A

Reliable Generation of High-Performance Matrix Algebra

THOMAS NELSON, University of Colorado, Boulder
GEOFFREY BELTER, University of Colorado, Boulder
JEREMY G. SIEK, University of Colorado, Boulder
ELIZABETH JESSUP, University of Colorado, Boulder
and BOYANA NORRIS, Argonne National Laboratory

Scientific programmers often turn to vendor-tuned Basic Linear Algebra Subprograms (BLAS) to obtain portable high performance. However, many numerical algorithms require several BLAS calls in sequence, and those successive calls result in suboptimal performance. The entire sequence needs to be optimized in concert. Instead of vendor-tuned BLAS, a programmer could start with source code in Fortran or C (e.g., based on the Netlib BLAS) and use a state-of-the-art optimizing compiler. However, our experiments show that optimizing compilers often attain only one-quarter the performance of hand-optimized code. In this paper we present a domain-specific compiler for matrix algebra, the Build to Order BLAS (BTO), that reliably achieves high performance using a scalable search algorithm for choosing the best combination of loop fusion, array contraction, and multithreading for data parallelism. The BTO compiler generates code that is between 16% slower and 39% faster than hand-optimized code.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*; *Code generation*

General Terms: Languages, Performance

Additional Key Words and Phrases: Autotuning, Domain-Specific Languages, Linear Algebra, Genetic Algorithms

1. INTRODUCTION

Traditionally, scientific programmers use linear algebra libraries such as the Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1990, 1988; Lawson et al. 1979] and the Linear Algebra PACKage (LAPACK) [Anderson et al. 1999] to perform their linear algebra calculations. A programmer links an application to vendor-tuned or autotuned implementations of these libraries to achieve efficiency and portability. For programs that rely on kernels with high computational intensity, such as matrix-matrix multiply, this approach can achieve near-optimal performance [Whaley and Dongarra 1998]. However, memory bandwidth, not computational capacity, limits the performance of many scientific applications [Anderson et al. 1999], with data movement expected to dominate the performance in the foreseeable future [Amarasinghe et al. 2009].

A tuned BLAS library can perform loop fusion to optimize memory traffic only in the limited scope of a single BLAS function, which performs a small number of mathematical operations. Moreover, separately compiled functions limit the scope of parallelization on modern parallel architectures. Each BLAS call spawns threads and must synchronize before returning, but much of this synchronization is unnecessary when considering the entire sequence of matrix algebra operations. The BLAS Technical Forum identified several new routines that combine sequences of BLAS, thereby enabling a larger scope for optimization [Blackford et al. 2002; Howell et al. 2008]. How-

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.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ever, the number of useful BLAS combinations is larger than is feasible to implement for each new architecture. Increasing the size of BLAS adds a correspondingly larger burden on library maintainers and optimizers.

Instead of using vendor-optimized BLAS, a scientific programmer can start with source code in Fortran or C, perhaps based on the Netlib BLAS, and then use a state-of-the-art optimizing compiler to tune the code for the architecture of interest. However, our experiments with two industrial compilers (Intel and Portland Group) and one research compiler (Pluto [Bondhugula et al. 2008]) show that, in many cases, these compilers achieve only one-quarter of the performance of hand-optimized code (see Section 6.2). This result is surprising because the benchmark programs we tested are sequences of nested loops with affine array accesses and the optimizations that we applied by hand (loop fusion, array contraction, and multithreading for data parallelism) are well established. Nevertheless, for some benchmarks, the compilers fail to recognize that an optimization is legal; and for other benchmarks, they miscalculate the profitability of choosing one combination of optimizations over another combination.

These observations demonstrate that achieving *reliable*, automatic generation of high-performance matrix algebra is nontrivial. In particular, the three main challenges are (1) recognizing whether an optimization is legal, (2) accurately assessing the profitability of optimizations and their parameters, and (3) efficiently searching a large, discontinuous space of optimization choices and parameters. In this article, we present our recent improvements to the Build to Order BLAS (BTO) compiler. It is the first compiler that solves all three challenges in the domain of dense matrix algebra.

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 an optimized kernel in C. This input language helps solve the problem of determining whether an optimization is legal: it makes all data dependencies explicit, so there is no difficulty recognizing whether an optimization is semantics-preserving. Further, BTO uses a carefully designed internal representation for transformation choices that rules out many illegal transformations while at the same time succinctly representing all the legal choices. To accurately assess profitability, the BTO compiler relies on a hybrid approach that we have presented in our prior work [Belter et al. 2009]: BTO uses an analytic model for coarse-grained pruning and empirical timing to make the ultimate decisions. Early prototypes of BTO are described in several papers [Belter et al. 2009, 2010; Karlin et al. 2011a; Siek et al. 2008]. The current article considers more optimizations than in our prior work and describes a new search algorithm that is scalable with respect to the number of optimizations and their parameters. In particular, we present a special-purpose genetic algorithm whose initial population is the result of a greedy, heuristic search. This search strategy quickly finds a near-optimal combination of code transformations in an otherwise intractable search space.

The following are the technical contributions of this article.

(1) We present an internal representation for optimization choices that is *complete* (includes all legal combinations of loop fusion, array contraction, and multithreading for data parallelism) but that inherently rules out many illegal combinations, thereby greatly reducing the search space (Section 4).
(2) We present a scalable and effective search strategy: a genetic algorithm with an initial population seeded by a greedy search. We describe this strategy in Section 5 and show in Section 6.2 that it produces code that is between 16% slower and 39% faster than hand-optimized code.
(3) We compare this genetic/greedy search strategy with several other strategies to reveal the rationale behind this strategy (Section 6.4).

We discuss related work in Section 7 and conclude the article in Section 8 with a brief discussion of future work. But before presenting the technical contributions, we show in the next section how the BTO compiler is used in the context of a numerical application.

2. A TYPICAL USE SCENARIO OF THE BTO COMPILER

BTO is meant to be used by long-running, performance-critical numerical applications with a significant linear algebra component. Therefore, the BTO compiler spends more time than does a typical general-purpose compiler to achieve performance that is on par with hand-tuned code. For typical examples, the BTO compiler takes less than 2 minutes to produce an optimized subprogram. Here we demonstrate the workflow of using BTO by optimizing the bidiagonalization algorithm of Howell et al. [2008]. The pseudocode for a portion of this algorithm is given in Fig. 1.

$$
\begin{aligned}
&\vdots \\
&u \leftarrow \text{householder}(A_{i:m,i}) \\
&\alpha \leftarrow -u_1 \\
&\hat{A} \leftarrow \hat{A} - \hat{u}z^T - \hat{w}\hat{v}^T \\
&v^T \leftarrow v^T + \alpha u^T \hat{A} \\
&w \leftarrow \beta \hat{A} v \\
&(s, v, k) \leftarrow \text{householder}(v) \\
&w \leftarrow (sA_{i+1} + w)/k \\
&z \leftarrow \bar{u} - \bar{u}v^T v \\
&\vdots
\end{aligned}
$$

Fig. 1. Excerpt of the bidiagonalization algorithm of Howell et al. [2008].

The first step in using BTO is to identify which parts of an algorithm can be expressed in terms of matrix and vector operations, and ideally, which of them are most time-critical for the overall execution. Those portions of the code are then written as BTO Kernel input files. The sequence of three statements starting with the assignment to $\hat{A}$ in Fig. 1 can be expressed as the kernel given in Fig. 2. This is the GEMVER kernel, as described in that work and added to the BLAS standard. The BTO compiler takes the specification in Fig. 2 and outputs a highly tuned C function. The user then compiles the C function using their native C compiler, such as the Intel C Compiler [Intel 2012].

The second step is to insert a call to the BTO-generated C function in the appropriate place within the enclosing algorithm. It is straightforward to call and link to C functions from most languages that are popular in scientific computing (especially Fortran, C, and C++). If needed, BTO could easily be modified to generate Fortran instead.

One advantage of the BTO process, in contrast to the standard BLAS-based process, is ease of readability and maintainability. By expressing complex mathematical algorithms in terms of high-

```
GEMVER
in    u : vector(column), z : vector(column), a : scalar, b : scalar
inout A : matrix(column), v : vector(column), w : vector(column)
{
  A = A - u * z' - w * v'
  v' = a * (u' * A) + v'
  w = b * (A * v)
}
```

Fig. 2. BTO kernel input file for GEMVER.

level linear algebra, the programmer creates highly legible and modifiable code. The programmer does not need to match the desired linear algebra computations against the list of existing BLAS functions. And this is done without sacrificing performance; in Section 6 we show improved performance over BLAS and hand-tuned approaches. The BTO compiler is particularly effective at optimizing sequences of matrix-vector operations (like level 2 BLAS) on large matrices and vectors, where big gains can be obtained by reducing memory traffic through loop fusion.

Table I shows the performance results of using a BLAS-based approach compared with using BTO to generate the GEMVER kernel that is used within the bidiagonalization algorithm. These results are on a 12-core Intel Westmere, doing a full bidiagonalization (the DGEBRD LAPACK routine) on matrices of size 4096 by 4096. The bidiagonalization algorithm repeatedly invokes GEMVER on smaller and smaller submatrices. The row labeled MKL BLAS in Table I is a BLAS-based implementation of GEMVER based on Howell's code. We link to Intel's MKL implementation of the BLAS. The row labeled BTO replaces the Howell GEMVER implementation with a BTO-generated kernel. The BTO kernel was autotuned for larger matrices; for matrices smaller than 1024, the BTO kernel dispatches to the BLAS. The results show a 13% overall improvement in performance by using the BTO-generated kernel compared with the BLAS-based kernel. The improvement on GEMVER alone is four times the performance of the MKL BLAS for matrices of order 2048 or greater. The bidiagonalization algorithm is an example of a typical application where BTO can improve performance even over vendor-tuned libraries. In addition, the high-level input language is easy to use and understand.

Table I. Performance results for bidiagonalization.

Version	Performance (GFLOPS)
BTO	2.69
MKL BLAS	2.38

3. OVERVIEW OF THE BTO COMPILER

This section gives an overview of the BTO compiler, emphasizing the pieces most relevant for understanding the new search algorithm. Throughout this section we use the example kernel BATAX, which performs $y \leftarrow \beta A^T Ax$ for matrix A, vectors x and y, and scalar β. The BTO specification for BATAX is given in Fig. 3.

```
BATAX
in  x : vector(column), beta : scalar, A : matrix(row)
out y : vector(column)
{
    y = beta * A' * (A * x)
}
```

Fig. 3. The BTO kernel input file for $y \leftarrow \beta A^T Ax$.

The compiler has three main components: lowering, search, and transformation. The lowering component compiles the high-level dataflow representation into a sequence of loops and scalar operations, represented as a hierarchical dataflow graph. The search component takes the lowered dataflow graph and tries to determine the best combination of transformations. For each point that the search algorithm empirically evaluates, it requests a set of code transformations, such as applying loop fusion or adding data parallelism to some of the loops. The transformation component performs those changes and then BTO compiles and executes the specified code variants and reports the performance results back to the search algorithm.

Just prior to lowering, the parser creates a dataflow graph. For example, Fig. 4 shows the dataflow graph for the BATAX kernel. The square boxes correspond to the input and output matrices and

vectors, and the circles correspond to the operations. The operations are labeled with numbers, which we use to identify the operations in the remainder of the paper. The transpose is not numbered because it doesn't require any work in the code output. Instead, BTO interprets the second matrix-vector product as occurring on the transpose of A, changing the loops without moving any data.

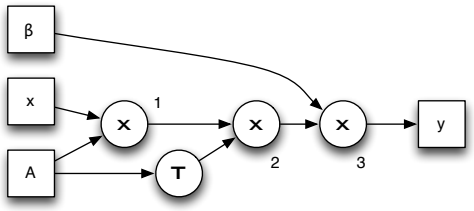

Fig. 4. Dataflow graph for $y \leftarrow \beta A^T Ax$.

In addition to the dataflow graph, the BTO compiler uses a type system based on a container abstraction to describe the iteration space of the matrices and vectors. Containers may be oriented horizontally or vertically and can be nested. We assume that moving from one element to the next in a container is a constant-time operation and good for spatial locality, but we place no other restrictions on what memory layouts can be viewed as containers. The types are defined by the following grammar, in which R designates row (horizontal traversal), C designates column (vertical traversal), and S designates scalar.

$$\begin{array}{lll} \text{orientations} & O & ::= C \mid R \\ \text{types} & T & ::= O\texttt{<}T\texttt{>} \mid S \end{array}$$

During the creation of the dataflow graph, each node is assigned a type. The input and outputs are assigned types derived from the input file specification, whereas the types associated with intermediate results are inferred by the BTO compiler.

Figure 5 shows several types with a corresponding diagram depicting the container shapes: a row container with scalar elements (upper left), a nested container for a row-major matrix (right), and a partitioned row container (lower left). Partitions are a general type construct used to introduce data parallelism into the program, as we discuss shortly.

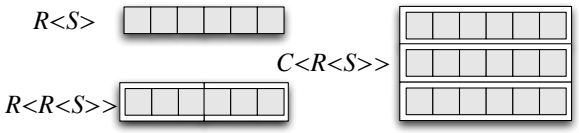

Fig. 5. A vector, partitioned vector, and matrix with their corresponding types.

3.1. Lowering to Loops over Scalar Operations

BTO lowers matrix and vector operations to vector and then to scalar operations by repeatedly examining the types of the containers and then introducing loops and appropriate lower-level operations. Table II shows some of the lowering rules for BTO. More details about these rules are given by [Belter et al. 2009]. Consider operation 1 (Ax) of the BATAX kernel. The matrix A is row-major, and the vector x is a column-vector, so we have the following types:

$$A : C\texttt{<}R\texttt{<}S\texttt{>>} \qquad \text{and} \qquad x : C\texttt{<}S\texttt{>}.$$

Table II. Linear algebra lowering rules.

Rule Name	Op and Operands	Result Type
`trans`	$O\langle\tau\rangle^T$	$O^T\langle\tau^T\rangle$
`s-add`	$S + S$	S
`add`	$O\langle\tau_l\rangle + O\langle\tau_r\rangle$	$O\langle\tau_l + \tau_r\rangle$
`s-mult`	$S \times S$	S
`dot`	$R\langle\tau_l\rangle \times C\langle\tau_r\rangle$	$\sum(\tau_l \times \tau_r)$
`l-scale`	$S \times O\langle\tau\rangle$	$O\langle S \times \tau\rangle$
	$R\langle\tau_l\rangle \times R\langle\tau_r\rangle$	$R\langle R\langle\tau_l\rangle \times \tau_r\rangle$
	$C\langle\tau_l\rangle \times R\langle\tau_r\rangle$	$R\langle C\langle\tau_l\rangle \times \tau_r\rangle$
`r-scale`	$O\langle\tau\rangle \times S$	$O\langle\tau \times S\rangle$
	$C\langle\tau_l\rangle \times C\langle\tau_r\rangle$	$C\langle\tau_l \times C\langle\tau_r\rangle\rangle$
	$C\langle\tau_l\rangle \times R\langle\tau_r\rangle$	$C\langle\tau_l \times R\langle\tau_r\rangle\rangle$

```
t0 = A * x   ⇒   for i=1:n                    ⇒   for i=1:n
                   t0(i) = A(i,:) * x               for j=1:m
                                                      t0(i) += A(i,j) * x(j)

t0 = A * x   ⇒   for j=1:m                    ⇒   for j=1:m
                   t0 += A(:,j) * x(j)              for i=1:n
                                                      t0 += A(i,j) * x(j)
```

Fig. 6. Lowering $t_0 \leftarrow Ax$ to loops and scalar operations. The pseudocode uses MATLAB notation.

Thus, the only match in Table II is the second `r-scale` rule. This rule introduces a loop (with index i) that iterates down the rows of A, as shown in the sequence of transformations at the top of Fig. 6. Inside the i loop, we have the product of a row-vector (type $R\langle S\rangle$) and a column-vector (type $C\langle S\rangle$), which is lowered by rule `dot` to another loop (with index j) that computes the inner product.

Suppose the matrix A had instead been specified in Fig. 3 as column-major. Then the type of A would be $R\langle C\langle S\rangle\rangle$ and the lowering would instead proceed as shown in the bottom sequence of Fig. 6. First, the rule `dot` applies, which introduces an loop that goes across the columns of A. The inner operation is then the product of a column-vector and a scalar, so the `r-scale` rule applies, and BTO introduces a loop that goes down each column of A, scaling it by $x(j)$.

3.2. Applying Legal Transformations

The main transformations we focus on in this article are loop fusion, array contraction, and data parallelism. Each code transformation must be checked to ensure legality: the onus of preserving correctness lies on the transformation code, not the search algorithm. This approach allows the search strategies to overapproximate the legal space safely. After correctness is verified, the code transformations are applied to the lowered loops so that the final code variant can be tested.

3.2.1. Loop Fusion and Array Contraction. Figure 7 shows an example of applying fusion to the BATAX kernel. As we saw above, the first matrix-vector product (operation 1) is lowered to a sequence of inner products. On the other hand, the second matrix-vector product (operation 2) operates on A^T, which has type, $R\langle C\langle S\rangle\rangle$, so it is lowered into a linear combination of columns of A^T (similar to the example at the bottom of Fig. 6). Now, both outer loops of operations 1 and 2 iterate over the rows of A, so they can be fused as shown in Fig. 7. The final transformation, shown in Fig. 7, is array contraction, replacing the temporary array t0 with the scalar α. Operation 3 cannot be fused with operations 1 and 2 because it depends on the temporary vector t1, and all of t1 is updated in each iteration of the i loop. Thus, the i loop must finish before the j loop may begin.

BTO breaks fusion legality into two main tests: data dependency tests and iteration space tests. Data dependency tests are based on the dataflow graph. Two operations cannot fuse if there is another node in a path between them. Figure 8 shows a case where nodes 1 and 3 cannot fuse

```
(1)  t0 = A * x
(2)  t1 = A' * t0
(3)  y = beta * t1
```

⇓

```
for i=1:m
  t0(i) = A(i,:) * x
for i=1:m
  t1 += A(i,:) * t0(i)
for j=1:n
  y(j) = beta * t1(j)
```

⇒

```
for i=1:m
  t0(i) = A(i,:) * x
  t1 += A(i,:) * t0(i)
for j=1:n
  y(j) = beta * t1(j)
```

⇒

```
for i=1:m
  α = A(i,:) * x
  t1 += A(i,:) * α
for j=1:n
  y(j) = beta * t1(j)
```

Fig. 7. Lowering $y \leftarrow \beta A^T Ax$, fusing two loops, and contracting the temporary array t0.

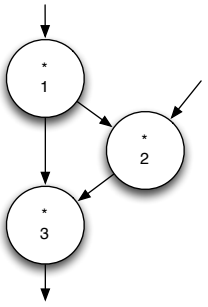

Fig. 8. Dataflow graph showing fusion potential.

because of the path from 1 to 3 through 2. Also, if the two nodes are in a pipeline (the output of one operation is an input to another operation), the first operation's output cannot be a summation (i.e., a `dot` rule). The second test is the iteration space test: two loops can fuse only if both loops are iterating over the same iteration space. This situation is determined by examing the types and seeing which lowering rule generates the loop. From the lowering rule we can determine which data elements are being iterated over and in what order, and before fusion we ensure that those iterations are the same for both loops. As a heuristic, we additionally reject fusions of loops with no shared data; this cuts down the search space without losing useful loop fusions.

3.2.2. Data Parallelism. In addition to loop fusion, BTO introduces data parallelism by partitioning the data. From the point of view of types, a partition is an additional nesting of containers. A partition splits one of the existing dimensions of the data. For example, the row-major matrix A (of type C<R<S>>) can be partitioned with horizontal cuts by adding an outer C container,

$$C<C<R<S>>>,$$

or it can be partitioned with vertical cuts by adding an outer R container,

$$R<C<R<S>>>.$$

We refer to the choice of a C or R partitioning as the *axis* of partitioning.

Figure 9 shows in pseudocode how the matrix-vector product Ax can be partitioned in two ways, corresponding to the two partitions of A described above. Recall that the container abstraction is not tied to the actual memory layout and, in the case of partitioning, does not change the physical layout of the data. Because of the extra container, lowering creates an additional abstract loop, in this case the outermost loop on the left and right of Fig. 9. During code generation, this abstract

loop is transformed into code that spawns threads. In this example b is the block size, that is, the number of iterations given to each thread.

```
for k=1:b:n                          for k=1:b:m
  for i=k:k+b-1                        for i=1:n
    for j=1:m                            for j=k:k+b-1
      t(i) += A(i,j) * x(j)                t(i) += A(i,j) * x(j)
```

Fig. 9. Two ways to partition $t = Ax$.

BTO uses the pthreads library for shared-memory parallelism and ensures that it introduces threading only where no threads will write to the same memory. When a summation must be done in parallel, BTO generates code to introduce a temporary structure to gather the results of each thread and perform the reduction after joining.

The legality of every partitioning must also be checked for each operation. In the absence of fusion, doing so is simply of a matter of checking the type of each operand and the result of a given operation. We introduce a partition to the data in an operation by choosing which of the lowering rules in Table II will generate the code for that operation. We then add either a row or column partition to whichever of the data types are involved in the partition operation. For example, the `dot` rule iterates over a row of the left argument and column of the right argument, so to apply `dot` partition to operation 1, BTO makes the following transformation:

$$C\langle R\langle S\rangle\rangle \times C\langle S\rangle = C\langle S\rangle$$
$$\Downarrow$$
$$R\langle C\langle R\langle S\rangle\rangle\rangle \times C\langle C\langle S\rangle\rangle = C\langle S\rangle$$

The partitioning above generates the code in the right of Figure 9. Similarly the code in the left of Figure 9 generated by applying an `r-scale` partition, which adds a column type to the left operand and result:

$$C\langle R\langle S\rangle\rangle \times C\langle S\rangle = C\langle S\rangle$$
$$\Downarrow$$
$$C\langle C\langle R\langle S\rangle\rangle\rangle \times C\langle S\rangle = C\langle C\langle S\rangle\rangle$$

4. REPRESENTING AND SEARCHING THE TRANSFORMATION SPACE

As we saw in the previous section, the BTO compiler can apply several transformations: loop fusion, array contraction, and data parallelism through multithreading. However, these transformations may be applied many different and conflicting ways, and BTO's job is to find which combination of code transformations results in the best performance. This is nontrivial to achieve because there are many trade-offs, such as register-pressure versus maximizing data locality for cache. We refer to all possible combinations of transformation choices as the *search space* for a given BTO Kernel. This search space is sparse, consisting of a high ratio of illegal to legal programs. Further, within the legal programs, only a handful achieve good performance. The search space is also discrete because performance tends to cluster with no continuity between clusters. Efficiently searching this space is the goal, and doing so requires a well-designed representation of the search space. This section describes the search space and the challenges with regard to representing the space efficiently. We present a domain-specific representation that enables BTO to eliminate many illegal points without spending any search time on them. This section also sets up the discussion of the new search algorithm in Section 5.

The optimization search space we consider here has three dimensions: loop fusion, axes of data partitioning, and number of threads for a parallel partition. Even considering only these three dimensions, there is a combinatorial explosion of combinations that BTO considers. The search space

could have included more choices, but some optimizations are almost always profitable. For example, BTO always applies array contraction in the process of fusing loops; the resulting reduction in memory traffic almost always improves performance. Similarly, BTO always chooses a lowering of operations that traverses matrices along contiguous memory. (For example the inner loop over a column-major matrix always traverses over a column.) If no such lowering is possible, BTO reports an error to the user.

4.1. A Straightforward but Inefficient Representation of the Search Space

To demonstrate the challenge of developing an efficient representation of the search space, we start by presenting a straightforward representation and explain why it results in an inefficient search. Loop fusion can be described as a graph, with loop nests as nodes and edges labeled with the amount of fusion between the two loop nests. Each node can have a Boolean describing whether or not it is partitioned and two integer parameters: one for the number of threads and one for the axis of partitioning. This representation can be easily encoded in a vector of bits or integers, which can then be input to off-the-shelf search algorithms.

However, this approach fails to capture important information about the search space, and, as a result, contains a large number of illegal points. For a matrix-vector product, with one level of data partitioning and a maximum thread count of 8, there are over 1.2 million combinations of loop fusion and thread parallelism. In our experiments, the search time using this representation was dominated by discarding illegal points. The problem with this representation is that it does not capture the interactions between choices in fusion and partitioning. We now summarize two important features that this representation does not encode.

Fusion is an equivalence relation. Consider the first line of the GEMVER kernel, which consists of two outer products and two matrix subtractions, each of which we have labeled with a subscript.

$$A \leftarrow A -_d u \cdot_a z^T -_c w \cdot_b v^T$$

We can represent the fusion possibilities for the above with an adjacency matrix M, where $M(i, j)$ shows the depth of fusion between the loop nests of i and j: an entry of 1 means that the outer loops are fused whereas an entry of 2 means that both the outer and inner loops are fused. Below, we show a valid fusion choice on the left and an invalid fusion choice on the right.

	a	b	c	d
a		2	2	1
b			2	1
c				1
d				

	a	b	c	d
a		2	1	1
b			2	1
c				1
d				

The matrix on the left describes fusing the outer loop of all four operations; but only a, b, and c have the inner loop fused. The matrix on the right indicates fusing the inner loop of a with b and b with c, but not a with c, which of course is impossible. We can describe these constraints as forcing the relation specified in the adjacency matrix to be an equivalence relation at every depth. That is, if operation i is fused with operation j and j is fused with k, then k must be fused with i.

Fused operations must use the same number of threads. Consider a fuse graph that specifies a fusion of operations a and b but then a partition that specifies that a use 4 threads and b use 6 threads. Partitioning the two operations with different thread counts prevents these two operations from being fused.

With a maximum thread count of 8, taking these two restrictions into account brings the number of points in the search space down to just over 1,000, or less than one-tenth of a percent of the number of points without these restrictions.

4.2. An Efficient Representation of the Search Space

Designing a representation that respects the restrictions discussed in Section 4.1 requires domain knowledge. At the expense of having to custom-build the search algorithm, we designed a representation that disallows, with no search time required, a large number of illegal points.

Loop fusion is represented by *fuse-sets*. Each operation (node in the dataflow graph) is given a unique identifier, and each abstract loop is represented with curly braces $\{\}$.

A single loop operation is represented as $\{ID\}$, where ID is a number identifying an operation node in the dataflow graph. For example, operation 3 of BATAX, scaling a vector, is represented as $\{3\}$. A two-loop operation, such as a matrix-vector product, is represented as $\{\{ID\}\}$. When discussing a specific loop, we annotate it using a subscript after the first curly brace, as in $\{_i1\}$, where i describes an axis of the iteration space. We use i to describe the iteration over rows of a matrix and j for columns of a matrix. A complete iteration space for a matrix can be described as $\{_i\{_j\}\}$ or $\{_j\{_i\}\}$. For example, operation 1 of BATAX, the matrix-vector product shown in Fig 6, can be described as $\{_i\{_j1\}\}$ (for row-major A) or $\{_j\{_i1\}\}$ (for column-major A). The fuse-set representation of all three operations of BATAX is

$$\{_i\{_j1\}\}\{_i\{_j2\}\}\{_j3\}.$$

Fusion is described by putting two operations within the $\{\}$. For example, fusing the outer loops of operations 1 and 2 of BATAX changes the fuse-set representation to

$$\{_i\{_j1\}\{_j2\}\}\{_j3\}.$$

This notation encodes the equivalence relation of loop fusion, disallowing a large number of illegal fusion combinations.

In BTO, fuse-sets are more general than described so far. They also represent the partitioning loops used for data parallelism. The matrix-vector operation of $\{_i\{_j1\}\}$ can be partitioned as $\{_{p(i)}\{_i\{_j1\}\}\}$ or $\{_{p(j)}\{_i\{_j1\}\}\}$, where the $\{\}$s annotated with $p(i)$ and $p(j)$ describe the new iteration axis and the existing i or j loop variables that the partition affects. The search tool must specify which existing loop is being modified and how many threads should be used. Fuse-sets can represent any level of nesting this way and describe both C loops and data parallelism. The following is the fuse-set description of the BATAX kernel with operations 1 and 2 fused and with partitioning added to both loops:

$$\{_{p(i)}\{_i\{_j1\}\{_j2\}\}\}\{_{p(j)}\{_j3\}\}.$$

By extending the fuse-set representation to partitioning, thread counts can be assigned to each partition fuse-set, eliminating the consideration of points with mismatched thread counts within a fused operation. BTO uses this representation to enumerate or manipulate the fuse-sets and to generate the search space. This approach allows BTO to avoid touching the majority of the illegal points one would encounter using the straightforward representation.

In the fuse-set representation, every configuration of fuse-sets represents a possible fusion, unlike with a graph-based representation. In addition, by annotating the partition fuse-sets with the relevant data for that partitioning, the effective partition search-space is reduced, throwing out nonsensical combinations of partition parameters. Figure 10 shows a graphical representation of an overly general search space and what area of that search space BTO currently searches. The gray areas represent illegal programs. This area is large, and spending time in it makes search times intractable. This section describes a representation that allows BTO to spend time only on the section labeled *BTO Considered Search Space*, which contains many fewer illegal points. To further improve search times, within the legal space, BTO prunes points it deems unlikely to be unprofitable.

4.3. Incremental Type-Pruning

Although the representation used by BTO greatly reduces the illegal points in the search space, a significant number of illegal points remain. Identifying them as early as possible is key to a fast search.

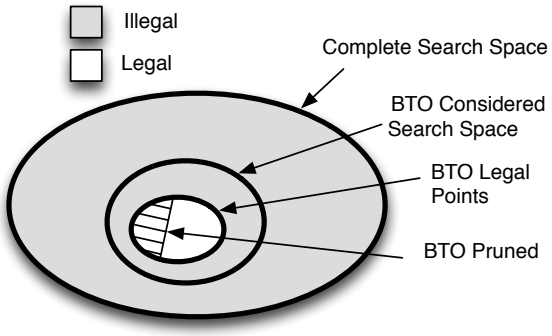

Fig. 10. Visualization of the search space, showing how BTO avoids searching a large portion of illegal points.

The representation says nothing about the data dependencies of the operations: BTO must check transformation legality using the type information and dataflow graph as described in Section 3.

One of the most difficult parts of the search is finding data partitionings that allow operations to be fused. To facilitate pruning illegal fusion-partitioning combinations, we added an incremental type-pruning approach to make it easy for any search to request legal transformations.

Before a search begins, BTO determines every legal partitioning for each operation in the kernel. Each operation has at most three possible axes of partitioning. Each of these partitioning axes has a corresponding lowering rule in Table II: `dot`, `l-scale`, or `r-scale`. For example, the two partitionings added in Fig. 9 correspond to the `l-scale` and `dot` lowering rules, respectively. In each case the lowering rule uniquely determines how the types change for the partitionings.

During the search, the search algorithm can ask for all the possible ways to partition a specific fuse-set while maintaining fusion. The type-checker looks at the pregenerated partitionings for each operation in the fuse-set and eliminates any partitioning combinations whose data-types conflict or whose partitionings create a reduction that prevents pipeline fusion. The type-checker returns a list of all combinations that are not eliminated in this way. This list may consist of zero to many combinations that work for a fuse-set, but all will be legal. This approach quickly rules out the illegal combinations, leaving only the legal points to consider.

To illustrate these ideas, we consider the partitioning and fusion choices in the BATAX kernel. Below we show two ways to partition operation 1 of BATAX, `t0 = A * x`. We use MATLAB's colon notation for a complete iteration and k for the subblock on which to operate in parallel. On the right is the representation as a fuse set. Partitioning (X) cuts the rows of A and vector $t0$, while partitioning (Y) cuts the columns of A and the vector x. Partitioning (Y) leads to a reduction at the parallel level, so $t0$ is not available for use until after a join.

$$\begin{array}{ll|l} (X) & t0(k) = A(k,:) \times x & \{_{p(i)}\{_i\{_j 1\}\}\} \\ (Y) & t0 \mathrel{+}= A(:,k) \times x(k) & \{_{p(j)}\{_i\{_j 1\}\}\} \end{array}$$

Operation 2 of BATAX, `t1 = A'* t0`, can be partitioned in the following ways.

$$\begin{array}{ll|l} (Z) & t1(k) = A(k,:) \times t0 & \{_{p(j)}\{_i\{_j 2\}\}\} \\ (W) & t1 \mathrel{+}= A(:,k) \times t0(k) & \{_{p(i)}\{_i\{_j 2\}\}\} \end{array}$$

BTO generates all four of these partitioning possibilities before search begins. Then, at some point during the search, the search algorithm may ask how to partition operations 1 and 2 while keeping them in the same fuse-set. Data dependence analysis says that partition (Y) of operation 1 will cause fusion to fail, because (Y) introduces a reduction, so operation 1 must be partitioned by using method (X). However, this limits the options for operation 2. Since matrix A is shared, in order to achieve fusion after partitioning, A needs to be accessed the same way in both partition

```
Search(program, N, num_generations) =
    organism1 ← Max−Fuse(program)
    population ← N randomly chosen mutations of organism1
    for j = 1 to num_generations do
        empirically evaluate the population
        new_population ← {Best Organism Found}
        for i = 1 to N do
            parent1 ← the better performing of two randomly chosen organisms from the population
            parent2 ← the better performing of two randomly chosen organisms from the population
            new_population ← new_population ∪ { crossover(parent1, parent2) }
        mutate every organism within new_population
        population ← new_population
    return the best performing organism within the population
```

Fig. 11. Pseudocode for the MFGA search algorithm.

loops. From partitioning (X) we see that A is accessed as $A(k,:)$. Because operation 2 accesses the transpose of A, we must select partitioning (W), accessing A^T as $A(:,k)$. In terms of the types, both (X) and (W) convert A from type C<R<S>> to type C<C<R<S>>>. This can also be expressed in the fuse-set notation: the partitions introduced in (X) and (W) both generate $\{_{p(i)}\}$. In large fuse-sets, the likelihood of finding a correct set of operation partitions randomly is small. BTO uses this more intelligent approach to quickly work through the type constraints. In this example BTO successfully prunes the space and chooses (X) and (W) without having to individually test each fusion/partitioning combination.

5. GENETIC/GREEDY SEARCH STRATEGY

This section describes the BTO search strategy, which is based on a genetic algorithm whose initial population is determined by a greedy search that tries to maximally fuse loops. We refer to this search strategy as MFGA, for Maximal Fusion followed by a Genetic Algorithm. Section 6 presents empirical results that support this design compared to some alternatives.

Genetic algorithms are a category of global optimization metaheuristics inspired by biological evolution [Mitchell 1998]. In our setting, a set of transformations choices creates a code variant, which plays the role of an organism in the genetic algorithm. A genetic algorithm operates on a population of organisms. At each generation, the worst organisms are removed from the population and are replaced with newly generated organisms. Genetic algorithms do not always provide simple off-the-shelf solutions because the results are highly sensitive to (1) a representation for organisms (which we accomplished in Section 4), (2) a definition of mutation, and (3) a definition of crossover. We present the mutation and crossover operations in this section. A high-level outline of the MFGA algorithm is given in Figure 11.

The user can run the genetic algorithm for as much or as little time as desired by specifying the number of generations. We also include an option to terminate the search if there has been no improvement after a user-defined period of time. The GEMVER kernel is representative of the largest kernels that the BTO is intended to be used on, and it takes less than 2 minutes to find the best optimizations for it by using MFGA.

5.1. The Max-Fuse Greedy Search

The search begins with a greedy Max-Fuse (MF) heuristic: we attempt to fuse as many of the loops as possible to the greatest depth possible, using the representation described in Section 4. The MF search starts from unfused but partitioned variants of the kernel. Continuing with the BATAX example, the following fuse-sets represent the unfused but partitioned kernel. The X, Y, and Z are unknowns that represent the partitioning axis and are solved during the MF search.

$$\{_X\{_i\{_j 1\}\}\}\{_Y\{_i\{_j 2\}\}\}\{_Z\{_j 3\}\}$$

The MF search attempts to fuse loops in a depth-first manner, starting with outermost loops and moving toward the innermost loops. The search is greedy in that it immediately fuses two loops when doing so is legal, even though that fusion might disable another opportunity for fusion somewhere else in the program. The MF search uses the dataflow graph and type constraints to check legality. Suppose MF can fuse the loops labeled X and Y. To fuse these loops, we need $X = Y$, so we proceed with the fusion and constrain ourselves to $X = Y$.

$$\begin{aligned} &\{_X\{_i\{_j1\}\}\}\{_X\{_i\{_j2\}\}\}\{_Z\{_j3\}\} \\ \Rightarrow\ &\{_X\{_i\{_j1\}\}\{_i\{_j2\}\}\}\{_Z\{_j3\}\} \end{aligned}$$

Recall from Section 4.3 that only one partitioning axis allows fusion of the two operations. Referring to the dataflow graph and type constraints, MF realizes that X must be $p(i)$ because the alternative, $p(j)$, would mean that the necessary results from operation 1 would not be available for operation 2. After fusing outermost loops, MF proceeds to fuse loops at the next level down. In this example, MF fuses the i loops surrounding operations 1 and 2.

$$\begin{aligned} &\{_{p(i)}\{_i\{_j1\}\}\{_i\{_j2\}\}\}\{_Z\{_j3\}\} \\ \Rightarrow\ &\{_{p(i)}\{_i\{_j1\}\{_j2\}\}\}\{_Z\{_j3\}\} \end{aligned}$$

Going down one more level, MF considers fusing the j loops surrounding operations 1 and 2; but as we discussed in Section 3.2, $\{_j2\}$ depends on the result of $\{_j1\}$, and $\{_j1\}$ is an inner product, so its result is not ready until the loop is complete, and therefore it cannot be fused with $\{_j2\}$.

Popping back to the outermost loops, MF next considers whether the $p(i)$ loop can be fused with Z. The $p(i)$ loop requires a reduction before the final vector scaling of operation 3, so 3 must reside in its own thread. Finally, there is only one axis of iteration in operation 3, so Z must be $p(j)$. Therefore, the MF search produces the following organism:

$$\{_{p(i)}\{_i\{_j1\}\{_j2\}\}\}\{_{p(j)}\{_j3\}\}.$$

In the worst case, each loop will try unsuccessfully to fuse with each other loop. Thus the MaxFuse greedy heuristic is $O(n^2)$, where n is the number of loops in the program.

5.2. Mutation

Our mutation operator applies *one* of the following four changes: (1) add or remove fusion, (2) add or remove a partitioning, (3) change the partition axis, or (4) change the number of threads. Mutations are constrained to the set of legal organisms; for example, attempting to further fuse an already maximally fused organism will fail, resulting in no change. However, mutations might randomly remove fusions or partitions. The following shows the removal of a partition from operation 3.

$$\{_{p(i)}\{_i\{_j1\}\{_j2\}\}\}\{_{p(j)}\{_j3\}\} \Rightarrow \{_{p(i)}\{_i\{_j1\}\{_j2\}\}\}\{_j3\}$$

5.3. Selection and Crossover

After the initial generation of organisms and each subsequent generation, we compile and empirically evaluate every organism and record its runtime to serve as the organisms' *fitness*, that is, the value the search tries to minimize. We then select $2N$ of the fittest organisms to be parents for the next generation, where the population size N can be user specified but defaults to 20.

Parent Selection Method. The population evolves through tournament selection [Mitchell 1998]: k random organisms are chosen to be potential parents, and the one with the best fitness becomes an actual parent. This process balances hill climbing with exploration, allowing less-fit organisms to sometimes become parents, and thus helping the algorithm escape locally optimal solutions that are not globally optimal. Larger values of k cause the algorithm to converge more quickly on a solution, whereas smaller values of k cause the algorithm to converge more slowly but increase exploration. BTO uses $k = 2$ to favor exploration.

Crossover. The crossover function takes two parent organisms and randomly chooses features of the two parents to create a child organism. The key strength of genetic algorithms is that crossover can sometimes combine the strengths of two organisms. Our crossover function generates the child recursively from two parents, making fusion decisions at each level and ensuring that those decisions remain valid for inner levels.

Our crossover function uses the fuse-set representation of each expression as described in Section 3 and performs crossover by comparing the two fuse-sets. Continuing with the BATAX example, consider the following two organisms a and b.

$$a : \{_{p(i)}\{_i\{_j 1\}\{_j 2\}\}\}\{_j 3\}$$
$$b : \{_{p(i)}\{_i\{_j 1\}\}\}\{_{p(i)}\{_i\{_j 2\}\}\}\{_{p(j)}\{_j 3\}\}$$

Parent a partitions and partly fuses operations 1 and 2 but does not partition operation 3. Parent b has all partitions turned on but has not fused operations 1 and 2.

Crossover chooses which parent to emulate for each operation, working from the outermost fuse level inward. Each step constrains the possibilities for the other operations. In our example, crossover might choose parent a for the outermost level of operation 1, meaning 1 and 2 exist in the same thread (also using parent a's partitioning axis $p(i)$ and thread number choice). Crossover then might choose parent b for the next level, iteration i. This mechanism forces operation 1 and 2 not to be fused, resulting in $\{_i\{_j 1\}\}\{_i\{_j 2\}\}$. Then the crossover moves to operation 3, and the process continues. If b is chosen, the final child becomes $\{_{p(i)}\{_i\{_j 1\}\}\{_i\{_j 2\}\}\}\{_{p(j)}\{_j 3\}\}$.

The genetic algorithm repeats the tournament selection process N times, creating a new generation of organisms. It caches fitness values: if crossover ever produces an organism that was already tested in a previous generation, the genetic algorithm uses the old fitness in order to save search time.

5.4. Search for Number of Threads

BTO uses a fixed number of threads to execute all of the data-parallel partitions in a kernel. We refer to this as the *global thread number* heuristic. An alternative is to use potentially different numbers of threads for each partition, which we refer to as the *exhaustive thread* search. In Section 6.4.3, we present data that shows that the exhaustive approach takes much more time but does not lead to significantly better performance.

BTO includes the search for the best number of threads in the MFGA algorithm. The initial number is set to the number of cores in the target computer architecture. The mutation function either increments or decrements the thread number by 2. The crossover function simply picks the thread number from one of the parents. After the genetic algorithm completes, MFGA performs an additional search for the best number of threads by testing the performance when using thread counts between 2 and the number of cores, incrementing by 2.

6. RESULTS

We begin this section with a comparison of the performance of BTO-generated routines and several state-of-the-art tools and libraries that perform similar sets of optimizations, as well as hand-optimized code (Section 6.2). BTO generates code that is between 39% faster and 16% slower than hand-optimized code. For only a few of the kernels do other automated tools and libraries achieve comparable performance to BTO and hand-optimized code.

The later parts of this section evaluate the MFGA algorithm in more detail. We first compare MFGA to an exhaustive search, showing that MFGA finds routines that perform within 2% of the best possible routine (Section 6.3). We next present empirical results that explain our choices in design of the MFGA algorithm (Section 6.4). We defend the choice of starting with a greedy search based solely on fusion (instead of fusion and parallelism combined), we show why the genetic algorithm is needed in addition to the greedy search, and we justify using the same thread count for all of the parallel loops in a kernel.

```
// q = A * p
dgemv('N',A_nrows,A_ncols,1.0,A,lda,p,1,0.0,q,1);
// s = A' * r
dgemv('T',A_nrows,A_ncols,1.0,A,lda,r,1,0.0,s,1);
```

Fig. 12. Example sequence of BLAS calls that implement BICGK.

6.1. Test Environment and Kernels

The results in this section are for the kernels shown in Table III. Some of these kernels respond well to loop fusion and data parallelism, whereas others do not. Some of these kernels are in the updated BLAS [Blackford et al. 2002] but have not been adopted by vendor-tuned BLAS libraries. These kernels also represent various uses of the BLAS. For example, the DGEMV kernel maps directly to a BLAS call while others are equivalent to multiple BLAS calls. As an example, Fig. 12 shows the sequence of BLAS calls that implement the BICGK kernel. The first three kernels in Table III are vector-vector kernels; the rest are matrix-vector kernels.

Table III. Kernel specifications.

Kernel	Operation
AXPYDOT	$z \leftarrow w - \alpha v$ $\beta \leftarrow z^T u$
VADD	$x \leftarrow w + y + z$
WAXPBY	$w \leftarrow \alpha x + \beta y$
ATAX	$y \leftarrow A^T A x$
BICGK	$q \leftarrow Ap$ $s \leftarrow A^T r$
DGEMV	$z \leftarrow \alpha Ax + \beta y$
DGEMVT	$x \leftarrow \beta A^T y + z$ $w \leftarrow \alpha Ax$
GEMVER	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ $x \leftarrow \beta B^T y + z$ $w \leftarrow \alpha Bx$
GESUMMV	$y \leftarrow \alpha Ax + \beta Bx$

The computers used for testing include recent AMD and Intel multicore architectures, which we describe in Table IV. We ran performance experiments on square matrices of order 10,000 and vectors of dimension 10,000 for matrix-vector computations, and vectors of dimension 1,000,000 for vector-vector computations. We filled the matrices and vectors with random numbers. BTO is designed for the user to specify the problem size. In future work, we plan to investigate having BTO generate kernels that perform well over a range of sizes by using the standard approach of dispatching to differently optimized kernels based on size.

Table IV. Specifications of the test machines.

Processor	Cores	Speed (GHz)	L1 (KB)	L2 (KB)	L3 (MB)
Intel Westmere	24	2.66	12 x 32	12 x 256	2 x 12
AMD Phenom II X6	6	3.3	6 x 64	6 x 512	1 x 6
AMD Interlagos	64	2.2	64 x 16	16 x 2048	8 x 8

6.2. Comparison with Similar Tools

We begin by placing BTO performance results in context by comparing them with several state-of-the-art tools and libraries. Recall that BTO performs loop fusion and array contraction and makes use of data parallelism, but BTO relies on the native C compiler for lower-level optimizations such as loop unrolling and vectorization. In these experiments we used the Intel C Compiler (ICC) [Intel 2012].

We begin by presenting detailed comparisons of the results on an Intel Westmere and then briefly summarize similar results on the AMD Phenom and Interlagos. We compare BTO to the following:

(1) the best general-purpose commercial compilers: ICC and the PGI C Compiler (PGCC) [Portland Group 2012],
(2) the Pluto [Bondhugula et al. 2008] research compiler,
(3) a BLAS-based implementation of the kernels using Intel's Math Kernel Library (MKL) [Intel 2012], and
(4) a hand-tuned implementation.

The input for ICC, PGCC, and Pluto was a straightforward and unoptimized version of the kernels written in C. The hand-tuned implementation was created by an expert in performance tuning who works in the performance library group at Apple, Inc. The expert applied loop fusion, array contraction, and data parallelism. The compiler flags we used with ICC were *"-O3 -mkl -fno-alias,"* and the flags for PGCC were *"-O4 -fast -Mipa=fast -Mconcur -Mvect=fuse -Msafeptr"* (*"-Msafeptr"* not used on Interlagos). Data parallelism is exploited by ICC, PGCC, Pluto, and MKL by using OpenMP [Dagum and Menon 1998]. BTO and the hand-tuned versions use Pthreads [Mueller 1999]. Figure 13 shows the speedup relative to ICC on the y-axis for the linear algebra kernels in Table III. (ICC performance is 1.) On the left are the three vector-vector kernels, and on the right are the six matrix-vector kernels.

Analysis of the General Purpose Commercial Compilers. PGCC tends to do slightly better than ICC, with speedups ranging from 1.1 to 1.5 times faster. Examining the output of PGCC shows that all but GESUMMV and GEMVER were parallelized. However, PGCC's ability to perform loop fusion was mixed; it fused the appropriate loops in AXPYDOT, VADD, and WAXPBY but complained of a "complex flow graph" on the remaining kernels and achieved only limited fusion.

Analysis of the Research Compiler. The Pluto results show speedups ranging from 0.7 to 5.7 times faster than ICC. The worst-performing kernels are AXPYDOT, ATAX, and DGEMVT. These three kernels represent the only cases where Pluto did not introduce data parallelism. For the remaining two vector-vector kernels, VADD and WAXPBY, Pluto created the best-performing result, slightly better than the BTO and hand-tuned versions. Inspection shows that the main difference between Pluto, hand-tuned, and BTO in these cases was the use of OpenMP for Pluto and Pthreads for hand-tuned and BTO. The fusion is otherwise identical and the difference in thread count has little effect. For the matrix-vector operations, if we enable fusion but not parallelization with Pluto's flags, then Pluto matches BTO with respect to fusion. With both fusion and parallelization enabled, however, Pluto sometimes misses fusion and/or parallelization opportunities. For example, BICGK is parallelized but not fused. The GEMVER results depend on the loop ordering in the input file. For GEMVER, Pluto performs either complete fusion with no parallelism or incomplete fusion with parallelism; the latter provides the best performance and is shown in Figure 13.

Analysis of MKL BLAS. The MKL BLAS outperform ICC by factors ranging from 1.4 to 4.2. The calls to BLAS routines prevent loop fusion, so significant speedups, such as those observed in AXPYDOT and GESUMMV, can instead be attributed to parallelism and well-tuned vector implementations of the individual operations. We were unable to determine why the BLAS perform so well for AXPYDOT. Surprisingly, the MKL BLAS DGEMV does not perform as well as Pluto and BTO. Given the lack of fusion potential in this kernel, we speculate that differences in parallelization are the cause.

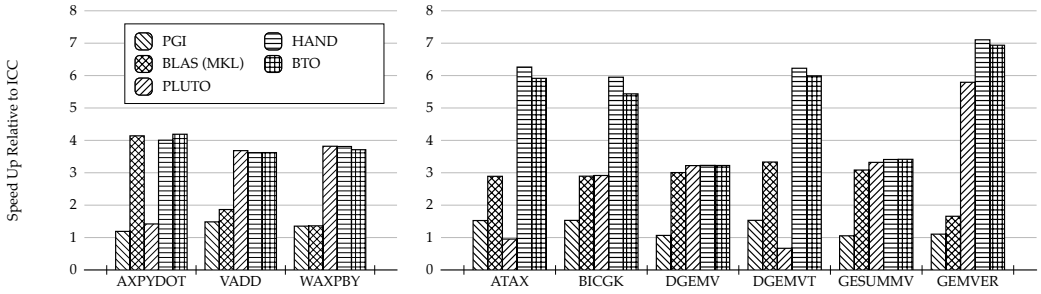

Fig. 13. Performance data for Intel Westmere. Speedups relative to unfused loops compiled with ICC (ICC performance is 1 and not shown). The left three kernels are vector-vector operations, while the right six are matrix-vector operations. In all cases, BTO generates code that is between 16% slower and 39% faster than hand-optimized code and significantly faster than library and compiler-optimized versions.

Analysis of the Hand-Tuned Implementation. The hand-tuned implementation is intended as a sanity check. For the vector-vector operations, the hand-tuned version is within a few percent of the best implementation. Typically the fusion in both the hand-tuned and the best tool-based version are identical, with the primary difference being either the thread count or what appears to be a difference between Pthreads and OpenMP performance. For the matrix-vector operations, the hand-tuned version is the best for all but DGEMV and GESUMMV, where it is equal to the best alternative.

Analysis of the Results for BTO. The BTO performance results show speedups ranging from 3.2 to 6.9 times faster than ICC. For the vector-vector operations, the performance is similar to the hand-tuned version in all cases. Inspection shows that for AXPYDOT, BTO was slightly faster than the hand-tuned version because BTO did not fuse the inner loop, while the hand-tuned version did. BTO performed slightly worse than the hand-tuned version on WAXPBY because of a difference in thread counts. Similarly, BTO's performance on the matrix-vector operations is close but slightly lower than that of the hand-tuned version. BTO fused loops the same way as the hand-tuned implementation for BICGK, GEMVER and DGEMVT, with the only difference being in thread counts. For ATAX, both BTO and the hand-tuned version fused the same and selected the same number of threads, but BTO was slightly slower because of data structure initializations. In the hand-tuned version the initialization occurred in the threads, while in BTO's case it occurred in the main thread. We intend to change this in future versions.

Results on AMD Phenom and Interlagos. The results on AMD Phenom and AMD Interlagos are similar to the Intel results discussed above, as shown in Table V and Table VI, respectively. BTO outperforms BLAS and Pluto versions for every kernel on Phenom, and all but two kernels on Interlagos. The Pluto-generated code for the matrix-vector operations tended to perform worse than that produced for the other methods evaluated. On this computer, achieving full fusion while maintaining parallelism is of great importance. As previously discussed, Pluto tends to achieve fusion or parallelism but struggles with the combination. These results demonstrate the difficulty of portable high-performance code generation even under autotuning scenarios.

Summary. Compared with the best alternative approach for a given kernel, BTO performance ranges from 16% slower to 39% faster. Excluding hand-written comparison points, BTO performs between 14% slower and 229% faster. Pluto, ICC, PGCC, and BLAS all achieve near-best performance for only a few points; however, BTO's performance is the most consistent across kernels and computers. Excluding the hand-optimized results, BTO finds the best version for 7 of 9 kernels on the Intel Westmere, all 9 kernels on the AMD Phenom, and 7 of 9 kernels on the AMD Interlagos. Surprisingly, on the AMD Phenom, BTO surpassed the hand-optimized code for 7 of the 9 kernels and tied for one kernel.

Table V. Performance data for AMD Phenom. BLAS numbers from AMD's ACML. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). Best-performing version in bold.

Kernel	BLAS	Pluto	HAND	BTO
AXPYDOT	0.97	1.81	1.58	**1.86**
VADD	0.84	1.33	1.50	**1.83**
WAXPBY	0.79	1.40	1.68	**1.91**
ATAX	1.27	0.69	**2.92**	**2.92**
BICGK	1.27	0.80	2.80	**2.84**
DGEMV	1.67	0.71	1.85	**1.89**
DGEMVT	1.67	0.71	1.85	**1.89**
GEMVER	1.04	1.61	**2.61**	2.34
GESUMMV	1.63	0.63	1.74	**1.75**

Table VI. Performance data for AMD Interlagos. BLAS numbers from AMD's ACML. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). Best performing version in bold.

Kernel	BLAS	Pluto	HAND	BTO
AXPYDOT	0.82	1.60	1.73	**1.61**
VADD	0.43	1.05	1.14	**1.15**
WAXPBY	0.34	1.06	**1.16**	1.11
ATAX	2.49	0.43	4.09	**4.28**
BICGK	2.35	1.60	3.03	**4.22**
DGEMV	**2.45**	0.89	1.66	2.07
DGEMVT	2.43	0.43	**4.08**	4.03
GEMVER	1.70	2.00	**4.15**	4.05
GESUMMV	**2.36**	0.37	1.65	2.03

6.3. MFGA Compared with Exhaustive Searches

In this section, we show how the performance of BTO's MFGA search algorithm compares with the best version that can be produced by using exhaustive or nearly exhaustive search strategies on Intel Westmere. These strategies require long-running searches that can take days to complete. For the smaller kernels, an exhaustive search is possible. For larger kernels, exhaustive search was not possible, so we instead use a strategy that is exhaustive with respect to each optimization but orthogonal between optimizations. For the largest kernels, GEMVER and DGEMV, even the orthogonal approach took too much time, not completing even after weeks of running. We compared the performance of kernels produced by MFGA as percentage of the exhaustive search for smaller kernels or as a percentage of the orthogonal search for larger kernels such as DGEMVT and GESUMMV. MFGA produces kernel performance within 1-2% of the best performance.

6.4. Evaluation of Search Methods

In this section, we examine the data that led to creating the MFGA search strategy. All the experiments in this section were performed on the Intel Westmere.

6.4.1. Orthogonality of Fusion and Thread Search. The MFGA strategy starts with Max-Fuse, a heuristic specifically for fusion parameters. It ends with a search only over thread count parameters. These additions significantly improve overall performance, as we show below. However, to have confidence in the usefulness of these heuristics, we want to know whether these aspects of the search are orthogonal. For example, if the best fusion combination depended heavily on the number of threads being used, using a fusion-only heuristic would be ineffective. In general when designing a problem-specific search strategy, it helps to understand how heavily interdependent the search pa-

rameters are. We test this in our problem by creating an explicitly orthogonal search and comparing its performance with that of an exhaustive search for feasible kernels.

We define *orthogonal search* as first searching only the fusion parameters, then using only the best candidate, searching every viable thread count. We evaluated the effectiveness and search time of the orthogonal search as compared with an exhaustive search using the smaller kernels: ATAX, AXPY-DOT, BICGK, VADD, and WAXPBY. For all kernels, orthogonal search found the best-performing version while taking 1-8% of the time of exhaustive search, demonstrating that searching the space orthogonally dramatically reduces search time without sacrificing performance. This reduction in search time results in part from the chosen orthogonal ordering. By searching the fusion space first, we often dramatically reduce the number of data-parallel loops and hence the size of the subsequent thread-count search space.

The MFGA strategy is not a strictly orthogonal search: after applying the Max-Fuse heuristic, the genetic algorithm can change fusion and partitioning parameters simultaneously. However, knowing that fusion decisions are usually correct regardless of thread parameters allows us to use heuristics like Max-Fuse that focus on fusion alone.

6.4.2. Fusion Search. Next we focus on fusion strategies. In this section we analyze our choice of using a combination of a genetic algorithm and the Max-Fuse heuristic.

We compare four search strategies on our most challenging kernel, GEMVER. In particular, we test random search, our genetic algorithm without the Max-Fuse heuristic, the Max-Fuse heuristic by itself, and the combination of the Max-Fuse heuristic with the genetic algorithm (MFGA). As described in Section 5, the random search strategy and the genetic algorithm use the same mutation schemes, and thus their comparison shows the benefit of the crossover and selection methods.

Figure 14 shows the performance over time of each of the search methods. (MF is a single point near 3 GFLOPS.) Because the search is stochastic, each of the lines in the chart is the average of two runs. MFGA finds the optimal point in less than 10 minutes on average. Without the MF heuristic, GA alone eventually reaches 90% of MFGA but requires over an hour of search time. The Random search plateaus without ever finding the optimal value. The MF heuristic by itself achieves 40% of MFGA.

We conclude that a combination of GA and MF is the best strategy for the fusion portion of the search.

6.4.3. Thread Search. Using the MFGA heuristic described in the previous section, we explore several thread search strategies, including the *global* thread number and the *exhaustive* strategies discussed in Section 5.4. The baseline test is the MFGA search with number of threads set equal to the number of cores (24 for these experiments), which we refer to as the *const* strategy. Recall also that the *global* strategy starts with MFGA and then searches over a single parameter for all loop nests for the number of threads. Recall that the *exhaustive* search replaces the single thread parameter with the full space of possible thread counts, namely, by considering the number of threads for each loop nest individually.

The results for seven kernels are given in Fig. 15. The top chart shows the final performance of the best version found in each case. Searching over the thread space improves the final performance compared with using a constant number of threads (e.g., equal to the number of cores), with negligible difference in kernel performance between the global thread count (fixed count for all threads) and fully exhaustive approaches (varying thread counts for different operations). The bottom chart in Figure 15 shows the total search cost of the different thread search approaches, demonstrating that global thread search improves scalability without sacrificing performance.

7. RELATED WORK

In this section we describe the relationship between our contributions in this paper and related work in four areas of the literature: loop restructuring compilers, search strategies for autotuning, partitioning matrix computations, and empirical search.

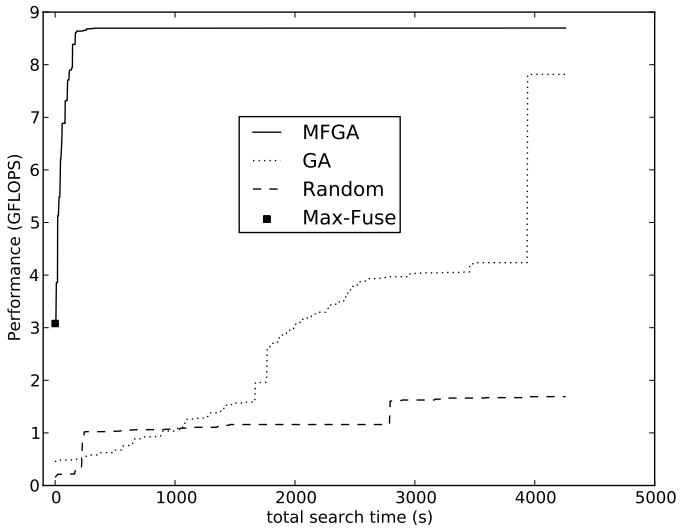

Fig. 14. GEMVER performance over time for different search strategies on Intel Westmere. MFGA finds the best version more quickly and consistently than does either search individually.

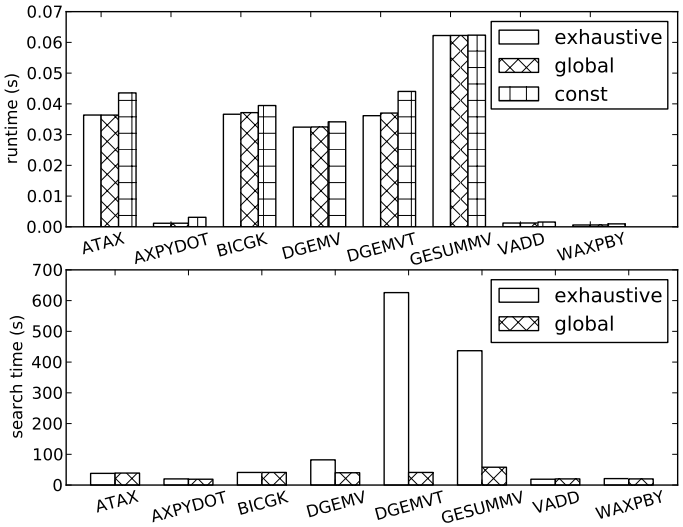

Fig. 15. Best runtime (top) and search time (bottom) for exhaustive and global searches. A constant thread number (e.g., equal to the number of cores) cannot achieve the runtime performance of either global or exhaustive thread search. Searching over a global thread count results in a much shorter search time without significantly worsening kernel performance.

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 as a weighted graph whose nodes are loops and whose edges are labeled with the runtime cost savings resulting from loop fusion. Because the paral-

lelization 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 orthogonal 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 orthogonal search technique in Section 6.4.1.

Bondhugula et al. [2008] employs the heuristic of maximally fusing loops. Loop fusion is generally beneficial, but too much can be detrimental because 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 where possible here.

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 difficult (a very small number of combinations achieve good 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. The genetic algorithm they employ uses a vector-based approach similar to the one described in 4.1, which doesn't translate well to this search problem.

Pueschel et al. [2005] use a genetic algorithm as one search strategy for autotuning discrete signal processing. They translate their search space into trees of rules for breaking discrete transforms into simpler units. They develop a unique crossover and mutation scheme for these ruletrees based on swapping and manipulating subtrees.

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, and so on. 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.

8. CONCLUSIONS AND FUTURE WORK

For many problems in high-performance computing, the best solutions require extensive testing and tuning. We present an empirical autotuning approach for dense matrix algebra that is reliable and scalable. Our tool considers loop fusion, array contraction, and shared memory parallelism.

Our experiments have shown that the BTO autotuning system outperforms standard optimizing compilers and a vendor-optimized BLAS library in most cases, and our results are competitive with hand-tuned code. We also describe how we developed our search strategies and tested the usefulness of each part of the search.

We plan to implement two big expansions of functionality: distributed-memory support through MPI and extension of matrix formats to include triangular, banded, and sparse matrices. These extensions will improve the usefulness of BTO while also providing an important stress test for the scalability of the search algorithms and code generation.

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

ALMASI, G., ROSE, L. D., MOREIRA, J., AND PADUA, D. 2003. Programming for locality and parallelism with hierarchically tiled arrays. In *The 16th International Workshop on Languages and Compilers for Parallel Computing*. College Station, TX, 162–176.

AMARASINGHE, S., CAMPBELL, D., CARLSON, W., CHIEN, A., DALLY, W., ELNOHAZY, E., HALL, M., HARRISON, R., HARROD, W., HILL, K., ET AL. 2009. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*.

ANDERSON, W. K., GROPP, W. D., KAUSHIK, D. K., KEYES, D. E., AND SMITH, B. F. 1999. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (CDROM)*. Supercomputing '99. ACM, Portland, Oregon.

BALAPRAKASH, P., WILD, S., AND HOVLAND, P. 2011. Can search algorithms save large-scale automatic performance tuning? *Procedia CS 4*, 2136–2145.

BELTER, G., JESSUP, E. R., KARLIN, I., AND SIEK, J. G. 2009. Automating the generation of composed linear algebra kernels. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, New York, 1–12.

BELTER, G., SIEK, J. G., KARLIN, I., AND JESSUP, E. R. 2010. Automatic generation of tiled and parallel linear algebra routines. In *Fifth International Workshop on Automatic Performance Tuning (iWAPT 2010)*. Berkeley, CA, 1–15.

BILMES, J., ASANOVIC, K., CHIN, C.-W., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*. ACM Press, New York, 340–347.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. 2002. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software 28,* 2, 135–151.

BONDHUGULA, U., GUNLUK, O., DASH, S., AND RENGANARAYANAN, L. 2010. 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. ACM, New York, 343–352.

BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008. 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)*. Tucson, AZ, 101–113.

CHEN, C., CHAME, J., AND HALL, M. 2008. CHiLL: A framework for composing high-level loop transformations. Tech. Rep. 08-897, Department of Computer Science, University of Southern California.

DAGUM, L. AND MENON, R. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5,* 1, 46–55.

DARTE, A. AND HUARD, G. 2000. Loop shifting for loop parallelization. Tech. Rep. 2000-22, Ecole Normale Superieure de Lyon.

DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 16,* 1, 1–17.

DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software 14,* 1, 1–17.

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw. 27,* 4, 422–455.

HARTONO, A., NORRIS, B., AND SADAYAPPAN, P. 2009. Annotation-based empirical performance tuning using Orio. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE Computer Society, Washington, DC, 1–11. Also available as Preprint ANL/MCS-P1556-1008.

HOWELL, G. W., DEMMEL, J. W., FULTON, C. T., HAMMARLING, S., AND MARMOL, K. 2008. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Trans. Math. Softw. 34*, 14:1–14:33.

INTEL. 2012. Intel Composer. `http://software.intel.com/en-us/articles/intel-compilers`.

KARLIN, I., JESSUP, E., BELTER, G., AND SIEK, J. G. 2011a. Parallel memory prediction for fused linear algebra kernels. *SIGMETRICS Perform. Eval. Rev. 38*, 43–49.

KARLIN, I., JESSUP, E., AND SILKENSEN, E. 2011b. Modeling the memory and performance impacts of loop fusion. *Journal of Computational Science In press*.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Softw. 5,* 3, 308–323.

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

MITCHELL, M. 1998. *An introduction to genetic algorithms*. The MIT Press.

MUELLER, F. 1999. Pthreads library interface. Tech. rep., Florida State University.

PORTLAND GROUP. 2012. Portland group compiler. `http://www.pgroup.com`.

POUCHET, L.-N., BONDHUGULA, U., BASTOUL, C., COHEN, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. 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. IEEE Computer Society, Washington, DC, 1–11.

PUESCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GACIC, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93,* 2, 232–275.

QASEM, A., JIN, G., AND MELLOR-CRUMMEY, J. 2003. Improving performance with integrated program transformations. Tech. Rep. TR03-419, Department of Computer Science, Rice University. October.

QASEM, A., KENNEDY, K., AND MELLOR-CRUMMEY, J. 2006. 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.

SIEK, J. G. 1999. A modern framework for portable high performance numerical linear algebra. M.S. thesis, University of Notre Dame.

SIEK, J. G., KARLIN, I., AND JESSUP, E. R. 2008. Build to order linear algebra kernels. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2008)*. Miami, FL, 1–8.

TIWARI, A., CHEN, C., CHAME, J., HALL, M., AND HOLLINGSWORTH, J. K. 2009. A scalable autotuning framework for compiler optimization. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*. Rome, Italy.

VUDUC, R., DEMMEL, J. W., AND BILMES, J. A. 2004. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications 18,* 1, 65–94.

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, Washington, DC, 1–27.

YI, Q. AND QASEM, A. 2008. 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*. Springer-Verlag, Berlin, 343–355.

YI, Q., SEYMOUR, K., YOU, H., VUDUC, R., AND QUINLAN, D. 2007. POET: Parameterized optimizations for empirical tuning. In *Proceedings of the Parallel and Distributed Processing Symposium, 2007*. IEEE, Long Beach, CA, 1–8.

ZHAO, Y., YI, Q., KENNEDY, K., QUINLAN, D., AND VUDUC, R. 2005. Parameterizing loop fusion for automated empirical tuning. Tech. Rep. UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.