Exploring the Optimization Space for Build to Order Matrix Algebra

Geoff Belter
Dept. of ECEE
University of Colorado
Boulder, CO 80309
belter@Colorado.EDU

Elizabeth Jessup
Dept. of Computer Science
University of Colorado
Boulder, CO 80309
Elizabeth.Jessup@Colorado.EDU

Ian Karlin
Dept. of Computer Science
University of Colorado
Boulder, CO 80309
Ian.Karlin@Colorado.EDU

Thomas Nelson
Dept. of Computer Science
University of Colorado
Boulder, CO 80309
Thomas.Nelson@Colorado.EDU

Boyana Norris
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439
norris@mcs.anl.gov

Jeremy Siek
Dept. of ECEE
University of Colorado
Boulder, CO 80309
Jeremy.Siek@Colorado.EDU

ABSTRACT

The Build to Order (BTO) system compiles a sequence of matrix and vector operations into a high-performance C program for a given architecture. We focus on optimizing programs where memory traffic is the bottleneck. Loop fusion and data parallelism play an important role in this context, but applying them at every opportunity does not necessarily lead to the best performance. We present an empirical and exhaustive characterization of the optimization space for these two optimizations reporting its size and how many points in the space are close to the fastest option. We show how optimizations of different parts of the program affect one another and how the best choices depend on the computer system. We also evaluate the suitability of several algorithms for searching the space. We leverage these findings to ensure that the BTO compiler produces kernels that outperform vendor-tuned BLAS on a variety of modern computer architectures.

1. INTRODUCTION

Traditionally scientific programmers use linear algebra libraries such as the Basic Linear Algebra Subprograms (BLAS) (Dongarra et al., 1988, 1990; Lawson et al., 1979) and the Linear Algebra PACKage (LAPACK) (Anderson et al., 1999) to perform their linear algebra calculations. Programmers link their application to an implementation of these libraries that is tuned for the target machine, thereby achieving efficient, portable applications. For programs that rely on kernels with high computational intensity, such as matrix-matrix multiply, this approach can achieve near optimal performance. However, memory bandwidth, not computational capacity, limits the performance of many scientific applications (Anderson et al., 1999), with data movement expected to dominate the costs in the foreseeable future (Amarasinghe et al., 2009).

Because each BLAS routine performs a single mathematical operation, such as matrix-vector multiplication, a tuned BLAS library has a limited scope within which to optimize memory traffic. The BLAS Technical Forum suggested several new routines that combine sequences of old routines, thereby enabling loop fusion to decrease memory traffic (Blackford et al., 2002; Howell et al., 2008). However, the number of BLAS combinations that would benefit from fusion is larger than is feasible to implement for each new architecture. For example, the four new kernels added into the BLAS standard are not implemented in any major BLAS distribution.

Also, on modern parallel architectures calls to BLAS routines limits the possibilities for threading optimizations. A sequence of BLAS calls can only be computed in parallel for the duration of each call, forcing unnecessary synchronization. By combining operations, we create opportunities for greater parallelism.

To automate the generation of composed linear algebra routines for memory bound calculations, we developed the Build to Order Matrix Algebra (BTO) compiler. BTO accepts as input a sequence of matrix and vector operations in a subset of MATLAB, together with a specification of the storage formats for the inputs and outputs, and produces optimized kernels in C. With respect to storage formats, BTO currently supports row-major and column-major dense matrices. We plan to expand support to include symmetric, triangular, banded, and sparse matrices. An initial prototype of the BTO system fused loops at every opportunity (Siek et al., 2008). The next refinement of the system added the ability to explore all possible fusions and used a hybrid search strategy that combined analytic modeling with empirical performance testing (Belter et al., 2009). Later improvements to BTO included the ability to produce shared memory parallel code (Belter et al., 2010) and analytically model shared memory parallel systems (Karlin et al., 2011a).

While the hybrid exhaustive search is over a limited set of parameters, it becomes impractical when searching over large numbers of optimization parameters. In this paper, we present improvements to BTO's search capabilities and present an analysis of various search strategies for fusion combined with data parallelism. In particular, we make the following contributions:

1. We present a redesigned BTO compiler that can partition computations to exploit data parallelism and that supports multiple search strategies through a new interface (Section 2).

2. We characterize the space of all combinations of loop fusion and data parallel transformations, analyze how these transformations interact, and evaluate the extent to which they are orthogonal (Section 3.1).

SC '011 Seattle, Washington USA

3. We compare and contrast four search strategies (Section 3.3) and show how using them within BTO leads to speedups of up to 180% compared to vendor-tuned linear algebra libraries (Section 3.4).

In Section 2, we outline the workings of the BTO compiler and describe the interface for plugging search strategies into the BTO compiler. In Section 3, we discuss strategies for searching through particular problems and define the search heuristics for BTO. We also present an empirical comparison of BTO to popular linear algebra libraries for a test suite of kernels. In Section 4, we discuss related work. In Section 5, we present conclusions and future work.

2. BTO OVERVIEW

This section provides an overview of the BTO compilation system with an emphasis on two recent additions to it. The BTO compiler takes a high-level description of a sequence of matrix algebra operations, in a subset of MATLAB, and produces C code optimized for a particular target architecture.

The first release of BTO performed loop fusion and array contraction, exhaustively searched through all possible combinations of loop fusion decisions, and used analytic modeling or empirical testing to assess the profitability of each point in the search space. The new release of BTO, which we introduce here, adds a partitioning mechanism that enables data parallelism. With this addition, the number of optimization decisions has grown dramatically. The second addition to BTO is a framework for plugging in search algorithms to control which points in the search space should be evaluated.

Figure 1 shows an example input file for BTO, in this case, the GEMVT subprogram of the updated BLAS (Blackford et al., 2002). The user of BTO specifies the input types (including storage formats) and a sequence of matrix, vector, and scalar operations, but the user does not specify how the operations are to be implemented (that is, what kinds of loops, how many threads, etc.) The BTO compiler produces a C implementation in two broad steps. It first chooses how to implement the operations in terms of loops, being mindful to maximize spatial locality by traversing memory via contiguous accesses. It then searches for the combination of optimization decisions that maximizes performance. In a prior paper, we described how the BTO compiler performs loop fusion (Belter et al., 2009). In Section 2.1 of this paper, we describe the new support in BTO for partitioning computations to exploit data parallelism and in Section 3 we report on the results of searching over the combinations of loop fusion and loop parallelization decisions.

```
GEMVT
in:
    y : vector, z : vector, A : matrix
    alpha : scalar, beta : scalar
out:
    x : vector, w: vector
{
    x = beta * (A' * y) + z
    w = alpha * (A * x)
}
```

Figure 1: The BTO input file for the GEMVT kernel.

Throughout the compilation process, BTO utilizes a dataflow graph representation. Figure 2 shows the dataflow graph representation of the GEMVT kernel. The square boxes correspond to the input and output matrices and vectors, and the circles correspond to the operations. The numbers next to the circles are simply ID numbers that we use for reference later in the paper.

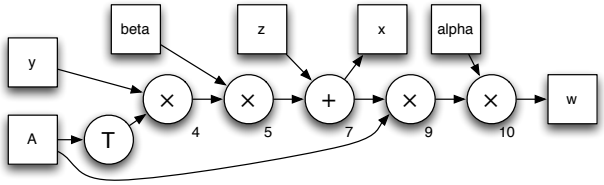


Figure 2: Example dataflow graph for GEMVT.

The BTO compiler utilizes a simple type system to keep track of which traversal order is contiguous. The type system is based on a container abstraction. Containers may be oriented horizontally or vertically, and containers may be nested inside of containers. The types are defined by the following grammar, in which R is for row, C is for column, and S is for scalar.

$$\begin{array}{lcll} \text{orientations} & O & ::= & C \mid R \\ \text{types} & \tau & ::= & O\langle\tau\rangle \mid S \end{array}$$

Figure 3 shows several types with a corresponding diagram depicting the container shape. The upper-left is a row container whose elements are scalars. The nesting of containers enables both the description of matrices, such as the row-major matrix on the right, and the description of partitioned matrices and vectors. The lower-left diagram depicts a row that has been partitioned in half by adding an outer row container. During the creation of the dataflow graph, each node in the dataflow graph is assigned a type. The input and outputs are assigned a type derived from the input file specification, whereas the types associated with intermediate results are inferred by the BTO compiler.

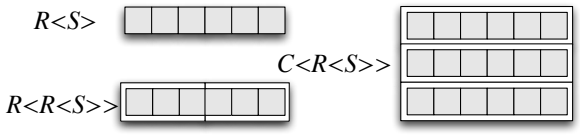


Figure 3: A vector, partitioned vector, and matrix with their corresponding types.

For each arithmetic operation, BTO recursively introduces loops to implement operations in terms of one lower dimension. For example, a matrix operation is lowered to a loop over vector operations, and the vector operations are subsequently lowered into loops over scalar operations. At each step, the loop order is chosen to match the type specification thereby ensuring contiguous memory traversals. Figure 4 shows the dataflow graph for the matrix-vector product $y \leftarrow Ax$ in which A is row-major. In the figure, loops are represented as subgraphs surrounded by dotted lines. Refer to Belter et al. (2009) for further details concerning this process.

2.1 Partitioning

To enable data parallelism and tiling for cache, we partition operations and their operands. The partitioning is accomplished by adding zero or more extra layers of containers to the type for each node in the dataflow graph. For example, the decision to partition the row in Figure 3 is encoded by changing its type from $R\langle S\rangle$ to $R\langle R\langle S\rangle\rangle$. To more conveniently discuss the rules for partitioning containers, we write a type as a list of orientations (omitting the

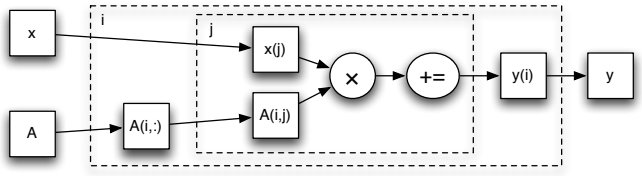


Figure 4: Example dataflow graph of the operation $y \leftarrow Ax$ with loops expressed as subgraphs.

scalar). For example, we write C, R as shorthand for $C\langle R\langle S\rangle\rangle$. We use a comma to append two lists and to add an element to the front or back of a list.

The recursive nature of the type system and our code generation process allows for the introduction of partitions with few modifications to the optimization and code generation phases of BTO.

The BTO compiler makes partitioning decisions on a per-operation basis, as directed by a search algorithm. The partitioning decisions are subject to the restrictions described in Table 1. Consider a loop that performs the operation $y \leftarrow x + z$ and look at the `add/sub` row of Table 1. In the **Operation** column, the fact that the variable τ is used for both operands specifies that the type of x and y must be identical and be of the form u, O, l, were u and l are possibly empty lists of orientations. In this case, suppose $\tau = R$, so u and l are empty and $O = R$. The **Type** column specifies which type is changed by the partitioning, in this case the type τ of x,y, and z. The new type that results from partitioning is given in the **Partition** column. In this case, we duplicate the $O = R$, adding it as an outer layer, so the resulting type is R, R.

Algo	Operation	Type	List	Partition
`add/sub`	$\tau = \tau + \tau$	τ	u, O, l	O, u, O, l
`mult-bc`	$\tau_c = \tau_a \times \tau_b$	τ_b τ_c	u_b, R, l_b u_c, R, l_c	R, u_b, R, l_b R, u_c, R, l_c
`mult-ac`	$\tau_c = \tau_a \times \tau_b$	τ_a τ_c	u_a, C, l_a u_c, C, l_c	C, u_a, C, l_a C, u_c, C, l_c
`mult-ab*`	$\tau_c = \tau_a \times \tau_b$	τ_a τ_b	u_a, R, l_a u_b, C, l_b	R, u_a, R, l_a C, u_b, C, l_b
`scale`	$\tau = S \times \tau$ $\tau = \tau \times S$	τ	u, O, l	O, u, O, l
`trans`	$\tau_t = \tau_n$	τ_t τ_n	u_t, O_t, l_t u_n, O_n, l_n	O_t, u_t, O_t, l_t O_n, u_n, O_n, l_n
			where $O_t \neq O_n$	

Table 1: Linear algebra partitioning rules. Algorithms marked with '*' require a introduction of a reduction operation.

In some situations, we would like to treat a matrix as having more than one type. Consider the earlier example GEMVT shown in Figure 1 and suppose we wish to partition the matrix-vector product Ax but not $A'y$. To partition Ax we must change the type for A, but then the type of A would be inconsistent with the non-partitioned operation $A'y$. To resolve this issue, we introduce cast nodes into the graph. A cast node represents a modification to how we view the matrix; it does not cause any actual data movement. When Ax is partitioned it will introduce a cast of A allowing the use of A in $A'y$ to remain consistent. Should two or more similar casts be introduced on the same node, the casts may be removed enabling additional optimization such as loop fusion.

2.2 Search Framework

In this section we define the space of transformations that we consider in BTO and explain how we represent a point in the search space. This representation defines an interface between the BTO compiler and search strategies that we can plug into BTO. In Section 3.2 we discuss several strategies that we have evaluated for searching the space of transformations.

The two transformations that we consider in this paper are loop fusion and shared-memory data parallelism. To enable a wide variety of search strategy plugins, such as off-the-shelf optimization algorithms, we represent a combination of transformations as an array of integers.

We must balance a trade-off between generality and searchability in designing the search space. A restrictive search space is easier to search, but if the space is too small, we risk cutting out the fastest code versions. In our space characterization we emphasize completeness, at the cost of search speed.

The integer array is composed of two parts: the first part represents fusion decisions and the second part represents parallelization decisions. To represent the fusions, we use a fully connected graph where the nodes are operations that are identified by the unique numbers attached to the operations in the dataflow graph. Each operation in the input to BTO may correspond to a nest of loops, so we need to express the depth to which fusion can occur. Thus, each edge in the graph is labeled with an integer: 0 for no fusion, 1 to fuse the top-most loop of the loop nest, 2 to fuse the top two loops, etc. This representation can express all possible fusion combinations; you cannot fuse two inner loops if you haven't yet fused their enclosing loops. On the flip side, this representation does include invalid points because in general it is not possible to fuse every pair of loops. In the near future we plan to add constraints to the search plugin interface so that we can express which combinations of fusions are illegal and thereby enable the search strategies to focus just on the legal points.

The following adjacency matrix shows a particular combination of fusion decisions for GEMVT kernel. The rows and columns of the matrix are labeled by the ID numbers for the operation nodes in the dataflow graph of Figure 2. This matrix encodes the decision to fuse operations 4 and 5 (to depth 2) and then separately fuse 7 and 9 (to depth 1).

	4	5	7	9	10
4	x	2	0	0	0
5	x	x	0	0	0
7	x	x	x	1	0
9	x	x	x	x	0

The graph need not include every pair to be fused: we take the transitive closure of the graph to determine all of the fusions. So, for example, if the search plugin specifies 0 and 1 to be fused, and 1 and 2 to be fused, but not 0 and 2, BTO will still fuse all three into a single loop. Taking the transitive closure is important for the orthogonal search strategies discussed in Section 3.2.

The second part of the array represents a table of partitions with a row for each operation in the input kernel. Each row contains three parameters that we search over. The first is a boolean value indicating whether or not to partition the operation. The second is the direction to apply the partition and the third is the number of threads to divide the computation into. To explain what we mean by "direction", consider the most general case of a matrix-matrix multiplication,

$$C_{m,n} = \sum_k A_{m,k} * B_{k,n}$$

There are three directions in which we can partition this operation: m, n, and k. If we partition along the rows of A, for example, we should also partition the rows of C (the 'm' direction). For vector

Kernel	Operation
WAXPBY	$w = \alpha x + \beta y$
ATAX	$y = A^T(Ax)$
GEMVT	$x = \beta(A^T y) + z$ $w = \alpha(Ax)$
GEMVER	$B = A + u_1 * v_1^T + u_2 * v_2^T$ $x = \beta(B^T y) + z$ $w = \alpha(Bx)$

Table 2: Kernel specifications

operations or matrix-vector operations, the possible directions are a subset of these three. These directions correspond to the rules given in Table 1. For the GEMVT example, we might have a table like the following:

Operation ID	Partition?	Direction	Number of Threads
4	1	m	4
5	1	k	4
7	0	-	-
9	1	n	2

It is straightforward to represent the partitioning table as an array of integers, thereby giving us the second part of the overall array that represents a point in the search space.

This representation can express arbitrary partitions and fusions, but it creates a huge space of points to search over. Most points in this space are either illegal or redundant. The illegal points are automatically rejected by the BTO compiler. We are in the process of adding constraints to the search plugin interface to avoid the illegal and redundant points.

In Section 3.2 we define four search strategies and compare and contrast them with respect to their ability to find good points in the space, that is, points with high performance, and with respect to how many points have to be inspected, which determines how much time it takes to perform the search.

3. EXPERIMENTAL EVALUATION

Our empirical evaluation has three parts. First, we analyze the entire space of fusion and parallelization decisions across four kernels: WAXPBY, ATAX, GEMVT, and GEMVER, shown in Table 2. ATAX is used in linear least squares calculations (Kincaid and Cheney, 2002). The other three are from the updated BLAS (Blackford et al., 2002). In particular, GEMVT and GEMVER are used in Householder bi-diagonalization, which is used to find the singular value decomposition of a matrix (Howell et al., 2008). The search space for the kernels WAXPBY, ATAX, and GEMVT is relatively small; we do an exhaustive *empirical* evaluation of the search space for these kernels. The search space for GEMVER is too large for an empirical evaluation; we instead do an exhaustive *analytic* evaluation of its search space, using a model that we developed prior to this work (Karlin et al., 2011a).

In doing these evaluations, our goal is to determine how difficult it is to find the highest-performing point in the search space and what search strategies are most effective. In particular, we are interested in which optimization parameters can be considered independently. If two code transformations are independent, the improvement from applying one transformation is consistent regardless of whether or not you apply the other transformation. If they are not independent, there may be destructive or constructive interference, where applying one transformation changes the effect of other transformations. If indeed two code transformation are independent, then we need not search over all combinations of the decisions, but can instead consider one decision after the other, thereby drastically reducing the search space.

The second part of our evaluation goes more in depth to answer the independence question by studying two variations of orthogonal search and comparing the produced routines to those found using exhaustive search and random search.

The final part of our evaluation compares the code generated by BTO for each kernel to implementations of the kernels using the vendor-tuned BLAS. For the BTO versions, we take the best point in the space, determined by the exhaustive search, and compile it to C using the BTO code generator. For the vendor-tuned versions, we select the shortest sequence of BLAS routines needed to implement the kernel.

3.1 Analyzing the Full Search Space

We found that visualizing combinations of individual optimization decisions and their interactions is extremely difficult because the space is high-dimensional, with many dimensions containing only a few discrete choices. Nevertheless, we have found that a particular kind of scatter plot is reasonably good for analyzing the space of fusions and parallel partitioning. Figures 5, 6, and 7 present the exhaustive empirical data for WAXPBY, ATAX, and GEMVT. The search space for GEMVER is too large to collect exhaustive empirical data. (The empirical, exhaustive search ran for over 24 hours without finishing.) We instead present the data for fusion but not partitioning in Figure 8. This data was collected on an Intel Core i5, with a dual-core 2.6 GHz processor and 4GB of memory.

The scatter plots can be understood as follows. Each column of boxes contains all the points with the same number of loops fused. For example, the left-most column of Figure 7 contains the points in which no loops were fused. The next column to the right has the points where two loops were fused. Each row of boxes contains the points in which the same number of operations were partitioned for parallelism. The bottom row has the point with no partitioning; the next row up has the points where only one operation was partitioned. Within each box, the x-axis is labeled with our internal version IDs and the y-axis is labeled with the runtime of the kernel in seconds, so lower is better.

Turning our attention to Figure 5, we see that the best performing WAXPBY version is in the upper-rightmost box, in which all three operations are fused and partitioned. Similarly, for the ATAX kernel (Figure 6) the upper-rightmost box contains the best performing version. For GEMVT (Figure 7) the best performing version is in the rightmost column and second-highest row. This version is fully fused (it is not legal to fuse all 5 operations) but only 4 of the 5 operations are partitioned for parallelism.

In the figures, the best versions are found in the upper right portion of the graphs, indicating that max fusions and nearly complete partitioning is required to achieve high performance. It is worth noting that in all of these cases, maximal fusion is best, however the best version of GEMVT does not have maximal partitioning. Closer examination of GEMVT shows that four of the five operations that can be fused; these four must be fused and partitioned to achieve good performance. But partitioning the fifth operation (a vector scale on the result of fused loops) can have negative effects. Closer inspection of WAXPBY shows that the four points in the upper right box corrspond to four different ways to fuse the inner loops, and completely fusing both inner and outer loops is necessary for best performance. Finally, the two points in the upper right box of ATAX represent both operations fused and partitioned, and differ only by inner loop fusion. Again fusing both inner and outer loops is necessary for best performance.

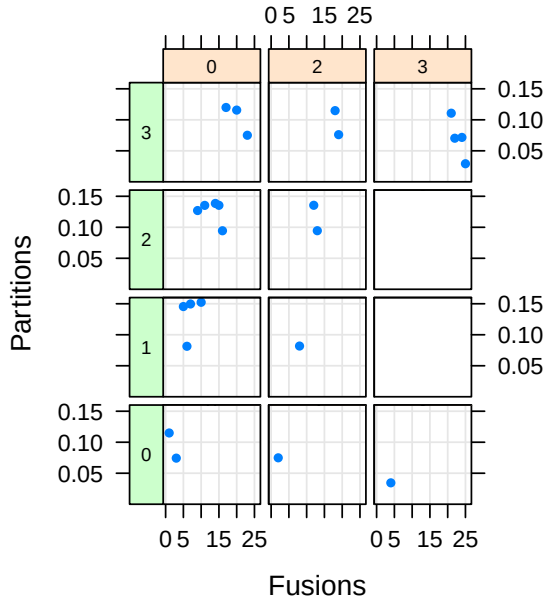


Figure 5: The space of all fusion and partitioning decisions for the WAXPBY kernel. The outer x-axis is the number of loops that are fused, while the outer y-axis is the number of operations that are partitioned for parallelism. The inner x-axis is our internal version IDs and the inner y-axis is the runtime in seconds (so lower is better).

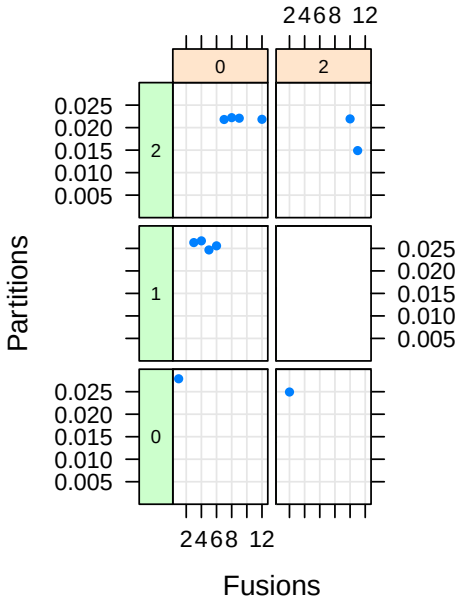


Figure 6: The space of all fusion and partitioning decisions for the ATAX kernel. The outer x-axis is the number of loops that are fused, while the outer y-axis is the number of operations that are partitioned for parallelism. The inner x-axis is our internal version IDs and the inner y-axis is the runtime in seconds (so lower is better).

Moving up a column of scatter plots increases the amount of partitioning in the program. If always adding more partitioning were beneficial, we should observe decreasing runtimes as we move up the columns of boxes. In both Figure 5 and Figure 7 we observe little or no runtime reductions moving up most of the columns. The two exceptions are in the rightmost columns.

Moving right across a row of scatter plots represents increasing amounts of loop fusion in the program. If adding more fusion always improved performance, runtimes would decrease as we move right across any given row. In the case of WAXPBY (Figure 5) the bottom row shows this trend indicating that, in the absence of partitioning, it is best to increase the amount of fusion. The second row shows similar trends. However, in the top two rows, when two or three of the operations are partitioned, the trend becomes less apparent. In contrast to WAXPBY, when moving right across any row for the GEMVT kernel (Figure 7) we observe fairly level bands of runtimes. There is no trend toward reducing runtime with more fusion.

Although the best performance often occurs with maximal fusion and partitioning, the intermediate fusion and partitioning steps do not necessarily improve performance, which makes this space difficult for direct search optimization methods. For example, applying partitioning to the unfused or partially fused kernel variants rarely provides significant runtime improvements, as can be seen by choosing a particular column and comparing the results across rows.

Fusion in GEMVER.

There are 648 unique loop fusion combinations that BTO can generate for the GEMVER kernel. The performance varies widely based on the fusion decisions, as shown in Figure 8. The x-axis is labeled with the unique combinations of fusion ranging from no fusion on the left, to all legal fusions on the right. The y-axis is labeled with the runtime of the kernel in seconds (lower is better), for which each boxplot presents the average behavior of the given grouping.

The lowest runtimes are found only on the right portion of the graph, so considerable fusion is important for high performance GEMVER. However, many versions that have high levels of fusion still perform poorly. This is evident in the performance range shown on the right portion of the graph. Moreover, in Figure 8 we observe four distinct bands of runtimes that overlap in the fusion dimension. The band with the lowest runtime (at the bottom of the graph) starts nearly two thirds of the way across the x-axis. This means that significant speedups can be achieved by focusing on a few key operations, but the best performance requires fusing a larger set.

Search Difficulty.

To determine the number of points in the search space that are close to the optimal we compared the runtime of all versions to the best version produced by the compiler. We show the number of versions within 5% of the best performing kernel in Table 3.1. For GEMVT on the Core i5 only two versions (including the best) are within 10% of the best performing version. That translates to 0.6% of the search space being close to best. On the 48 core AMD no other version other than the best was within 10% of the best performing routine. Considering only loop fusion for GEMVER: on the core 2 Duo, there are 18 versions within that threshold out of 648 (2.8%). On the 48 core AMD for GEMVER, there are two versions within 10% of best (0.2%). Finally, for GEMVER on the core i5 six (0.9%) of the versions are close to the best.

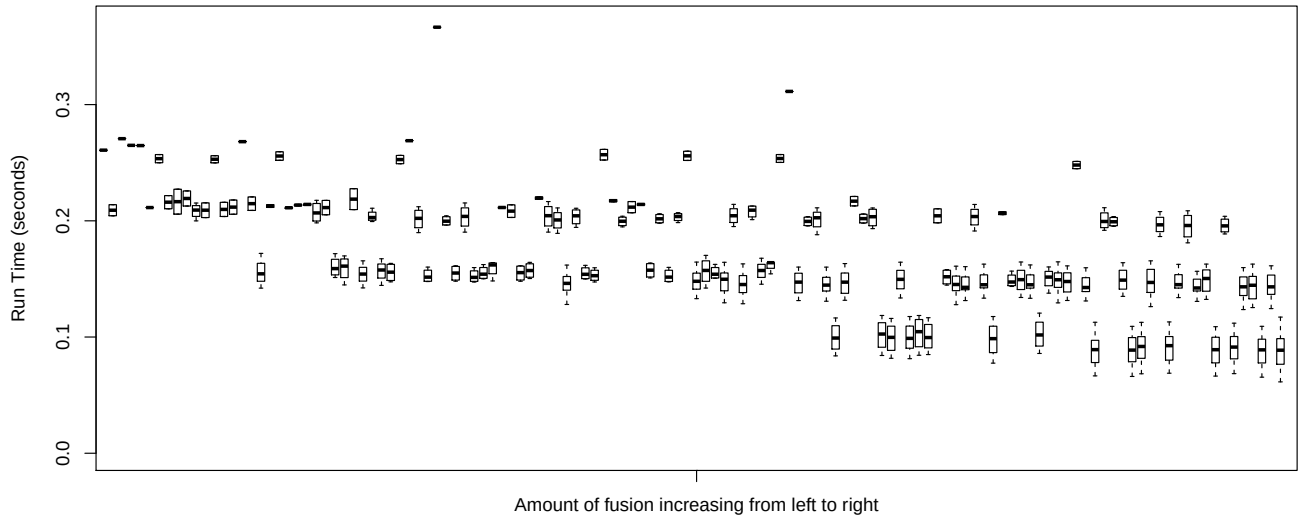


Figure 8: Performance comparison of all the fusion decisions for GEMVER. The number of loops fused increases from left to right.

Kernel	Machine	Total Points	Percent Within 10% of Best
GEMVT	Core i5	326	0.6%
GEMVT	48-core AMD	326	0.3%
GEMVER fusion	Core i5	648	0.9%
GEMVER fusion	48-core AMD	648	0.3%
GEMVER fusion	Core 2 Duo	648	2.8%

Table 3: The search space for the larger kernels, GEMVT and GEMVER, is difficult. Only a few percent or less of the points in the space are within 10% of the best with respect to runtime.

3.2 The Search Strategies

To further characterize the search space, we compare four search strategies with respect to how fast they find good points in the search space. In the following paragraphs we describe and motivate each of these strategies.

Exhaustive Search.

Exhaustive search involves testing every version of a kernel empirically. For small kernels, such as WAXPBY, ATAX, and GEMVT, it is possible to empirically test every version. However, for GEMVER, the space is too large to search exhaustively. We include exhaustive search to provide a good baseline for comparison and to provide the gold standard with respect to finding the best point in the space.

Random Search.

For *random* search we select a set of values for each tuning parameter randomly and then test the routine. Because not all possible combinations of parameters are legal, we also perform a correctness test to make sure the routine produced is valid. Random search provides a baseline for comparison with other search heuristics, as well as an indicator of how easy the space is to search. Generally speaking, if random search does well on a problem, the problem is fairly easy.

Orthogonal Search.

In *orthogonal* search, each tuning parameter in the search space is defined as a *dimension*. In our case, this means each loop fusion decision (for a pair of operations) and each partitioning option represents one dimension in the search space. Orthogonal search is carried out by varying the parameters of one dimension while holding the other dimensions constant. Then each successive dimension is explored using the best found values for the previous dimensions (Seymour et al., 2008). For problems where the optimal value of search dimensions do not influence each other, orthogonal search finds the optimal solution in all dimensions in a very short amount of time. However, when the value chosen for one dimension impacts the best value for another dimension, orthogonal search can lead to suboptimal choices. Moreover, when using orthogonal search, the order in which dimensions are explored can change the result of the search.

Hybrid Orthogonal/Exhaustive.

Our *hybrid* search strategy is a combination of orthogonal and exhaustive search. This strategy tries all combinations of fusion decisions, without partitioning. It then takes the best of those combinations and tries all combinations of partitioning decisions. This strategy is motivated by the observation that, in the kernels we have studied, the best choice of fusions is seldom affected by the partitioning decisions.

3.3 Search Strategy Comparison

To evaluate the search methods' usefulness within BTO we ran tests on the four kernels in Table 2 on an Intel Core i5, with a two-core 2.6 GHz processor and 4GB main memory. Table 4 summarizes the results of the search strategies. For each strategy and kernel, we list the empirical runtime of the best version found by the strategy, we indicate whether the search strategy found the best version, and the number of points in the space that were tested in the

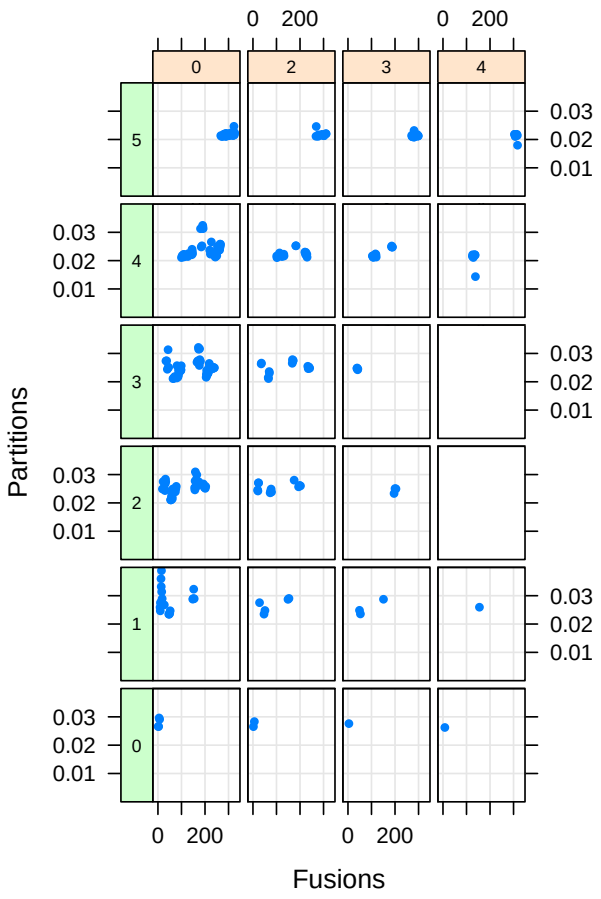


Figure 7: The space of all fusion and partitioning decisions for the GEMVT kernel. The outer x-axis is the number of loops that are fused, while the outer y-axis is the number of operations that are partitioned for parallelism. The inner x-axis is our internal version IDs and the inner y-axis is the runtime in seconds (so lower is better).

execution of the strategy.

For GEMVER, to reduce the search time, we used the memory model of Karlin et al. (2011a) instead of empirical performance testing. For both WAXPBY and ATAX, orthogonal, hybrid, and exhaustive all find the same optimal version. For GEMVT, the hybrid finds the best version, but orthogonal search fails.

For GEMVT, the two best performing versions differ only in whether or not the final scale operation is partitioned, which does not have a significant impact on performance. The exhaustive and hybrid search strategies occasionally differ in choosing to add thread partitions to that final operation, depending on noise in the runtime. The orthogonal strategy fails to find the optimal version because fusing loops 7 & 9 without also fusing 4 and 5 results in worse performance, as shown in Figure 9. This graph has boxplots showing the runtime distributions of the fusion variants without any partitioning: it is a more detailed view of the points in the bottom row of Figure 7.

The GEMVT result is interesting because fusing only one loop increases runtime, whereas fusing additional operations into that same loop reduces runtime to less than the unfused kernel. Looking at the dataflow graph in Figure 2 we see that it is fusing the

Problem	Search Strategy	Best Found Runtime (s)	Best Version Found?	Number Versions Tested
WAXPBY	Exhaustive	0.059	Yes	25
	Hybrid	0.059	Yes	4
	Orthogonal	0.058	Yes	4
	Random	0.069	No	11
ATAX	Exhaustive	0.118	Yes	12
	Hybrid	0.116	Yes	3
	Orthogonal	0.116	Yes	3
	Random	0.180	No	11
GEMVT	Exhaustive	0.111	Yes	326
	Hybrid	0.109	Yes	11
	Orthogonal	0.172	No	6
	Random	0.181	No	11
GEMVER	Exhaustive*	0.198	Yes*	8456
	Hybrid	0.198	Yes	651
	Orthogonal	0.244	No	132
	Random	-	No	-

Table 4: Comparison of the exhaustive, orthogonal, hybrid, and random search strategies on four test kernels. For GEMVER, exhaustive search is marked with * because this data is based on the predicted performance from out analytic model instead of empirical testing.

matrix-vector multiplication Ax in operation 9 into the addition from operation 7 that increases runtime. It is worth noting that the orthogonal search is particularly vulnerable to variation in the empirical timing results. For example, when trying to decide if fusing operation 4 into the fused loop with `{5,7}` tests can easily find `{4,5,7}` to be slower, especially if only a single empirical run is produced.

For GEMVER the result is similar to GEMVT: both exhaustive and hybrid find the best version and orthogonal does not. In this example, hybrid tests 7.6 % of the points that full exhaustive tests, representing a drastic reduction in search time.

For GEMVER random search strategy performed particularly poorly because all of the points it selected were illegal.

3.4 Comparison with BLAS

To place the performance of the routines generated by BTO in context, we compare BTO results with vendor tuned BLAS. We compare to both AMD's Core Math Library (ACML) and Intel's Math Kernel Library (MKL). The vendor comparison routines are comprised of a series of calls into the library. In both cases the libraries are fully threaded. Figure 10 shows the performance in MFLOPS on the y-axis and a range of matrix orders on the x-axis for several kernels and running on a 4 socket, 48 core AMD. For GEMVER, (Figure 10(a)) we show three different runs, each with a different number of threads, for BTO. Utilizing all 48 cores is not ideal; for larger matrix orders, 24 threads is optimal while for smaller matrix orders, we see that 12 threads is best. On average BTO is 2.7 times faster than ACML on GEMVER. For larger orders of GEMVT (Figure 10(b)), we observe a factor of two performance improvement, while for smaller orders ACML performs significantly better. The ATAX kernel (Figure 10(d)) shows performance similar to that of GEMVT, with BTO outperforming ACML by 2.2 times for larger matrix order, but performing worse for small order. With WAXPBY (Figure 10(c)) we see on average 1.6 times speedup over ACML. The loop fusion is the significant portion of

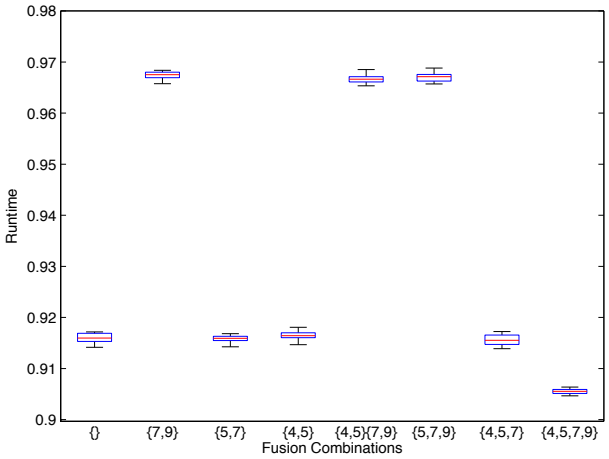


Figure 9: A comparison of runtimes for different fusion combinations of GEMVT. The operation numbers correspond to Figure 2

the performance difference and allows BTO to outperform ACML even for small matrix orders. For GEMVT, WAXPBY, and ATAX it is best to utilize only a small portion of the available cores.

Figure 11 presents the same set of experiments from a 2 core Intel i5 using the MKL BLAS library. The results for GEMVER (Figure 11(a)) and WAXPBY (Figure 11(c)) are similar with BTO outperforming Intel MKL on average by 2.8 and 2.4 times respectively. For GEMVT (Figure 11(b)) and ATAX (Figure 11(d)) BTO's throughput is 10-20% greater than MKL's throughput. We attribute the smaller performance gains relative to the tuned BLAS library to a highly optimized hand-tuned GEMV in MKL. The optimization enables two separate GEMV calls to perform nearly as well as the fused versions created by BTO, which rely on a native compiler for all but loop fusion and parallelism.

4. RELATED WORK

Loop Fusion and Parallelization.

Megiddo and Sarkar (1997) study the problem of deciding which loops to fuse in a context where parallelization choices have already been made (such as an OpenMP program). They model this problem with a weighted graph whose nodes are loops and whose edges are labeled with the run-time cost savings that would occur if the two loops were fused. The results from our experiments bring the accuracy of this model into question. For example, in Figure 7 we have a situation in which the cost savings of a combination of fusions is not simply the sum of the cost savings of the pairs of fusions in that combination. Furthermore, because the parallelization choices are fixed prior to the fusion choices, their approach sometimes misses the optimal combination of parallelization and fusion decisions.

Darte and Huard (2000), on the other hand, study the space of all fusion decisions followed by parallelization decisions. Pouchet et al. (2010) take a similar approach. They use a hybrid approach that exhaustively searches over fusion decisions then uses the polyhedral model with analytic models to make tiling and parallelization decisions. These approaches roughly correspond to the hybrid search technique in this paper.

Bondhugula et al. (2008) employ the heuristic of maximally fusing loops. Loop fusion is generally very beneficial, but too much can be detrimental as it can put too much pressure on registers and cache (Karlin et al., 2011b).

Bondhugula et al. (2010) develop an analytic model for predicting the profitability of fusion and parallelization and show speedups relative to other heuristics such as always fuse and never fuse. However, they do not validate their model against the entire search space, as we do here. It would be interesting to compare their model and their results to the best possible optimization choices found by our exhaustive search.

Search for Autotuning.

Vuduc et al. (2004) study the optimization space of applying register tiling, loop unrolling, software pipelining, and software prefetching to matrix multiplication. They show that this search space is very difficult (a very small number of combinations achieve the high performance) and they present a statistical method for determining when a search has found a point that is close enough to the best.

Balaprakash et al. (2011) study the effectiveness of several search algorithms (random search, genetic algorithms, Nelder-Mead simplex) to find the best combination of optimization decisions from among loop unrolling, scalar replacement, loop parallelization, vectorization, and register tilling as implemented in the Orio autotuning framework (Hartono et al., 2009). They conclude that the modified Nelder-Mead method is effective for their search problem. We are currently collaborating with these authors to integrate modified Nelder-Mead into the BTO compiler.

Chen et al. (2008) develop a framework for empirical search over many loop optimizations such as permutation, tiling, unroll-and-jam, data copying, and fusion. They employ an orthogonal search strategy, first searching over unrolling factors, then tiling sizes, etc. Tiwari et al. (2009) describe an autotuning framework that combines ActiveHarmony's parallel search backend with the CHiLL transformation framework.

Looptool (Qasem et al., 2003) and AutoLoopTune (Qasem et al., 2006) support loop fusion, unroll-and-jam and array contraction. AutoLoopTune also supports tiling. POET Yi et al. (2007) also supports a number of loop transformations

Partitioning Matrix Computations.

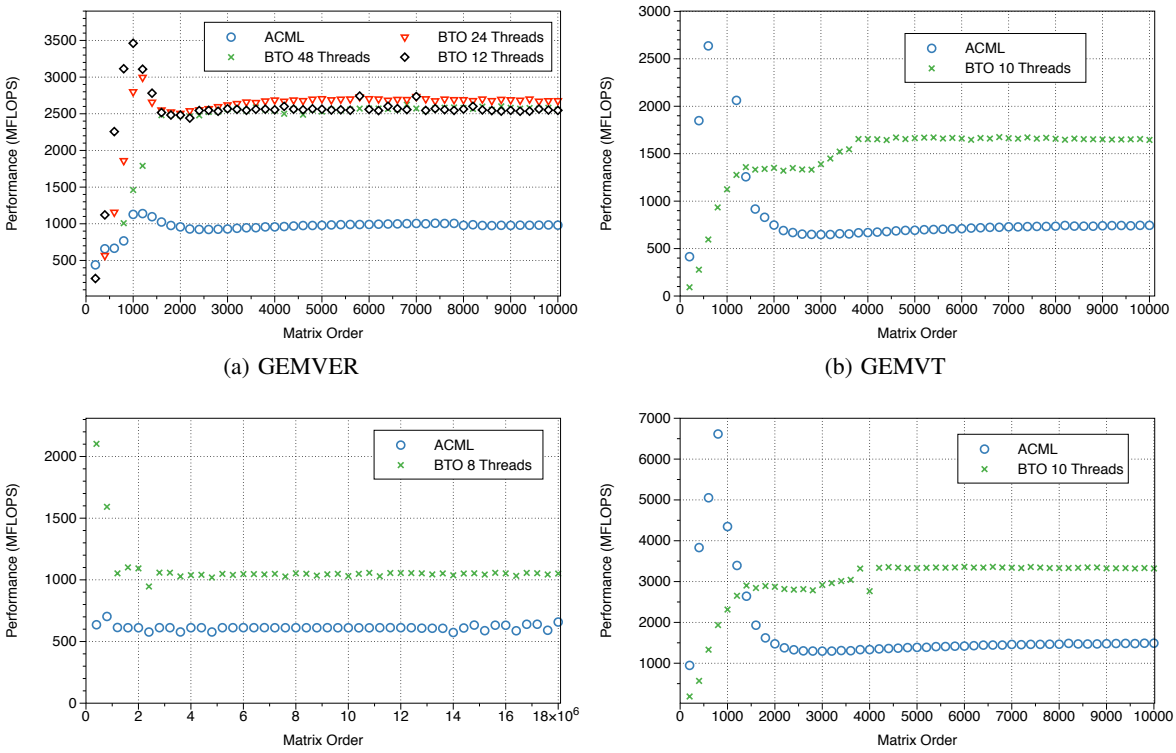
The approach to partitioning matrix computations described in this paper is inspired by the notion of a blocked matrix view in the Matrix Template Library (Siek, 1999). Several researchers have subsequently proposed similar abstractions, such as the hierarchically tiled arrays of Almasi et al. (2003) and the support for matrix partitioning in FLAME (Gunnels et al., 2001).

Search with Empirical Evaluation.

Bilmes et al. (1997) and Whaley and Dongarra (1998) autotune matrix multiplication using empirical evaluation to determine the profitability of optimizations. Zhao et al. (2005) use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Yi and Qasem (2008) apply empirical search to determine the profitability of optimizations for register reuse, SSE vectorization, strength reduction, loop unrolling, and prefetching. Their framework is parameterized with respect to the search algorithm and includes numerous search strategies.

5. CONCLUSIONS AND FUTURE WORK

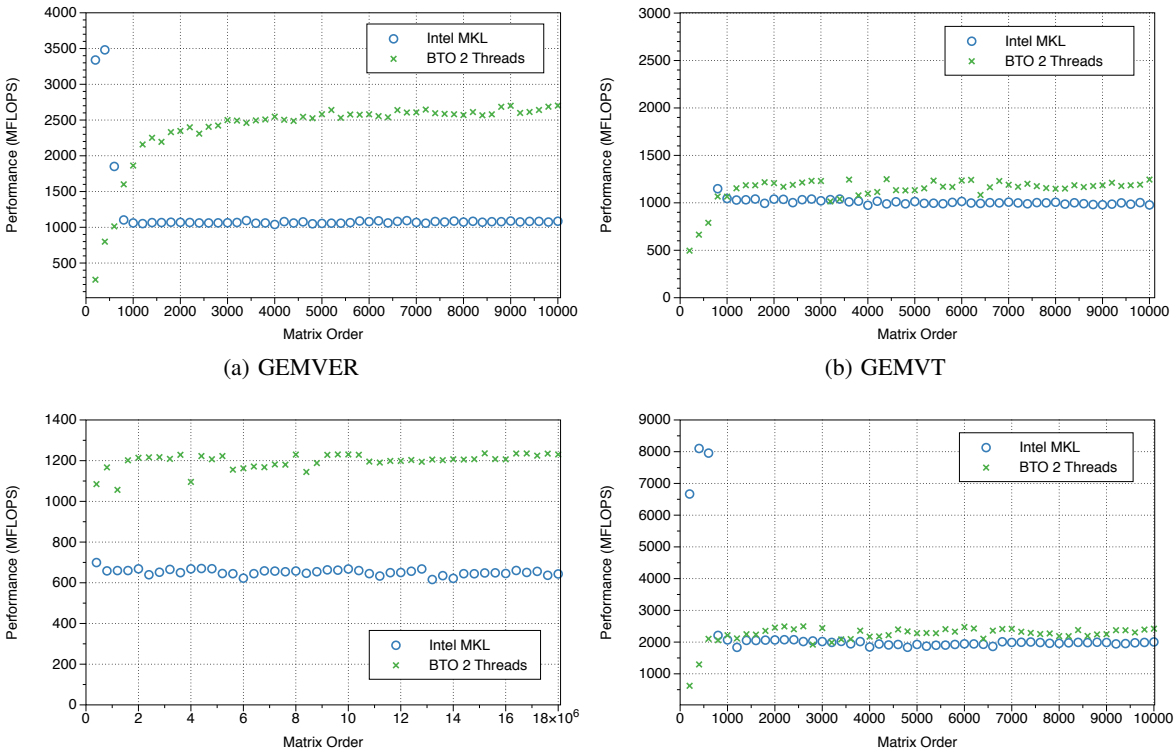
In this paper we described two extensions to the BTO compiler:



(a) GEMVER (b) GEMVT

(c) WAXPBY (d) ATAX

Figure 10: Performance comparison of BTO and Parallel ACML on a 48-core AMD Opteron.



(a) GEMVER (b) GEMVT

(c) WAXPBY (d) ATAX

Figure 11: Performance comparison of BTO and Parallel Intel MKL on a 2 Core i5.

partitioning computations for data parallelism and an interface to make the BTO compiler extensible with respect to search strategies. We present an analysis of the complexity and size of the optimization space for the combination of two optimizations: loop fusion and data parallelism. Using our new search interface, we implemented four search strategies in BTO. We compare and contrast each of these strategies and show that hybrid orthogonal/exhaustive and exhaustive search always find the optimal routine for the tests we performed, while random search never does. The orthogonal strategy only succeeds in finding the best routine for the two smallest kernels and fails to find the optimal routine for the two larger kernels. Finally, we compare the best version found by BTO to vendor BLAS implementations and show that BTO produces more efficient code.

To further improve the BTO compiler we plan on implementing other search strategies, such as a modified Nelder-Mead and genetic algorithms. We also intend to add other optimizations using our partitioning framework, such as cache tiling, vectorization, and targeting GPUs. Additionally, we plan to extend the BTO compiler to support more matrix formats, especially sparse matrices.

Acknowledgments

This work was supported by the NSF awards CCF 0846121 and CCF 0830458. This work was also supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

G. Almasi, L. D. Rose, J. Moreira, and D. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *The 16th International Workshop on Languages and Compilers for Parallel Computing*, pages 162–176, College Station, TX, 2003.

S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, et al. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*, 2009.

W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, Portland, Oregon, United States, 1999. ACM.

E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. DuCroz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorenson. LAPACK: A portable linear algebra library for high performance computers. In *Proceedings of Supercomputing '90*, pages 2–11, New York, NY, November 1990.

P. Balaprakash, S. Wild, and P. Hovland. Can search algorithms save large-scale automatic performance tuning? *Conditionally accepted for Sixth international Workshop on Automatic Performance Tuning (iWAPT2011)*, July 2011.

G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Portland, Oregon, 2009. ACM. ISBN 978-1-60558-744-8. doi: http://doi.acm.org/10.1145/1654059.1654119.

G. Belter, J. G. Siek, I. Karlin, and E. R. Jessup. Automatic generation of tiled and parallel linear algebra routines. In *In the Fifth International Workshop on Automatic Performance Tuning (iWAPT'10)*, pages 1–15, Berkeley, California, June 2010.

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 Transactions on Mathematical Software*, 28(2):135–151, June 2002.

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, pages 101–113, Tucson, AZ, June 2008.

U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM.

C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, Department of Computer Science, University of Southern California, June 2008.

A. Darte and G. Huard. Loop shifting for loop parallelization. Technical Report 2000-22, Ecole Normale Superieure de Lyon, May 2000.

J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

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.

A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: http://dx.doi.org/10.1109/IPDPS.2009.5161004. URL `http://www.mcs.anl.gov/uploads/cels/papers/P1556.pdf`. Also available as Preprint ANL/MCS-P1556-1008.

G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Trans. Math. Softw.*, 34:14:1–14:33, May 2008.

I. Karlin, E. Jessup, G. Belter, and J. G. Siek. Parallel memory prediction for fused linear algebra kernels. *SIGMETRICS Perform. Eval. Rev.*, 38:43–49, March 2011a. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1964218.1964226. URL `http://doi.acm.org/10.1145/1964218.1964226`.

I. Karlin, E. Jessup, and E. Silkensen. Modeling the memory and performance impacts of loop fusion. *Journal of Computational Science*, In Press, Corrected Proof, 2011b. ISSN 1877-7503. doi: DOI:10.1016/j.jocs.2011.03.002.

D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, third edition, 2002.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September

1979.

N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 282–291, New York, NY, USA, 1997. ACM.

L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, November 2010. IEEE Computer Society.

A. Qasem, G. Jin, and J. Mellor-Crummey. Improving performance with integrated program transformations. Technical Report TR03-419, Department of Computer Science, Rice University, October 2003.

A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing: Special Issue on Computer Science Research Supporting High-Performance Applications*, 36(9):183–196, May 2006.

K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *2008 IEEE International Conference on Cluster Computing*, pages 421–429. IEEE, 2008.

J. G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, 1999.

J. G. Siek, I. Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2008)*, pages 1–8, Miami, FL, April 2008.

A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, May 2009.

R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1): 65–94, 2004. doi: 10.1177/1094342004041293. URL `http://hpc.sagepub.com/content/18/1/65.abstract`.

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.

Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 343–355, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89739-2. doi: http://dx.doi.org/10.1007/978-3-540-89740-8_24.

Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Proceedings of the Parallel and Distributed Processing Symposium, 2007*, pages 1–8, Long Beach, CA, March 2007. IEEE. doi: 10.1109/IPDPS.2007.370637.

Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, December 2005.