

Build to Order Linear Algebra Kernels

Jeremy G. Siek*, Ian Karlin†, E. R. Jessup†

* Department of Electrical and Computer Engineering, University of Colorado
Email: jeremy.siek@colorado.edu

† Department of Computer Science, University of Colorado
Email: ian.karlin@colorado.edu, jessup@cs.colorado.edu

Abstract—The performance bottleneck for many scientific applications is the cost of memory access inside linear algebra kernels. Tuning such kernels for memory efficiency is a complex task that degrades the productivity of computational scientists. Software libraries such as the Basic Linear Algebra Subprograms (BLAS) ameliorate this problem by providing a standard interface for which computer scientists and hardware vendors have created highly-tuned implementations. Scientific applications often require a sequence of BLAS operations, which presents further opportunities for memory optimization. However, because BLAS are tuned in isolation they do not take advantage of these opportunities. This phenomenon motivated the recent addition of several routines to the BLAS that each perform a sequence operations. Unfortunately, the exact sequence of operations needed in a given situation is highly application dependent, so many more such routines are needed.

In this paper we present preliminary work on a domain-specific compiler that generates implementations for arbitrary sequences of basic linear algebra operations and tunes them for memory efficiency. We report experimental results for dense kernels, showing performance speedups of 15 to 120% relative to sequences of calls to GotoBLAS and vendor-tuned BLAS on Intel Xeon and IBM PowerPC platforms.

I. INTRODUCTION

Many scientific applications are memory bound (Gropp et al., 1999). Tuning for memory efficiency is a complex task that requires balancing interactions between algorithm, data structure, compiler, and computer architecture. To move this burden from the computational scientist to the computer scientist, standard interfaces were developed for linear algebra kernels—the Basic Linear Algebra Subprograms (BLAS) (Dongarra et al., 1990) —and highly-tuned implementations have been developed (Bilmes et al., 1997; ESSL; Goto and van de Geijn, 2008; MKL; Whaley and Dongarra, 1998). These implementations deliver peak performance on the BLAS level 3 routines (e.g., matrix-matrix multiplication) which have a high ratio of floating-point operations to memory accesses. The main optimization technique they use is blocking to improve the reuse of data in caches, registers, and the TLB (Goto and van de Geijn, 2008). However, for the BLAS level 1 and 2 operations, which have a lower ratio of floating-point operations to memory accesses, performance is a fraction of peak due to bandwidth limitations (Howell et al., 2008).

Scientific applications often require sequences of BLAS level 1 and 2 operations and many researchers have observed that such sequences, when implemented as a single specialized routine, can be optimized to reduce memory traffic (Baker et al., 2006; Howell et al., 2008; Vuduc et al., 2003). This

phenomenon motivated the recent addition of kernels such as GEMVER and GEMVT to the BLAS (Blackford et al., 2002) and their use in Householder bidiagonalization in LAPACK (Howell et al., 2008). The GEMVER kernel is a combination of outer products, matrix-vector products, and vector additions.

$$\begin{aligned} A &\leftarrow A + u_1 v_1^T + u_2 v_2^T \\ x &\leftarrow \beta A^T y + z \\ w &\leftarrow \alpha Ax \end{aligned}$$

Using the original BLAS, this sequence of operations would require two calls to the GER routine and two calls to the GEMV routine. For a matrix A larger than cache (the common case for many applications), this sequence of BLAS requires reading A from main memory four times and writing A twice. In contrast, if this kernel is implemented as a single routine, the loops can be fused so that A is only read from and written to main memory once.

Great care was taken in the original choice of BLAS to pick operations that would be widely applicable, thereby amortizing the fairly large per-routine implementation cost. The problem with larger kernels such as GEMVER and GEMVT is that they are much more application specific. To provide the best performance across a wide range of applications we need many such kernels.

In this paper we describe preliminary work on a domain-specific compiler whose input is a kernel specification, such as the above definition of GEMVER, and the desired input and output matrix formats, and whose output is a memory-efficient implementation. The key ingredients of our compiler are:

- 1) an input language that is general enough to express the BLAS and similar kernels but is relatively easy for the compiler to analyze,
- 2) a refinement-based approach that gradually lowers and optimizes a dataflow graph representation of the kernel,
- 3) an abstract representation of loops, i.e. iterations, that allows the compiler to handle a variety of matrix storage formats and parallel decompositions, and
- 4) a decoupling of the compiler into a general graph rewriting engine and a database of linear algebra knowledge.

Our compiler extends previous work on automatic tuning (Bilmes et al., 1997; Whaley and Dongarra, 1998) by raising the scope of optimization from just one linear algebra operation, such as matrix-matrix multiplication, to a sequence

```

GEMVER
in
  u1 : vector, u2 : vector,
  v1 : vector, v2 : vector,
  alpha : scalar, beta : scalar,
  y : vector, z : vector
inout A : dense column matrix
out x : vector, w : vector
{
  A = A + u1 * v1' + u2 * v2'
  x = beta * (A' * y) + z
  w = alpha * (A * x)
}

```

Fig. 1. An example specification. This kernel is the GEMVER operation from the updated BLAS (Blackford et al., 2002).

of operations. Our domain-specific approach is quite different from that of most general-purpose high-performance compilers, where the state of the art applies affine transformations to loops over dense arrays (Kandemir, 2005). While our approach is only applicable to linear algebra operations, linear algebra is an important domain and our approach enables the compiler to handle sparse as well as dense matrix formats (see section III-C).

The rest of the paper is organized as follows. Section II introduces the goals for our compiler and reviews tuning techniques for memory efficiency. Section III describes our compilation framework, and section IV compares the performance results of our prototype compiler to several BLAS implementations—GotoBLAS, MKL, and ESSL—on three dense kernels over two architectures. Section V discusses related work, and section VI outlines the considerable future research that remains to be done and presents our conclusions.

II. GENERATING MEMORY EFFICIENT KERNELS

Figure 1 shows an example input to our compiler. The name of the kernel is given at the top followed by the parameters with their kinds (matrix, vector, or scalar) and storage format. Our prototype compiler handles dense row and column-oriented matrices. The ultimate goal is to handle a wide range of matrices formats such as those found in the updated BLAS (Blackford et al., 2002). The body of the specification consists of a sequence of assignments with basic linear algebra operations on the right hand sides. Our current focus is on level 1 and 2 operations as there is more to be gained from combining and specializing them than from level 3 operations. We use the same syntax as MATLAB.

Our compiler generates an implementation of the kernel in C (a Fortran back-end is also planned) consisting of a sequence of loop nests that read and write elements from the matrices and vectors and perform arithmetic operations. The main challenge is to choose a set of loop nests that minimizes the memory traffic for a given platform.

Before discussing the details of the compiler, we review tuning techniques that reduce memory traffic. There are many techniques that improve performance in other ways, but that

is not the focus of the current paper. In general, to reduce traffic between main memory and cache, we need to improve *temporal locality* and/or *spatial locality*. A program that references the same piece of data multiple times in close succession has temporal locality. A program that successively references data that are close together has spatial locality. A cache can reduce the traffic to main memory when a program exhibits temporal locality because it stores the often-used pieces of data in higher-speed memory. A cache can also reduce traffic when a program exhibits spatial locality because data is moved from main memory to cache one cache line at a time (e.g., 128 bytes on the Pentium 4), so an access to the same cache line does not incur further traffic.

A. Loop fusion

The primary advantage of combining a sequence of BLAS into a single routine is that the loops from separate routines can be merged into a single loop, a technique called *loop fusion* (Bacon et al., 1994). If two loops reference the same matrices or vectors, the temporal locality of those references can be improved by merging the two loops into a single loop. To be a candidate for loop fusion, the loops must have compatible iterations, for example, they could both iterate over the column dimension or both iterate over the row dimension of the matrix. (There are also some conditions regarding data dependencies.) To illustrate loop fusion we show how it can be applied to the GEMVER kernel. The BLAS-based implementation of GEMVER consists of four procedure calls, each of which contains a loop nest. Below we show the loop nests and omit the procedure boundaries:

```

for j = 1:n
  A(:,j) ← A(:,j) + u1v1(j)
for j = 1:n
  A(:,j) ← A(:,j) + u2v2(j)
for j = 1:n
  x(j) ← βAT(j,:)y + z(j)
for j = 1:n
  w ← w + αA(:,j)x(j)

```

These loops can be fused into a single loop:

```

for j = 1:n
  A(:,j) ← A(:,j) + u1v1(j)
  A(:,j) ← A(:,j) + u2v2(j)
  x(j) ← βAT(j,:)y + z(j)
  w ← w + αA(:,j)x(j)

```

Assuming that a column of A can remain in cache throughout an iteration of the loop, the fused implementation only reads and writes A once from main memory. Loop fusion is also applicable at the vector-operation level. For example, the two rank-1 updates in GEMVER:

```

for i = 1:m
  A(i,j) ← A(i,j) + u1(i)v1(j)
for i = 1:m
  A(i,j) ← A(i,j) + u2(i)v2(j)

```

can be merged into a single loop:

```

for i = 1:m
  A(i,j) ← A(i,j) + u1(i)v1(j) + u2(i)v2(j)

```

Fusing the rank-1 updates reduces the memory traffic needed to write and read the result of the first rank-1 update.

While it is often beneficial to fuse loops, it is not always so. Suppose that there is only enough room in cache for two arrays of length m . Then it is not profitable to fuse the rank-1 updates in GEMVER because doing so brings three arrays of length m (u_1 , u_2 , and $A(:, j)$) through the memory hierarchy at the same time, causing some of $A(:, j)$ to be evicted. Thus the compiler needs to take into account the specifics of the architecture as well as matrix order, storage format, and sparsity.

B. Contiguous Data Access

The most efficient way to access a matrix is in the order in which the elements appear in memory. So, for example, a matrix stored in column-major order should be traversed one column at a time and a tridiagonal matrix should be traversed one diagonal at a time (Anderson et al., 1990). The reason is that contiguous access improves spatial locality. For sparse matrices, traversing according to the storage order is even more important, as traversing in non-storage order requires searching for each element. Thus, it is crucial to encode in the compiler’s database the iteration pattern that results in contiguous access for each matrix storage format.

C. Loop Blocking

Loop blocking (also called tiling) is a technique that increases temporal locality by subdividing a loop so that some data are reused before moving on to other data (Dongarra et al., 1990). Blocking is particularly effective at reducing memory traffic in BLAS level-3 kernels, where each element of a matrix is used n times. Blocking can also aid level-2 kernels when the vectors no longer fit in cache (Nishtala et al., 2004), which sometimes occurs in large sparse matrix computations. For example, consider the following matrix-vector multiplication on a column-major matrix A . To simplify the presentation, we assume A is dense.

```

for  $j = 1:n$ 
  for  $i = 1:m$ 
     $y(i) \leftarrow y(i) + A(i, j)x(j)$ 

```

Each inner loop traverses all of vector y , so if y is larger than cache then some of it will be evicted and read back in on each iteration. To apply blocking, an outer loop is introduced that iterates over blocks of y . The block size b is chosen so that blocks $y(s:e)$ and $A(s:e, j)$ fit together in cache. Thus, each block $y(s:e)$ is only read from and written to main memory once. In the following we assume that b divides m evenly.

```

for ( $s = 1, e = b; e \neq m; s += b, e += b$ )
  for  $j = 1:n$ 
     $y(s:e) \leftarrow y(s:e) + A(s:e, j)x(j)$ 

```

D. Matrix Storage Formats

Matrices often have special structure that can be exploited by an appropriate choice of storage format. For example, for a symmetric matrix there is no need to store the entire matrix, we can just store the lower (or upper) triangle. For a matrix

where all non-zero entries are close to the main diagonal, a banded storage format can be used. Not only does this reduce the amount of memory needed for the matrix, but it also speeds up computations on the matrix as no work is performed for the zero entries. Many matrices in practice are sparse, and formats such as Compressed Sparse Column (CSC) (Saad, 2003) save memory by not storing the zero elements. The CSC format stores each column as two parallel arrays: the elements and their row indices. While this greatly reduces the overall memory footprint, the bandwidth needed per floating-point operation is increased because the stored indices must be retrieved.

III. THE COMPILATION FRAMEWORK

Figure 2 gives an overview of the compilation process. The kernel specification is parsed into a dataflow graph. This dataflow graph is then iteratively processed until all of the implementation choices have been made. One example of such a choice is whether to apply an optimization to a particular point in the dataflow graph. Our prototype compiler always applies an optimization when it is legal, whether or not it may be profitable. We plan to add backtracking search to explore all the alternatives, and we discuss issues regarding choosing between alternatives, the size of the search space, and pruning in section III-E.

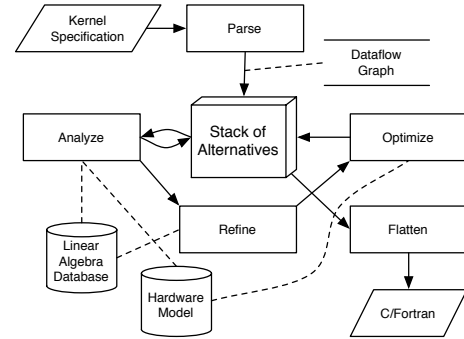


Fig. 2. Overview of the compilation process.

The compilation process consists of three phases: analysis, refinement, and optimization, which are together iterated until all of the implementation decisions have been made. The graph is then translated into C code.

A. The Dataflow Graph

An example dataflow graph for the GEMVER kernel (from figure 1) is given in figure 3. Each node represents a parameter of the kernel or an operation. The arrows indicate the flow of data. At first the graph specifies what operations are to be performed but does not contain any implementation details. For example, the symbol \times in the depicted graph stands variously for outer product, matrix-vector multiplication, and scalar-vector multiplication, and it does not yet specify, for example, whether the outer products compute a row or a column at a time of the result matrix.

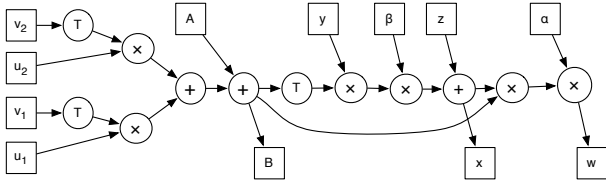


Fig. 3. Dataflow graph for the GEMVER kernel.

B. Analyze the Dataflow Graph

At the beginning of the analysis phase, each input and output node of the dataflow graph is labeled with its kind (scalar, vector, matrix, or unknown) and storage format (dense row-major, dense column-major, or unknown) according to the annotation given by the user. During analysis, the kind and storage format are computed for the intermediate nodes and algorithms are assigned to operation nodes. The choice of algorithm and the determination of kinds of storage formats are interrelated so they must be computed together. For example, the \times symbol in the graph in Figure 3 having the inputs u_1 and v_1^T can be implemented by iterating over rows first, or over columns first. The choice depends on how the result is used downstream in the dataflow graph. In this case, the result is added to the outer product of u_2 and v_2^T , so we still could choose either option as long as we make the same choice for both outer products. Going one more step downstream, there is an addition with A , which was annotated in Figure 1 to be column major. At this point it is clear that the outer-products should be computed in column-major order.

We do not hard code this knowledge of basic linear algebra in the compiler algorithm itself, but instead use a data-driven approach and store information regarding how to implement basic linear algebra in a separate database, which we refer to as the *linear algebra database*. This separation allows us to add new matrix formats, operations, and basic linear algebra algorithms without changes to the compiler algorithm. Figure 4 shows the beginning of such a database. Consider the database record for the matrix-vector multiply based on dot products of the rows of the matrix with the vector, i.e, the so-called *mv-dot* algorithm. The algorithm implements the multiplication (\times) operation for a row-major matrix (Arg 1 = R,M) and column vector (Arg 2 = C,V) and produces a column vector (Result = C,V). The *mv-dot* algorithm can be decomposed into independent operations using the equation given in the Specification column of the database, so the result of *mv-dot* may be streamed into a down stream computation using loop fusion or pipeline parallelism.

The analysis algorithm makes implementation choices using the most-constrained first strategy (also called minimum remaining values) from the artificial intelligence literature (Russell and Norvig, 2003). The compiler chooses the node with the fewest matching implementations (in the linear algebra database) and assigns a matching implementation to the node. If there is more than one match, the prototype compiler picks the first. Once we add backtracking, we'll push each alternative

onto the stack to be explored later. Once the choice is made, the kind and storage format labels for the operation node itself and for the input nodes are updated with the labels specified in the linear algebra database for the chosen algorithm. The algorithm then goes on to find the next node with the fewest matching implementations and repeats. In this phase, only the name of the algorithm is assigned to the operation node. The details of the implementation are not specified until the refinement phase.

C. Refine the Dataflow Graph

The refinement stage resolves the implementation for each operation node in the graph into a subgraph defining the details of the chosen algorithm. Each subgraph is an abstract representation of the loop that implements the given operation. In each case, the subgraph has an iteration strategy that specifies how to traverse the elements of the matrix or vector.

Figure 5 shows two steps of refinement for a matrix-vector multiplication. The first step expands the matrix-vector multiplication according to the *mv-dot* algorithm from figure 4. The refinement step replaces the \times node with a subgraph that computes the inner product of a row of the matrix—the node labeled $(i, :)$ —with the vector, storing the result in the i th element of the result vector—the node labeled (i) . The subgraph is labeled with $i = 1 : m$ indicating that the iteration strategy is to traverse the rows of the dense matrix. The second pass of refinement introduces another subgraph to implement the inner products (the dot algorithm from figure 4). The new subgraph iterates through each dense row, as indicated by the $j = 1 : n$ annotation. The sign Σ indicates that the sum of all of the iterations is computed.

It is straightforward to see how the refinement process can be augmented to handle other matrix formats, such as sparse or banded. The kinds of iteration strategies can be extended to include sparse or banded iterations. For example, suppose the matrix of figure 5 were a sparse matrix in Compressed Sparse Row format. The inner subgraph would use a sparse iteration strategy instead of a dense iteration strategy, so the j index would be retrieved from the matrix storage instead of simply incrementing from 1 to n .

D. Optimize the Dataflow Graph

In the optimization step, the dataflow graph is optimized by applying conditional graph rewrite rules. One example rewrite rule is *Merge Independent Subgraphs*: if two subgraphs share a common operand but do not depend on one another, and if the iteration strategies of the two subgraphs are compatible, then merge the subgraphs into a single subgraph. This rule is responsible for fusing the loops of the two matrix-vector products in the GEMVER kernel. Another example rule is *Merge Dependent Subgraphs*: if one subgraph depends on another subgraph, and the iteration strategies of the two subgraphs are compatible, then merge them into a single subgraph. This rule, for example, is responsible for fusing the loops of the two outer products in the GEMVER kernel.

Algorithm	Op	Specification	Arg 1	Arg 2	Result	Decompose?
v-add	+	$(x + y)(i) = x(i) + y(i)$	R/C, V	R/C, V	C, V	yes
v-trans	$(\cdot)^T$	$x^T(i) = x(i)$	R/C, V	-	C/R, V	yes
dot	\times	$x^T y = \sum_{i=1:n} x(i) y(i)$	R, V	C, V	S	no
outer	\times	$(xy^T)(i, j) = x(i) y(j)$	C, V	R, V	R/C, M	yes
v-scale	\times	$(\alpha x)(i) = \alpha x(i)$	R/C, V	S	R/C, V	yes
m-add	+	$(A + B)(i, j) = A(i, j) + B(i, j)$	R/C, M	R/C, M	R/C, M	yes
m-trans	$(\cdot)^T$	$A^T(i, j) = A(j, i)$	R/C, M	-	C/R, M	yes
mv-dot	\times	$(Ax)(i) = A(i, :) x$	R, M	C, V	C, V	yes
mv-comb	\times	$Ax = \sum_{j=1:m} A(:, j) x(j)$	C, M	C, V	C, V	no
m-scale	\times	$(\alpha A)(i, j) = \alpha A(i, j)$	R/C, M	S	R/C, M	yes

Fig. 4. A database characterizing some algorithms that implement basic linear algebra operations. Each argument is characterized as being row (R) or column (C) oriented, either a scalar (S), vector (V), or matrix (M). The database also records whether an algorithm can be decomposed into independent sub-computations.

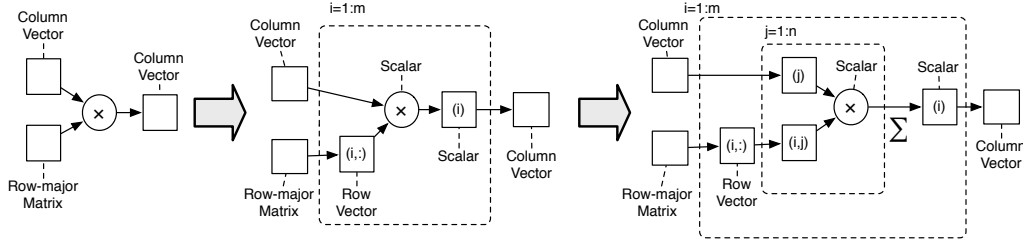


Fig. 5. The refinement of a matrix-vector product to a set of inner products and the further refinement of the inner products down to scalar multiplications and additions.

The choice of which optimizations to apply to various parts of the graph depends on the characteristics of the kernel, the algorithm, the matrix order and kind, and architectural features. The following is an instructive example: $y \leftarrow A^T A(x+z+w)$. Assume that the compiler has already merged the two vector additions into one subgraph, call it S_1 , and the two matrix products into another subgraph, call it S_2 . The question then is whether it is profitable to apply the Merge Dependent Subgraphs rule one more time to merge S_1 and S_2 . If the three vectors plus a row of A do not fit in cache, then the merger will not be profitable because it will cause each row of A to fall out of cache before the row gets used a second time. On the other hand, if all three vectors plus a row of A do fit in cache, then S_1 and S_2 should be merged. In general, we need a way to search through the many different combinations of optimization choices and choose a combination that minimizes memory traffic.

E. Search Space Exploration and Evaluation

The set of all possible combinations of implementation and optimization choices is called the search space (Kisuki et al., 2000). Although the dataflow graphs are quite small in our setting (usually less than 100 nodes), it is still important to prune the search space to avoid an exponential compilation time. Our prototype already performs some pruning: the optimization rewrite rules only apply under certain circumstances. However, as mentioned above we are sometimes overly aggressive in applying optimizations. To solve this problem we plan to use a combination of analytic model-based methods and empirical methods (Chen et al., 2007). We plan use a fast analytic cost function to prune out regions of the search space that will not produce competitive implementations. The cost function is

based on a model of the computer architecture, but does not necessarily account for all the details. Once the search space is narrowed, we will use empirical testing with representative data sets to determine the exact performance of the alternatives and choose the best one.

F. Translate the Dataflow Graph to C

A graph that cannot be further refined has been reduced to a collection of subgraphs representing loops over scalar operations. The generator outputs a C loop for each subgraph. The order in which the loops, and scalar operations within the loops, appear in the output is determined by performing a topological sort.

IV. PERFORMANCE EXPERIMENTS

In figures 7 and 8 we compare the performance of kernels generated by our prototype compiler to the performance of sequences of calls to BLAS. Figure 7 reports experiments on a 2.4 GHz Intel Xeon processor with 4GB RAM and 4Mb L2 cache. Figure 8 reports experiments on a 1.6 GHz IBM PowerPC 970 with 2.5GB RAM and 512k L2 cache. The figures show the measured Mflops rating versus matrix order for the three kernels defined in Figure 6. The graphs in the top row of figures 7 and 8 show the results for small matrices (order up to 1000) and the graphs in the bottom row show the results for large matrices (order from 1,000 to 10,000). Our primary goal was to improve performance on large matrices: for these kernels we see speedups between 15% and 120%.

The results from our prototype compiler are marked BTO for Build to Order. (We explain the curve marked BTO-vec below.) The Netlib BLAS (Netlib) are, for the most part, unoptimized and therefore serve as a baseline. Our generated

code differs from the Netlib BLAS only in that the compiler applies loop fusion; neither performs optimizations on the inner-loop such as loop unrolling, software pipelining, or vectorization. We also provide the performance numbers for the GotoBLAS (Goto and van de Geijn, 2008), the vendor-supplied Math Kernel Library (MKL), and Engineering and Scientific Subroutine Library (ESSL). These high-performance implementations include inner-loop optimizations and therefore serve as a good way to compare the benefits of those optimizations to loop fusion.

For the experiments of figure 7 we used the Intel icpc compiler with the `-O3` optimization flag for both the generated kernels and the Netlib BLAS. For figure 8 we used the IBM xlc compiler with the `-O3` optimization flag. Because the results for large and small matrices are so different, two different plots are shown for each kernel. The top rows of figures 7 and 8 show results for small matrices that fit in cache and the bottom rows show results for large matrices that do not. The results show that the inner-loop optimizations used in Goto, MKL, and ESSL give a large benefit for the small matrices but only a small benefit for large matrices. On the other hand, for matrices that are too large to fit in cache, loop fusion gives BTO a speedup of roughly 100%, 15%, and 50% compared to GotoBLAS and MKL on the Intel Xeon for `gemver`, `dgemvt`, and `bicgkernel` respectively. On the IBM PowerPC, the speedup is 100%, 5%, and 30% compared to GotoBLAS on the three kernels and 100%, 30%, and 60% compared to ESSL.

To give a rough idea of the performance of combining the inner-loop optimizations and loop fusion, we provide the curves marked BTO-vec which shows the results of compiling the BTO generated code using the `-xP` and `-noalias` flags of the underlying Intel compiler, and the `-O4`, `-qaltivec`, `-qarch=ppc970` flags of the IBM compiler. These flags enable vectorization of the inner-most loop. Combining vectorization with loop fusion results in Mflop ratings that are competitive with those of GotoBLAS and MKL for small matrices. It also gives a boost to the results for large matrix orders. BTO-vec achieves speedups of 120%, 25%, and 60% compared to Goto and MKL on the Intel Xeon. The vectorization flags, however, did not improve performance on the IBM PowerPC.

We believe that these results show promise for the automated generation of highly-tuned linear algebra kernels.

V. RELATED WORK

The MaJIC MATLAB compiler optimizes matrix expressions by re-associating them to reduce time complexity (Menon and Pingali, 1999), but it does not perform loop fusion across multiple operations as we do in this paper. Adding re-association to our compiler is an area of future work. The telescoping languages project (Kennedy et al., 2005) optimizes MATLAB scripts by analyzing and transforming library calls using techniques such as procedure specialization, strength reduction, and vectorization. However, they do not generate linear algebra library routines themselves, as we do here, so our work could complement theirs.

Several linear-algebra specific compilers in the literature generate implementations of a sparse matrix operation based on a specification written as a loop nest over a dense matrix (Bik et al., 1998; Pugh and Shpeisman, 1998). Our compiler differs in that specifications are written at a higher level (matrix algebra) and we optimize sequences of operations for memory efficiency. Domain-specific compilers are gaining momentum and there are notable examples in other domains (Allam et al., 2006; Püschel et al., 2005; Quinlan et al., 2006).

A closely related approach to domain-specific compilation is that of active libraries (Czarnecki et al., 2000). The work presented here is a generalization of earlier work by the first author on the Matrix Template Library (MTL) (Siek, 1999). There are other linear algebra libraries based on expression templates (Ahmed et al., 2000), but none of them optimize across statements due to inherent limitations of the expression template approach (Veldhuizen, 1995). There is an active library that applies loop fusion across statements at run-time using the TaskGraph library to represent dataflow (Russell et al., 2006). Our approach, in contrast, performs optimizations statically and it separates the compiler into a graph rewriting engine and linear algebra database for extensibility.

The Hierarchically Tiled Arrays (HTA) library (Bikshandi et al., 2006) provides an abstraction that makes it easier to express loop blocking—similar abstraction may be found in (Siek, 1999)—but HTA does not fully automate the implementation as we do here. Array libraries such as Blitz++ perform loop fusion to reduce memory traffic (Veldhuizen, 1998). The Formal Linear Algebra Methods Environment (FLAME) (Gunnels et al., 2001) facilitates the implementation of correct linear algebra algorithms, but it does not generate the implementations.

There is considerable literature on the optimization of arrays in languages such as HPF (Kennedy et al., 2007). Our work differs in that it is specific to linear algebra and can handle a wide range of matrix storage formats.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented a compilation framework for generating efficient implementations of arbitrary sequences of basic linear algebra operations. The compiler represents the operations in a dataflow graph and gradually refines the graph, making implementation decisions and applying high-level optimizations. The compiler is domain specific, incorporating knowledge of linear algebra data-structures and algorithms into the compiler via an extensible database. Our performance results show that this approach is promising: the compiler generates code that is 10-100% faster than state-of-the-art BLAS implementations.

There is significant research ahead of us. We plan to implement a hybrid analytic/empirical-based search of the optimization alternatives as discussed in section III-E. We also plan to expand the linear algebra database to include more matrix formats, especially sparse matrix formats. We have already begun looking at multi-core parallelization techniques and plan to incorporate those into the compilation framework

Label	Kernel	BLAS equivalent
gemver	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ $x \leftarrow \beta B^T y + z$ $w \leftarrow \alpha Bx$	<code>dcopy(m * n, A, B, 1);</code> <code>dger(m, n, 1.0, u1, 1, v1, 1, B, m);</code> <code>dger(m, n, 1.0, u2, 1, v2, 1, B, m);</code> <code>dcopy(n, z, x, 1);</code> <code>dgemv('T', m, n, beta, B, m, y, 1, 1.0, x, 1);</code> <code>dgemv('N', m, n, alpha, B, m, x, 1, 0.0, w, 1);</code>
dgemvt	$x \leftarrow \beta A^T y + z$ $w \leftarrow \alpha Ax$	<code>dcopy(m, z, x, 1);</code> <code>dgemv('T', m, n, b, A, m, y, 1, 1.0, x, 1);</code> <code>dgemv('N', m, n, alpha, A, m, x, 1, 1.0, w, 1);</code>
bicgkernel	$y \leftarrow Ax$ $z \leftarrow A^T w$	<code>dgemv('N', m, n, 1.0, A, m, x, 1, 0.0, y, 1);</code> <code>dgemv('T', m, n, 1.0, A, m, w, 1, 0.0, z, 1);</code>

Fig. 6. Kernels used in the experiments. The labels correspond to titles on the graphs in figures 7 and 8.

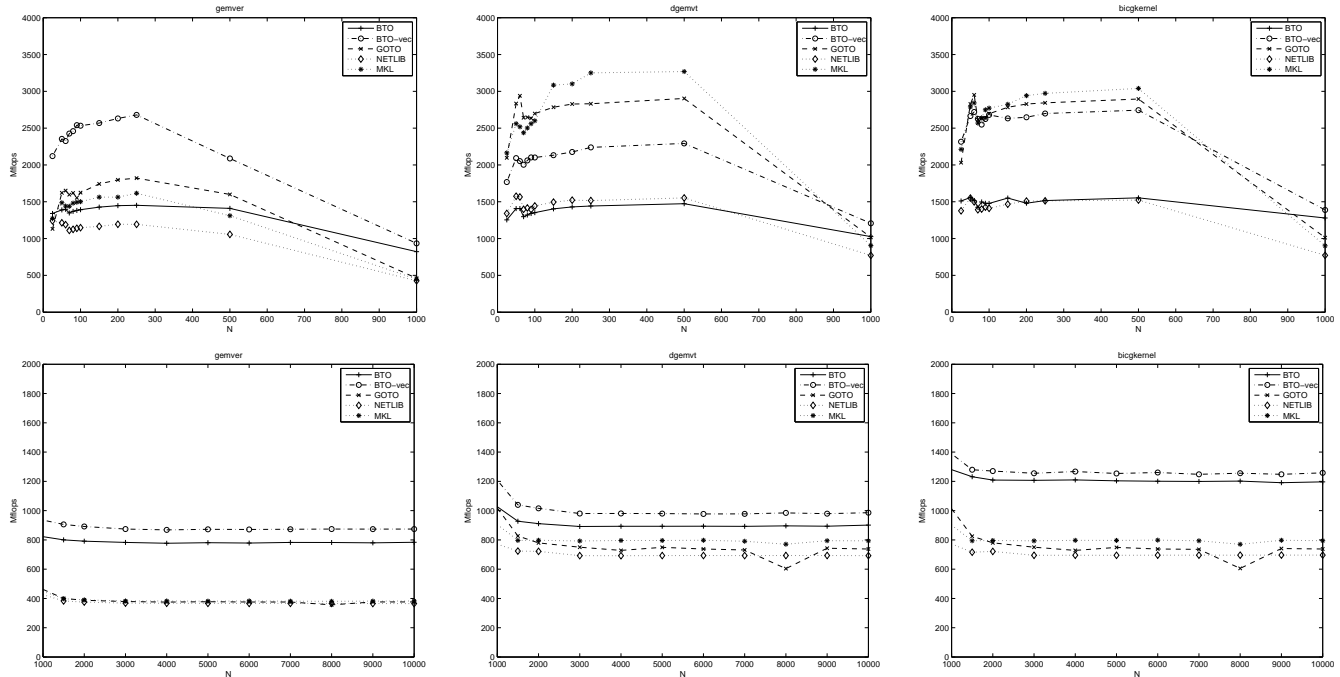


Fig. 7. Performance of our generated kernels versus implementing the kernels with a sequence of BLAS (Netlib, Goto, and Intel MKL). The graphs in the top row show the performance for small matrices (orders 10 to 1k) and the graphs on the bottom row show the performance for large matrices (1k to 10k). The experiments were run on a 2.4 GHz Intel Xeon processor with 4GB RAM and a 4Mb L2 cache.

over the next year. Generating parallel implementations should only require changing the translation to C with regards to how subgraphs are implemented. Instead of mapping a subgraph to a sequential loop, it can be mapped to several processors in a data-parallel decomposition. Alternatively, to achieve a thread-level pipeline decomposition, we can divide up the computations within a subgraph and assign them to different stages of a thread-level pipeline across two or more CPUs.

REFERENCES

N. Ahmed, N. Mateev, and K. Pingali. A framework for sparse matrix code synthesis from high-level specifications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, page 58, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

A. Allam, J. Ramanujam, G. Baumgartner, and P. Sadayappan. Memory minimization for tensor contractions using integer linear programming. *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006. doi: 10.1109/IPDPS.2006.1639717.

E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the*

1990 ACM/IEEE Conference on Supercomputing, pages 2–11, Washington, DC, USA, 1990. IEEE Computer Society. ISBN 0-69791-412-0.

D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/197405.197406>.

A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of GMRES. *SIAM J. Sci. Comput.*, 27(5):1608–1626, 2006.

A. J. C. Bik, P. J. H. Brinkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. The automatic generation of sparse primitives. *ACM Trans. Math. Softw.*, 24(2):190–225, 1998. ISSN 0098-3500. doi: <http://doi.acm.org/10.1145/290200.287636>.

G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fragueta, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-189-9. doi: <http://doi.acm.org/10.1145/1122971.1122981>.

J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PhiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-902-5. doi: <http://doi.acm.org/10.1145/263580.263662>.

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28

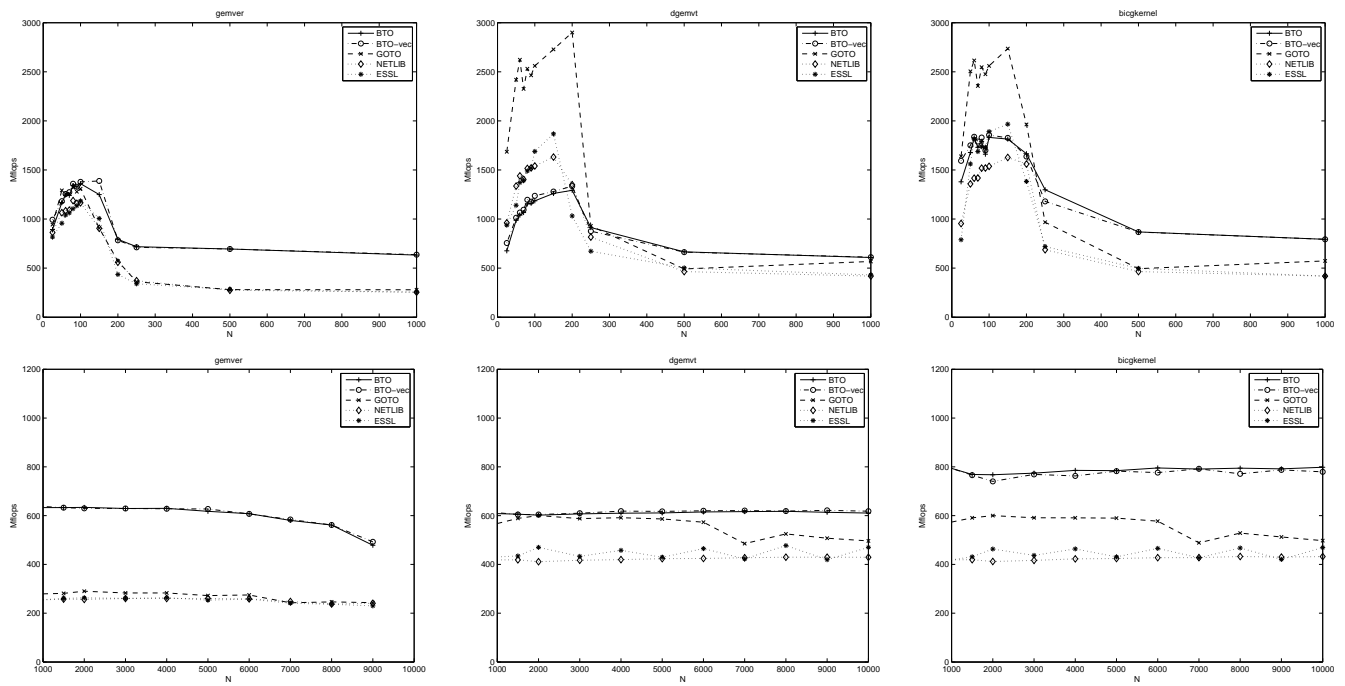


Fig. 8. Performance of our generated kernels versus implementing the kernels with a sequence of BLAS (Netlib, Goto, and IBM ESSL). The graphs on the top row show the performance for small matrices (orders 10 to 1k) and the graphs on the bottom row show the performance for large matrices (1k to 10k). The experiments were run on a 1.6 GHz IBM PowerPC 970 with 2.5GB RAM and 512k L2 cache.

- (2):135–151, 2002. ISSN 0098-3500. doi: <http://doi.acm.org/10.1145/567806.567807>.
- C. Chen, J. Shin, S. Kintali, J. Chame, and M. Hall. Model-guided empirical optimization for multimedia extension architectures: A case study. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4228369.
- K. Czarniecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, D. Musser, and R. Loos, editors, *Selected Papers from the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39, London, UK, 2000. Springer-Verlag. ISBN 3-540-41090-2.
- J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. ISSN 0098-3500. doi: <http://doi.acm.org/10.1145/77626.79170>.
- ESSL. Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL. <http://www.ibm.com/systems/p/software/essl.html>, 2007.
- K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.
- W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD'99*. Elsevier, 1999. URL citeseer.ist.psu.edu/gropp99towards.html.
- J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, 2001. ISSN 0098-3500. doi: <http://doi.acm.org/10.1145/504210.504213>.
- G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software*, 34(3), 2008. to appear.
- M. T. Kandemir. Improving whole-program locality using intra-procedural and inter-procedural transformations. *Journal of Parallel and Distributed Computing*, 65(5): 564–582, May 2005. doi: 10.1016/j.jpdc.2004.12.004. URL <http://dx.doi.org/10.1016/j.jpdc.2004.12.004>.
- K. Kennedy, B. Broom, A. Chauhan, R. J. 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(2), February 2005.
- K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1–7–22, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-766-X. doi: <http://doi.acm.org/10.1145/1238844.1238851>.
- T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0769506224. URL <http://portal.acm.org/citation.cfm?id=825767>.
- V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 434–443, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-164-X. doi: <http://doi.acm.org/10.1145/305138.305230>.
- MKL. Intel Math Kernel Library. <http://www.intel.com/cd/software/products/asm-na/eng/307757.htm>, 2007. URL <http://www.intel.com/cd/software/products/asm-na/eng/307757.htm>.
- Netlib. Netlib Repository. <http://www.netlib.org/>, 2007.
- R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical Report UCB/CSD-04-1335, University of California, Berkeley, June 2004. URL <http://www.eecs.berkeley.edu/~nrs/ucb-csd-04-1335.pdf>.
- W. Pugh and T. Shpeisman. SIPR: A new framework for generating efficient code for sparse matrix computations. In *LCPC '98: Languages and Compilers for Parallel Computing: 11th International Workshop*, volume 1656 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, August 1998.
- M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 93(2):232–275, 2005.
- D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. Annotating user-defined abstractions for optimization. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, 2006.
- F. Russell, M. Mellor, P. Kelly, and O. Beckmann. An active linear algebra library using delayed evaluation and runtime code generation. In *Workshop on Library-Centric Software Design*, 2006.
- S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 2nd edition, 2003.
- Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. ISBN 0898715342.
- J. G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, 1999.
- T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. ISSN 1040-6042. Reprinted in *C++ Gems*, ed. Stanley Lippman.
- T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, 1998.
- R. Vuduc, A. Gylassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and performance bounds for sparse $A^T Ax$. Technical Report UCB/CSD-03-1232, EECS Department, University of California, Berkeley, 2003.
- R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.