ijae T_ijcf
  for b, k
    T1 = f1(c, e, b, k)
    T2 = f2(a, f, b, k)
    Y += T1 T2
  E += X Y
```

array	space	time
X	1	V^4O^2
T1	1	C_iV^5O
T2	1	C_iV^5O
Y	1	V^5O
E	1	V^4

Figure 3. Use of redundant computation to allow full fusion.

```
for a^t, e^t, c^t, f^t
  for a, e, c, f
    for i, j
      X_aecf += T_ijae T_ijcf
  for b, k
    for c, e
      T1_ce = f1(c, e, b, k)
    for a, f
      T2_af = f2(a, f, b, k)
    for c, e, a, f
      Y_ceaf += T1_ce T2_af
  for c, e, a, f
    E += X_aecf Y_ceaf
```

array	space	time
X	B^4	V^4O^2
T1	B^2	$C_i(V/B)^2V^3O$
T2	B^2	$C_i(V/B)^2V^3O$
Y	B^4	V^5O
E	1	V^4

Figure 4. Use of tiling and partial fusion to reduce recomputation cost.

tributions to the components of Y, which are stored in an array of size B^4.

As the tile size B is increased, the cost of function computation for $f1, f2$ decreases by a factor of B^2, due to the reuse enabled. However, the size of the needed temporary array for Y increases as B^4 (the space needed for X can be reduced back to a scalar by fusing its producer loop with the loop producing E, but Y's space requirement cannot be decreased). When B^4 becomes larger than the size of physical memory, expensive paging in and out of disk will be required for Y. Further, there is diminishing returns on reuse of $T1$ and $T2$ after B^2 becomes comparable to C_i, since the loop producing Y now becomes the dominant one. So we can expect that as B is increased, performance will improve and then level off and then deteriorate. The optimum value of B will clearly depend on the cost of access at the various levels of the memory hierarchy.

The computation considered here is just one component of the $A3A$ term, which in turn is only one of very many terms that must be computed. Although developers of quantum chemistry codes naturally recognize and perform some of these optimizations, a collective analysis of all these computations to determine their optimal implementation is beyond the scope of manual effort. Further, the time required to develop codes to implement such computational models is quite large, especially since the tensor expressions can get quite complex - Fig. 5 shows an example of the kind of tensor expressions encountered when developing accurate computational models.

In the next section, we provide an overview of the compile-time optimizations that are being addressed in the context of the synthesis system that we are developing to aid quantum chemists in rapidly developing high-performance parallel codes for computations that they specify as a set of high-level tensor contractions.

4. Compile-time optimizations

We present in this section a brief description of the basic components of the system being developed, and the compile-time optimization issues addressed within each component. Some of these components are tightly coupled (for example, memory minimization and data distribution), and they are treated together as one combined module in the synthesis system.

Algebraic transformations: It takes input from the user as tensor expressions and synthesizes an output computation sequence. The Algebraic Transformations module uses the properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition. It searches for all possible ways of applying these properties to an input sum-of-products expression, and determines a combination that results in an equivalent form of the computation with minimal operation cost.

Memory minimization: The operation-minimal computation sequence synthesized by the Algebraic Transformation module might require an excessive amount of memory due to the large temporary intermediate arrays involved. The Memory Minimization module attempts to perform loop fusion transformations to reduce the memory requirements. This is done without any change to the number of arithmetic operations.

Space-time transformation: If the memory minimization module is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system, the computation is infeasible unless a space-time trade-off is performed. If no satisfactory transformation is found, feedback is provided to the Memory Minimization module, causing it to seek a different solution. If the Space-Time Transformation module is successful in bringing down the memory requirement below the disk capacity, the Data Locality Optimization module is invoked.

Data locality optimization: If the space requirement exceeds physical memory capacity, portions of the arrays must be moved between disk and main memory as needed, in a way that maximizes reuse of elements in memory. The same considerations are involved in minimizing cache misses — blocks of data are moved between physical memory and the space available in the cache.

Data distribution and partitioning: This component determines how best to partition the arrays among the processors of a parallel system. We assume a data-parallel model, where each operation in the operation sequence is distributed across the entire parallel machine. The arrays are to be disjointly partitioned between the physical memories of the processors. Since the data distribution pattern affects the memory usage on the parallel machine, this component is closely coupled with the memory minimization component.

Code generation: The back end of the synthesis system provides the output as pseudocode, Fortran or C code. The generated code can be either serial or parallel, using the MPI libraries. Depending on the circumstances, the synthesized code could also call highly-tuned, machine-specific Basic Linear Algebra Subprograms (BLAS) libraries, or optimized low-level functions from the existing quantum chemistry packages.

In the next sections, we provide some details about the optimizations implemented in some of these modules. For details of the operation minimization algorithm see [14, 15].

5. Memory minimization and space-time trade-offs

As discussed in Section 2, the operation minimization procedure results in the creation of intermediate temporary arrays. For typical computations in computational chemistry, the space required for storing these temporary arrays

```
hbar[a,b,i,j] == sum[f[b,c] * t[i,j,a,c], c] - sum[f[k,c] * t[k,b] * t[i,j,a,c], k,c] + sum[f[a,c] * t[i,j,c,b], c] - sum[f[k,c] * t[k,a] * t[i,j,c,b], k,c] - sum[f[k,j] * t[i,k,a,b], k] - sum[f[k,c] * t[j,c] * t[i,k,a,b], k,c] - sum[f[k,i] * t[j,k,b,a], k] - sum[f[k,c] * t[i,c] * t[j,k,b,a], k,c] + sum[t[i,c] * t[j,d] * v[a,b,c,d], c,d] + sum[t[i,j,c,d] * v[a,b,c,d], c,d] + sum[t[j,c] * v[a,b,i,c], c] - sum[t[k,b] * v[a,k,i,j], k] + sum[t[i,c] * v[b,a,j,c], c] - sum[t[k,a] * v[b,k,j,i], k] - sum[t[k,d] * t[i,j,c,b] * v[k,a,c,d], k,c,d] - sum[t[i,c] * t[j,k,b,d] * v[k,a,c,d], k,c,d] - sum[t[j,c] * t[k,b] * v[k,a,c,i], k,c] + 2 * sum[t[j,k,b,c] * v[k,a,c,i], k,c] - sum[t[j,k,c,b] * v[k,a,c,i], k,c] - sum[t[i,c] * t[j,d] * t[k,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[k,d] * t[i,j,c,b] * v[k,a,d,c], k,c,d] - sum[t[k,b] * t[i,j,c,d] * v[k,a,d,c], k,c,d] - sum[t[j,d] * t[i,k,c,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[i,c] * t[j,k,b,d] * v[k,a,d,c], k,c,d] - sum[t[i,c] * t[j,k,d,b] * v[k,a,d,c], k,c,d] - sum[t[j,k,b,c] * v[k,a,i,c], k,c] - sum[t[i,c] * t[k,b] * v[k,a,j,c], k,c] - sum[t[i,k,c,b] * v[k,a,j,c], k,c] - sum[t[i,c] * t[j,d] * t[k,a] * v[k,b,c,d], k,c,d] - sum[t[k,d] * t[i,j,a,c] * v[k,b,c,d], k,c,d] - sum[t[k,a] * t[i,j,c,d] * v[k,b,c,d], k,c,d] + 2 * sum[t[j,d] * t[i,k,a,c] * v[k,b,c,d], k,c,d] - sum[t[j,d] * t[i,k,c,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[j,k,d,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[k,a] * v[k,b,c,j], k,c] + 2 * sum[t[i,k,a,c] * v[k,b,c,j], k,c] - sum[t[i,k,c,a] * v[k,b,c,j], k,c] + 2 * sum[t[k,d] * t[i,j,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,d] * t[i,k,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,c] * t[k,a] * v[k,b,i,c], k,c] - sum[t[j,k,c,a] * v[k,b,i,c], k,c] - sum[t[i,k,a,c] * v[k,b,j,c], k,c] + sum[t[i,c] * t[j,d] * t[k,a] * t[l,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[k,l,c,d], k,l,c,d] + sum[t[k,a] * t[l,b] * t[i,j,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,c,d], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,c,a] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,b] * t[j,k,d,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,c,d], k,l,c,d] + 4 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,a] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[j,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,j,c,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,c,b] * t[k,l,a,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,a,c] * t[k,l,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[j,c] * t[k,b] * t[l,a] * v[k,l,c,i], k,l,c] + sum[t[l,c] * t[j,k,b,a] * v[k,l,c,i], k,l,c] - 2 * sum[t[l,a] * t[j,k,b,c] * v[k,l,c,i], k,l,c] + sum[t[l,a] * t[j,k,c,b] * v[k,l,c,i], k,l,c] - 2 * sum[t[k,c] * t[j,l,b,a] * v[k,l,c,i], k,l,c] + sum[t[k,a] * t[j,l,b,c] * v[k,l,c,i], k,l,c] + sum[t[k,b] * t[j,l,c,a] * v[k,l,c,i], k,l,c] + sum[t[j,c] * t[l,k,a,b] * v[k,l,c,i], k,l,c] + sum[t[i,c] * t[k,a] * t[l,b] * v[k,l,c,j], k,l,c] + sum[t[l,c] * t[i,k,a,b] * v[k,l,c,j], k,l,c] - 2 * sum[t[l,b] * t[i,k,a,c] * v[k,l,c,j], k,l,c] + sum[t[l,b] * t[i,k,c,a] * v[k,l,c,j], k,l,c] + sum[t[i,c] * t[k,l,a,b] * v[k,l,c,j], k,l,c] + sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,a] * t[i,k,c,b] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,b] * t[j,l,d,a] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,d,c], k,l,c,d] + sum[t[k,a] * t[l,b] * v[k,l,i,j], k,l] + sum[t[k,l,a,b] * v[k,l,i,j], k,l] + sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[l,k,c,d], k,l,c,d] + sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,d,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,c,b] * t[k,l,a,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,a,c] * t[k,l,b,d] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[l,c] * t[i,k,a,b] * v[l,k,c,j], k,l,c] + sum[t[l,b] * t[i,k,a,c] * v[l,k,c,j], k,l,c] + sum[t[l,a] * t[i,k,c,b] * v[l,k,c,j], k,l,c] + v[a,b,i,j]
```

Figure 5. A contraction expression from quantum chemistry.

can be several tera bytes, which makes the computation impractical. As shown in in Fig. 1(c), the problem with memory requirements of large intermediate arrays can be mitigated through loop fusion. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array that is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus lowers the memory requirement. The use of loop fusion can be seen to result in significant potential reduction to the total memory requirement. For a computation comprising of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. A data structure that we call a *fusion graph* can be used to facilitate enumeration of all possible compatible fusion configurations for a given computation tree.

Fig. 6 shows the fusion graph for the unfused form of the computation from Fig. 2. Corresponding to each node in a computation tree, the fusion graph has a set of vertices corresponding to the loop indices of the node of the computation tree. In Fig. 6, we do not show the operator tree corresponding to the computation, but directly illustrate the fusion graph. The potential for fusion of a common loop among a producer-consumer pair of loop nests is indicated in the fusion graph through a dashed *potential fusion* edge connecting the corresponding vertices. Leaf nodes in the fusion graph correspond to input arrays or primitive function evaluations and do not represent a loop nest. The edges from the leaves to their parents are shown as dotted edges and do not affect the fusion possibilities. If a pair of loop nests is fused using one or more common loops, it is captured in the fusion graph by changing the dashed potential-fusion edges to continuous fusion edges. If more than two loop nests are fused together, a chain of fusion edges results,

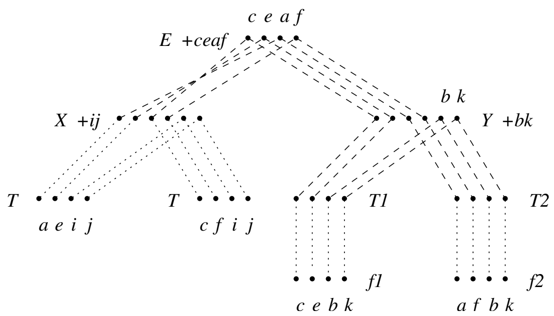


Figure 6. Fusion graph for unfused operation-minimal form of loop in Fig. 2

called a *fusion chain*. The *scope of a fusion chain* is the set of nodes it spans. The fusion graph allows us to characterize the condition for feasibility of a particular combination of fusions: the scope of any two fusion chains in a fusion graph must either be disjoint or a subset/superset of each other. Scopes of fusion chains do not partially overlap because loops do not (i.e., loops must be either separate or nested).

The fusion graph in Fig. 6 can be used to determine the fusion possibilities. On the left side of the graph, the edges corresponding to (a, e, c, f) can all be made fusion edges, suggesting that complete fusion is possible for the loop nests producing and consuming X, reducing it to a scalar. Similarly, on the right side of the graph, the edges corresponding to (c, e, a, f) can also be made fusion edges, reducing Y to a scalar. Further, by creating fusion edges for indices (c, e), the producer loop for $T1$ can be fully fused with the Y loop that consumes it. However, now the producer loop for $T2$ cannot be fused since the addition of any fusion edge (say for index a) will result in partially overlapping fusion chains for a and (c, e). The fully fused version from Fig. 3 can be represented graphically as shown in Fig. 7(a). Additional vertices have been added for indices (c, e) and (a, f), respectively, at the nodes corresponding to the producer loops for $T1$ and $T2$. Now, complete fusion chains can be created without any partial overlap in the scopes of the fusion chains. From the figure, it can be seen that in fact the redundant computation need only be added to one of $T1$ or $T2$ to achieve complete fusion. For example, removing the additional vertices for (a, f) at $T2$ does not violate the non-partial-overlap condition for fusion.

The fusion graph was used to develop an algorithm [13, 11] to determine the combination of fusions that minimizes the total storage required for all the temporary intermediate arrays. A bottom-up dynamic programming approach was used that maintains a set of pareto-optimal fusion configurations at each node, merging solutions from children nodes to generate the optimal configurations at a parent. The two metrics used are the total memory required under the subtree rooted at the node, and the constraints imposed by a configuration on fusion further up the tree. A configuration is inferior to another if it is *more or equally constraining* with respect to further fusions than the other, and uses no less memory. At the root of the tree, the configuration with the lowest memory requirement is chosen.

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

If after memory minimization the storage requirements still exceed the disk capacity, we need to recompute some (parts of) temporary arrays in order to further reduce the space requirements. The space-time trade-off algorithm we have developed [4] employs a combination of fusion and tiling to achieve a good balance between recomputation and memory usage. The first step of the space-time trade-off algorithm uses a dynamic programming approach similar to the memory minimization algorithm that maintains a set of pareto-optimal fusion/recomputation configurations, in which the recomputation cost is used as a third metric. Solutions exceeding the memory limit are pruned out. The result of the search is a set of loop structures with different combinations of space requirements and recomputation cost.

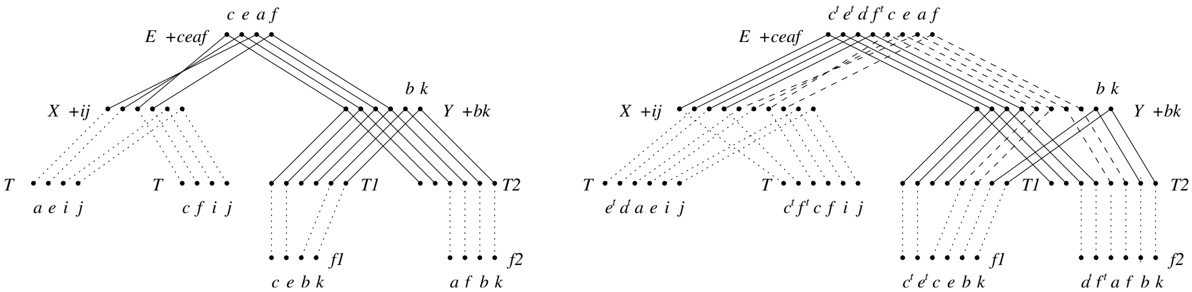
In the second step of the algorithm, recomputation indices are split into tiling and intra-tile loop pairs. By making intra-tile loops the inner-most loops, any recomputation only needs to be performed once per iteration of the tiling loop in exchange for increasing the storage requirements for temporaries in which the dimension corresponding to the tiled loop had been eliminated. For each solution from the first step of the algorithm, we then search for tile sizes that minimize the recomputation cost, and take the solution that results in the lowest recomputation cost.

6. Data locality optimization

Once a solution is found that fits onto disk, we optimize the data locality to reduce memory and disk access times. We developed algorithms [2, 3] that, given a memory-reduced (fused) version of the code, find the appropriate blocking of the loops in order to maximize data reuse. These algorithms can be applied at different levels of the memory hierarchy, for example, to minimize data transfer between main memory and disk (disk access minimization), or to minimize data transfer between main memory and the cache (cache optimization). In this section, we briefly describe the main points of our algorithm [3], focusing mostly on the cache management problem. For the disk access minimization problem, the same approach is used, replacing the cache size by the physical memory size.

We introduce a memory access cost model (*Cost*), an estimate on the number of cache misses, as a function of tile sizes and loop bounds. In a bottom-up traversal of the abstract syntax tree, we count for each loop the number (*Accesses*) of distinct array elements accessed in its scope. If this number is smaller than the number of elements that fit into the cache, then *Cost* = *Accesses*. Otherwise, it means that the elements in the cache are not reused from one loop iteration to the next, and the cost is obtained by multiplying the loop range by the cost of its inner loop(s).

Using this cost model, we can compute the total memory access cost for given tile sizes. The procedure is repeated for different sets of tile sizes, and new costs are computed. In the end the lowest possible cost is chosen, thus determining the optimal tile sizes. We define our tile size search



(a) Fully fused computation from Fig. 3.

(b) Partially fused computation from Fig. 4.

Figure 7. Fusion graphs showing redundant compution and tiling.

space in the following way: if N_i is a loop range, we use a tile size starting from $T_i = 1$ (no tiling), and succesively increasing T_i by doubling it until it reaches N_i. This approach ensures a slow (logarithmic) growth of the search space with increasing array dimension for large N_i. If N_i is small enough, an exhaustive search is performed instead.

7. Data partitioning and communication minimization

Given an operation-optimal operator tree, there are many choices for the parallel implementation of the operations. We have developed an algorithm [5] to determine the optimal data distribution that minimizes the total amount of inter-processor communication, subject to the available memory constraints of the parallel machine. We present in this section the fundamental aspects of the algorithm.

Since each node in the tree typically involves operations on large arrays, there is no need to operate on several tree nodes at the same time, i.e. intra-node parallelism is sufficient. We use a logical view of the processors as a multi-dimensional grid, where each array can be distributed or replicated along one or more of the processor dimensions. Let p_d be the number of processors on the d-th dimension of an n-dimensional processor array, so that the total number of processors is $p_1 \times p_2 \times \ldots \times p_n$. We use an n-tuple to denote the partitioning or *distribution* of the elements of a data array on an n-dimensional processor array. The d-th position in an n-tuple α, denoted $\alpha[d]$, corresponds to the d-th processor dimension. Each position may be one of the following: an index variable distributed along that processor dimension, a '*' denoting replication of data along that processor dimension, or a '1' denoting that only the first processor along that processor dimension is assigned any data. If an index variable appears as an array subscript but not in the n-tuple, then the corresponding dimension of the array is not distributed. Conversely, if an index variable appears in the n-tuple but not in the array, then the data is replicated along the corresponding processor dimension, which is the same as replacing that index variable with a '*'.

As an example, consider the 3-dimensional array $B[j,k,t]$, and suppose 64 processors form a $2 \times 4 \times 8$ array. The 3-tuple $\langle k,*,1\rangle$ specifies that the second dimension of B is distributed along the first processor dimensions, the first and third dimensions of B are not distributed, and that data are replicated along the second processor dimension and are assigned only to processors whose third processor dimension equals 1. Thus, a processor whose id is P_{z_1,z_2,z_3} will be assigned a portion of B specified by $B[1:N_j, \mathit{myrange}(z_1,N_k,p_1), 1:N_t)]$ if $z_3 = 1$ and no part of B otherwise, where $\mathit{myrange}(z,N,p)$ is the range $(z-1)\times N/p+1$ to $z \times N/p$.

A child array is redistributed before the evaluation of its parent if their distributions do not match. For instance, suppose the arrays $T_1[j,t]$ and $T_2[j,t]$ in the computation $T_3[j,t] = T_1[j,t] \times T_2[j,t]$ have distributions $\langle 1,t,j\rangle$ and $\langle j,*,1\rangle$, respectively. If we want T_3 to have distribution $\langle j,t,1\rangle$, T_1 would have to be redistributed from $\langle 1,t,j\rangle$ to $\langle j,t,1\rangle$ because the two distributions do not match. But for T_2 to go from $\langle j,*,1\rangle$ to $\langle j,t,1\rangle$, each processor just needs to give up part of the t-dimension of the array and no inter-processor data movement is required.

The goal is to determine the combination of the distribution n-tuples for the data arrays and the loop nests that minimizes the computational and communication costs. A dynamic programming algorithm that determines the distribution of dense arrays to minimize the computational and communication costs is given below.

1. Let T be an expression tree, and $\mathit{Cost}(v,\alpha)$ the minimal cost for the subtree rooted at v with distribution α. Initialize $\mathit{Cost}(v,\alpha)$ for each leaf node v in T and each distribution α as follows: $\mathit{Cost}(v,\alpha) = 0$ if $\mathit{NoReplicate}(\alpha)$, where $\mathit{NoReplicate}(\alpha)$ is a predicate meaning α involves no replication, and $\mathit{Cost}(v,\alpha) = \min_{\mathit{NoReplicate}(\beta)}\{\mathit{MoveCost}(v,\beta,\alpha)\}$ otherwise.

2. Perform a bottom-up traversal of T. For each internal node u and each distribution α, calculate $\mathit{Cost}(u,\alpha)$ as follows:
   Case (a): u is a multiplication node with two children

v and v'. We need both v and v' to have the same distribution, say γ, before u can be formed. After the multiplication, the product could be redistributed if necessary. Thus,

$$\begin{aligned} \mathit{Cost}(u,\alpha) &= \min_{\gamma}\{\mathit{Cost}(v,\gamma)+\mathit{Cost}(v',\gamma) \\ &+ \mathit{CalcCost}(u,\gamma)+\mathit{MoveCost}(u,\gamma,\alpha)\} \end{aligned}$$

Case (b): u is a summation node over index i and with a child v. v may have any distribution γ. If $i \in \gamma$, each processor first forms partial sums of u and then we either combine the partial sums on one processor along the i dimension or replicate them on all processors along that processor dimension. Afterwards, the sum could be redistributed if necessary. Thus,

$$\begin{aligned} \mathit{Cost}(u,\alpha) &= \min_{\gamma}\{\mathit{Cost}(v,\gamma)+\min_{i=1,2}(\mathit{CalcCost}_i(u,\gamma) \\ &+ \mathit{MoveCost}_i(u,\gamma,\alpha))\} \end{aligned}$$

In either case, save into $\mathit{Dist}(u,\alpha)$ the distribution γ that minimizes $\mathit{Cost}(u,\alpha)$.

3. When step 2 finishes for all nodes and all indices, the minimal total cost for the entire tree is $\min_\alpha\{\mathit{Cost}(T.\mathit{root},\alpha)\}$. The distribution α that minimizes the total cost is the optimal distribution for $T.\mathit{root}$. The optimal distributions for other nodes can be obtained by tracing back $\mathit{Dist}(u,\alpha)$ in a top-down manner, starting from $\mathit{Dist}(T.\mathit{root},\alpha)$.

The time complexity of this algorithm is $O(q^2|T|)$, where $|T|$ is the number of internal nodes in the expression tree and $q = O(m^n)$ is the number of different possible distribution n-tuples, and m is the number of index variables. The storage requirement for $\mathit{Cost}(u,\alpha)$ and $\mathit{Dist}(u,\alpha)$ is $O(q|T|)$.

8. Related work and discussion

The approach undertaken in this project bears similarities to some projects in other domains, such as the SPIRAL project which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [21, 9, 24]. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language called SPL by a systematic search through the space of possible implementations.

Other efforts in automatically generating efficient implementations of programs include FFTW [7], the telescoping languages project [10], ATLAS [23] for deriving efficient implementation of BLAS routines, the PHIPAC [1] project, and the TUNE project [22]. In addition, motivated by the difficulty of detecting and optimizing matrix operations hidden in array subscript expressions within loop nests, several projects have worked on efficient code generation from high-level languages such as MATLAB and Maple [6, 18, 19, 20].

While our effort shares some common goals with several of the projects mentioned above, there are also significant differences. Some of the optimizations we consider, such as the algebraic optimizations, memory minimization, and space-time trade-offs, do not appear to have been previously explored, to the best of our knowledge. We also take advantage of certain domain-specific properties of the computations; for example, since all expressions considered in this framework are tensor contractions, the loops of the resulting code are fully permutable, and there are no dependencies preventing fusion. This observation is crucial for the optimization algorithms of several components (memory minimization, space-time transformation, data locality). Also, some of the multi-dimensional arrays involved in the computation have certain domain-specific symmetry properties that can be exploited in order to lower the number of arithmetic operations, and thus total execution time.

While optimization of performance is a significant goal, more important in our context is the potential for dramatically reducing the developmental effort required of a quantum chemist to develop a new ab initio computational model. Currently, the manual development and testing of a reasonably efficient parallel code for a computational model such as the coupled cluster model typically takes several weeks to many months for a computational chemist. We aim to reduce the time to prototype a new model to under a day, through use of the synthesis system being developed.

References

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

[2] D. Cociorva, J. Wilkins, C. Lam, G. Baumgartner, P. Sadayappan, J. Ramanujam. Loop Optimizations for a Class of Memory-Constrained Computations. In *Proc. 15th ACM Intl. Conf. on Supercomputing*, 2001.

[3] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. *Proc. of the Intl. Conf. on High Performance Computing*, Lecture Notes in Computer Science, Vol. 2228, pp. 237–248, Springer-Verlag, 2001.

[4] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations.

Proc. of the 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[5] D. Cociorva, G. Baumgartner, C.-C Lam, P. Sadayappan, and J. Ramanujam. Memory-Constrained Communication Minimization for a Class of Array Computations. To appear in *Languages and Compilers for Parallel Computing*, 2002.

[6] L. De Rose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proc. 10th ACM Intl. Conf. on Supercomputing,* 1996.

[7] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. ICASSP 98,* Volume 3, pages 1381–1384, 1998, `http://www.fftw.org`.

[8] High Performance Computational Chemistry Group. NWChem, A computational chemistry package for parallel computers, Version 3.3, 1999. Pacific Northwest National Laboratory, Richland, WA 99352.

[9] J. Johnson, R. Johnson, D. Padua, and J. Xiong. Searching for the best FFT formulas with the SPL compiler. In *Languages and Compilers for High-Performance Computing,* Springer-Verlag, 2001.

[10] K. Kennedy, B. Broo, 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. *J. Parallel and Distributed Computing,* 2001.

[11] C.-C Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, 1999.

[12] C.-C Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Intl. Conf. on High Performance Computing*, 1999.

[13] C.-C Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage for a class of loops implementing multi-dimensional integrals. In *Languages and Compilers for Parallel Computing*, 1999.

[14] C.-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.

[15] C.-C Lam, P. Sadayappan, and R. Wenger. Optimization of a class of multi-dimensional integrals on parallel machines. In *8th SIAM Conf. on Parallel Processing for Scientific Computing,*, 1997.

[16] T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy,* pages 47–109, Kluwer Academic, 1997.

[17] J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.), *Encyclopedia of Computational Chemistry.* Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115–128, 1998.

[18] The Match Project. A MATLAB compilation environment for distributed heterogeneous adaptive computing systems. `www.ece.nwu.edu/cpdc/Match/Match.html`.

[19] Vijay Menon, Keshav Pingali. High-Level Semantic Optimization of Numerical Codes. In *Proc. ACM Intl. Conf. on Supercomputing.* 1999.

[20] Vijay Menon, Keshav Pingali. A Case for Source-Level Transformations in MATLAB. In *Proc. 2nd Conf. on Domain-Specific Languages.* 1999.

[21] J. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Puschel, and M. Veloso. SPIRAL: Portable library of optimized signal processing algorithms, 1998. `http://www.ece.cmu.edu/~spiral`.

[22] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's matrix multiplication for memory hierarchies. In *Proc. SuperComputing '98*.

[23] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. SC '98* (Electronic Publication), IEEE Publication, 1998.

[24] J. Xiong, D. Padua, and J. Johnson. SPL: A language and compiler for DSP algorithms. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2001.