Automating the Generation of Composed Linear Algebra Kernels

Geoffrey Belter
geoffrey.belter@colorado.edu
Dept. of ECEE,
University of Colorado

E. R. Jessup
jessup@cs.colorado.edu
Dept. of Computer Science,
University of Colorado

Ian Karlin
ian.karlin@colorado.edu
Dept. of Computer Science,
University of Colorado

Jeremy G. Siek
jeremy.siek@colorado.edu
Dept. of ECEE,
University of Colorado

ABSTRACT

Memory bandwidth limits the performance of important kernels in many scientific applications. Such applications often use sequences of Basic Linear Algebra Subprograms (BLAS), and highly efficient implementations of those routines enable scientists to achieve high performance at little cost. However, tuning the BLAS in isolation misses opportunities for memory optimization that result from composing multiple subprograms. Because it is not practical to create a library of all BLAS combinations, we have developed a domain-specific compiler that generates them on demand. In this paper, we describe a novel algorithm for compiling linear algebra kernels and searching for the best combination of optimization choices. We also present a new hybrid analytic/empirical method for quickly evaluating the profitability of each optimization. We report experimental results showing speedups of up to 130% relative to the GotoBLAS on an AMD Opteron and up to 137% relative to MKL on an Intel Core 2.

1. INTRODUCTION

The performance of many scientific applications and linear algebra kernels is limited by memory bandwidth [24], a situation that is likely to continue for the foreseeable future [36]. Computer scientists apply tuning techniques to improve data locality and create highly efficient implementations of the Basic Linear Algebra Subprograms (BLAS) [5, 18, 23, 28, 49] and LAPACK [6], enabling scientists to build high-performance software at reduced cost.

While tuned libraries for the level 3 BLAS and LAPACK routines perform at or near machine peak, level 1 and 2 BLAS routines, in which there is less data reuse, achieve only a fraction of peak [27]. However, sequences of level 1 and 2 BLAS routines appear in many scientific applications and these sequences represent further opportunities for tuning. In particular, fusing the loops of successive BLAS routines reduces memory traffic and increases performance [8, 27]. Four such combined routines have been added to the BLAS standard [13]. Many more combined routines are needed, but adding every possible combination to the BLAS standard is not feasible.

To automate the creation of combined routines for level 1 and level 2 operations, the authors have developed a compiler, named Build to Order BLAS (BTO), whose input is a sequence of statements of matrix and vector arithmetic in annotated MATLAB and whose output is a tuned implementation of that sequence in C++. (Generating C or Fortran instead would be a simple change because we use very few features specific to C++.) Our initial prototype applied loop fusion at every opportunity, regardless of profitability [47]. In this paper we present a more refined approach in which the compiler enumerates loop fusion decisions but uses a combination of analytic and empirical techniques to identify the most promising choices.

In particular, the contributions of this paper are as follows:

1. We present an algorithm that compiles linear algebra specifications into loops and enumerates the optimization choices arising from two variants of loop fusion (Section 3).
2. We present a hybrid analytic/empirical performance evaluation method that finds the best combination of optimization decisions in less than two minutes (Section 4).
3. We report experimental results showing speedups of up to 130% relative to the GotoBLAS [22] on an AMD Opteron and up to 137% relative to the Intel Math Kernel Library (MKL) [28] on an Intel Core 2 (Section 5).

The rest of the paper starts with some background and related work in Section 2. It then discusses the above contributions in Sections 3, 4, and 5. The paper ends with conclusions and plans for future work in Section 6.

2. BACKGROUND AND RELATED WORK

We start with a review of loop fusion, which we apply to obtain memory efficiency, then review the literature in domain-specific compilers, auto-tuning libraries, loop restructuring compilers, and methods for analyzing the profitability of optimizations.

Loop Fusion The transformation of one or more loops into a single loop is called loop fusion. If two loops reference the same matrices or vectors, the temporal locality of those references can be improved by performing loop fusion. When applied to memory-bound linear algebra computations, loop fusion can lead to significant runtime speedups [27]. To be a candidate for loop fusion,

```
for $j = 1{:}n$
    $A(:,j) \leftarrow A(:,j) + u_1 v_1(j)$
for $j = 1{:}n$
    $A(:,j) \leftarrow A(:,j) + u_2 v_2(j)$
for $j = 1{:}n$
    $x(j) \leftarrow \beta A^T(j,:) y + z(j)$
for $j = 1{:}n$
    $w \leftarrow w + \alpha A(:,j) x(j)$
```

$\Rightarrow$

```
for $j = 1{:}n$
    $A(:,j) \leftarrow A(:,j) + u_1 v_1(j)$
    $A(:,j) \leftarrow A(:,j) + u_2 v_2(j)$
    $x(j) \leftarrow \beta A^T(j,:) y + z(j)$
    $w \leftarrow w + \alpha A(:,j) x(j)$
```

Figure 1: Loop fusion applied to the GEMVER kernel of the updated BLAS.

the loops must have compatible iterations. For example, they can both iterate over the column dimension or both iterate over the row dimension of a matrix. We illustrate loop fusion in Figure 1, applying it to the GEMVER kernel of the updated BLAS [13]. Assuming that a column of matrix A remains in cache throughout an iteration of the loop, the fused implementation only reads and writes A once from main memory. The benefit comes from not having to make four calls to BLAS routines where each would read in the matrix from memory and two would also write A to memory.

While it is often beneficial to fuse loops, it is not always so. Suppose that there is only enough room in cache for two arrays of length m. Then it is not profitable to fuse the first two scaled vector additions on the right hand side of Figure 1 because doing so brings three arrays (u_1, u_2, and $A(:,j)$) through the memory hierarchy at the same time, causing some of $A(:,j)$ to be evicted. Thus, an optimizing compiler needs to account for the computer architecture as well as matrix order, storage format, and matrix sparsity.

Domain-specific Compilers The MaJIC and FALCON compilers for MATLAB optimize matrix expressions according to algebraic laws [34, 43], but they do not perform loop fusion. Several linear algebra specific compilers take a loop nest specification written over a dense matrix and produce implementations of a sparse matrix operations [10, 38]. Our work differs in that the input is matrix algebra (instead of loops), and we optimize sequences of operations for memory efficiency. The telescoping languages project [29] analyzes MATLAB scripts and optimizes them using procedure specialization, strength reduction, and vectorization. Similarly, Broadway [26] optimizes calls to libraries such as PLAPACK [4]. Our work differs in that we generate the linear algebra routines instead of making calls to pre-existing libraries.

Considerable research has gone into optimizing arrays in languages such as HPF [30], APL [15], and many others. Our compiler differs in that it applies domain-specific knowledge of linear algebra to handle a wide range of matrix storage formats. Domain-specific compilers are growing in popularity with diverse applications in fields such as DSP transforms [39], tensors [3] and optimizing user-defined abstractions [41].

High-Performance Libraries and Auto-tuning Active libraries are libraries that optimize themselves [17]. One example is ATLAS, which uses empirical guided iterative compilation to generate linear algebra operations such as matrix multiplication [49, 50]. We instead optimize sequences of matrix operations and focus on level 2 operations. Blitz++ [48], the Matrix Template Library [46], and Bernoulli [2] use expression templates to perform loop fusion inside individual statements but not across statements. The TaskGraph library uses a dataflow representation to apply loop fusion across statements at run-time [45]. Our approach fuses loops statically and separates the compiler into a graph rewriting engine and linear algebra database for extensibility.

Hierarchically Tiled Arrays (HTA) [11] provide a convenient abstraction for writing tiled algorithms. The Formal Linear Algebra Methods Environment (FLAME) [9, 25] partially automates the generation and implementation of correct linear algebra algorithms. Our compiler is fully automatic.

Loop Restructuring Compilers There is a long tradition of optimizing compilers for general purpose languages that restructure loops to improve data locality. The major approaches in this area are the unimodular [51], affine [32], and polyhedral [14] frameworks. All of them rely on dependence analysis [31] to determine when a transformation is legal. This works well for regular arrays but not sparse matrices. The input to our compiler is in terms of matrix operations instead of loops, so dependence analysis is not needed to determine the legality of our transformations.

Analytic Methods for Profitability Analysis Ferrante et al. [20] estimate the number of cache lines accessed in a perfectly nested loop using a capacity-based model. Our model also relies on capacity but handles arbitrary sequences of loop nests and takes reuse distances into account. Ghosh et al. [21] formulate equations for reuse distances and cache misses that provide a high-degree of accuracy. However, their model is expensive to evaluate. Rivera and Tseng [42] develop a conflict model to predict the profitability of array padding. Our model does not include conflict misses to reduce model evaluation time; we rely on empirical testing for that level of detail. Yotov et al. [53] develop an analytic model for matrix multiplication and show that analytic models can compete with empirical testing. Agakov et al. [1] apply machine learning in predicting the profitability of register allocation choices and instruction scheduling.

Empirical Methods for Profitability Analysis PHiPAC [12] and ATLAS [49, 50] pioneered the use of empirical search to evaluate the profitability of optimizations in the setting of auto-tuned libraries. Zhao et al. [55] use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Yi and Qasem [52] 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. Pouchet et al. [37] use decoupling heuristics and a genetic algorithm in the setting of a polyhedral model to search for high-performance loop nestings. In this paper we use analytic models to quickly discard unprofitable choices then use empirical testing to narrow in on the best.

Hybrid Methods for Profitability Analysis Recently a number of researchers have explored hybrid methods that use a combination of analytic modeling and empirical testing. Chen et al. [16] use analytic methods to select a small number of optimization variants, applying combinations of loop permutation, unrolling, register and loop tiling, copy optimization, and prefetching but not loop fusion. They use empirical testing to select the best variant, typically in three to eight minutes. Epshteyn et al. [19] consider loop tiling decisions in the context of matrix multiplication and use an explanation-based learning algorithm to adapt their analytic model based on empirical results.

Qasem [40] uses pattern-based direct search to find good combinations of loop fusion decisions. Because Qasem is targeting a general purpose compiler, he prefers the scalability of direct search over exhaustive search even though direct search sometimes misses the globally optimal solution. Yotov et al. [54] advocate using analytic methods to make rough, global decisions and then use empirical search locally to fine-tune performance.

In this paper we consider loop fusion, which is particularly challenging because fusion decisions interact with one another. To avoid local minima, we use exhaustive search combined with an-

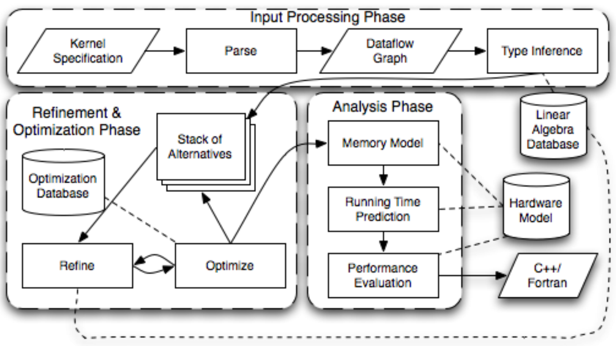


Figure 2: Overview of the compilation process.

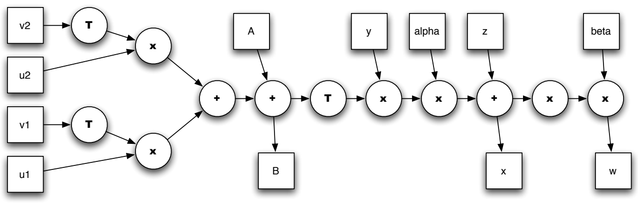


Figure 3: Dataflow graph of the GEMVER kernel.

alytic methods to quickly discard unprofitable combinations, then use empirical testing to select the best. Despite using exhaustive search, we generate high-performance code in less than two minutes (see Section 5). While exhaustive search may not be realistic for general purpose compilers, our results show that it is effective in the more restricted setting of linear algebra kernels.

3. THE COMPILATION FRAMEWORK

The BTO compiler generates a high-performance implementation from a kernel specification. In particular, the input consists of declarations for the input and output parameters followed by a sequence of statements written in MATLAB syntax. An example kernel specification for GEMVER is shown in Listing 1.

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

Listing 1: Compiler input for the GEMVER kernel.

The three major phases of the compilation process are input processing, optimization, and performance analysis (see Figure 2). We discuss input processing and optimization in this section and discuss performance analysis in Section 4.

The compiler parses the kernel specification into a dataflow graph; the graph for GEMVER is shown in Figure 3. Initially, each node in the graph represents an input or output parameter or an operation. Edges between nodes represent the flow of data. After parsing, the compiler performs type inference to determine the dimensionality and best traversal pattern for each node.

3.1 Type Inference

The compiler assigns a type to each node in the graph, where types are given by the following grammar:

$$\begin{array}{llcl} \text{orientations} & O & ::= & C \mid R \\ \text{types} & \tau & ::= & O{<}\tau{>} \mid S. \end{array}$$

The type S is for scalars, such as double-precision floating point numbers. A type of the form $O{<}\tau{>}$ describes a container with element type τ and orientation O. The orientation is either C for column or R for row. Orientation plays two roles: it describes the shape of the node, e.g., $C{<}S{>}$ is a column vector, and it describes the preferred traversal patterns which are chosen to correspond to physical memory layout. For example, $C{<}R{<}S{>}{>}$ describes a matrix whose rows are stored in contiguous memory (as in the language C) whereas $R{<}C{<}S{>}{>}$ describes a matrix whose columns are stored in contiguous memory (as in Fortran). We define the following transpose operator on orientations: $R^T = C, C^T = R$.

The type inference phase assigns a type to each node and at the same time chooses how to implement each operation node. The type inference is data driven, informed by a linear algebra knowledge base of which several rows are shown in Table 1. There is one row for each algorithm that implements a given operation. An algorithm is a valid implementation for an operation node in the graph if 1) the algorithm's operator (e.g., $+$ or $\times$) matches the operation label on the node, 2) the operand types match the types of the operands, and 3) the result type matches the type of the node. If no algorithm can be inferred for an operation node, the compiler reports an error.

The notation we use for result types deserves some explanation. The notation includes the use of the $+$ and $\times$ operators within the type. What this means is that the type inference algorithm is applied recursively to obtain the result type. For example, consider the `add` algorithm whose result type is specified as $O{<}\tau_l + \tau_r{>}$. Suppose the operands of a node labeled with the operation $+$ both have type $C{<}S{>}$. To compute the result type we recursively compute the result type for $S + S$. The only algorithm that applies is `s-add`, so the inner result type is S and therefore the outer result type is $C{<}S{>}$.

Consider the type that would be inferred for the node corresponding to u1 * v1' in the GEMVER kernel of Listing 1. Because matrix A has type $R{<}C{<}S{>}{>}$, the node for u1 * v1' must also have type $R{<}C{<}S{>}{>}$. In this case, the only applicable algorithm is `add` and that algorithm requires that the orientations of the two operands match. The compiler therefore chooses the `outer2` algorithm for u1 * v1' because that version of outer product has a result type that matches $R{<}C{<}S{>}{>}$.

As another example, consider the node corresponding to A' * y in Listing 1. The matrix A' has type $C{<}R{<}S{>}{>}$ and the vector y has type $C{<}S{>}$. Thus, the algorithm `cc-mult` is a match for this multiplication node and the result type is

$$C{<}R{<}S{>} \times C{<}S{>}{>} = C{<}S{>}.$$

Multiple algorithms may be valid choices for the same node. For example, the algorithms `outer1` and `outer2` may sometimes be valid choices for the same node. In such cases, our compiler makes an arbitrary choice. As future work, we plan to evaluate the performance of each option and choose the best.

3.2 Refinement and Optimization

The primary optimization used in the compiler is loop fusion, so as the compiler generates loops, it also chooses which loops to

Algo	Op and Operands	Result Type	Pipe
`add`	$O{<}\tau_l{>} + O{<}\tau_r{>}$	$O{<}\tau_l + \tau_r{>}$	yes
`s-add`	$S + S$	S	no
`trans`	$O{<}\tau{>}^T$	$O^T{<}\tau^T{>}$	yes
`s-mult`	$S \times S$	S	no
`rr-mult`	$R{<}\tau_l{>} \times R{<}\tau_r{>}$	$R{<}R{<}\tau_l{>} \times \tau_r{>}$	yes
`cc-mult`	$C{<}\tau_l{>} \times C{<}\tau_r{>}$	$C{<}\tau_l \times C{<}\tau_r{>}{>}$	yes
`dot`	$R{<}\tau_l{>} \times C{<}\tau_r{>}$	$\sum(\tau_l \times \tau_r)$	no
`outer1`	$C{<}\tau_l{>} \times R{<}\tau_r{>}$	$C{<}\tau_l \times R{<}\tau_r{>}{>}$	yes
`outer2`	$C{<}\tau_l{>} \times R{<}\tau_r{>}$	$R{<}C{<}\tau_l{>} \times \tau_r{>}$	yes
`scale`	$S \times O{<}\tau{>}$	$O{<}S \times \tau{>}$	yes

Table 1: Sample of the linear algebra knowledge base.

fuse. However, because of complex global interactions, the profitability of a particular fusion decision cannot be made in isolation, but rather must be made in the context of all the other decisions. For example, consider a sequence of three loops that are eligible for fusion. The decision to fuse the first two loops cannot be made without considering the third loop because fusing the second and third loops might result in better locality than fusing the first and second. Fusing all three loops, on the other hand, might result in lower performance by increasing register pressure or causing reused data to fall out of cache. To find the optimal set of choices, the BTO compiler quickly explores many combinations of fusion decisions.

The BTO compiler carries out implementation and optimization decisions by applying graph transformations to the dataflow graph. A graph transformation consists of a pattern that specifies to what kind of subgraphs the transformation applies and a rewrite rule that specifies what nodes and edges should be added or removed from the graph. The graph transformations we use come in two varieties, refinements and optimizations. A refinement makes an implementation choice by expanding higher-level operations into lower-level operations, such as expanding a vector operation into a loop over scalar operations. The graph transformations for these refinements are stored in the linear algebra knowledge base. The optimizing transformations, on the other hand, replace subgraphs with functionally-equivalent subgraphs that may provide better performance. The optimizing transformations are stored in a separate knowledge base.

The REFINE and OPTIMIZE algorithms are shown in Figure 4. The REFINE algorithm carries out the implementation choices that were made during type inference. It iterates through the graph, applying the graph transformation associated with the chosen algorithm. When graph transformations add nodes to the graph, they give new nodes larger indices than existing nodes. Thus, as nodes are added to the graph, they too are eligible for further refinement. We discuss particular refinements in Section 3.2.1.

The OPTIMIZE algorithm takes as input the graph produced by the REFINE algorithm and then explores optimization choices. We discuss particular optimizations in Section 3.2.2. The OPTIMIZE algorithm explores choices in a depth-first manner by maintaining a stack of tuples that represent work items. Each tuple contains a version of the dataflow graph and the current node.

If the current node is in the graph, then we push that graph back onto the stack with the node incremented by one to represent the decision not to optimize this node. The algorithm then searches for an applicable optimization and applies the optimization to a copy of the current graph. The transformed graph is also pushed onto the stack, with the node incremented by one. When we have finished making decisions for all the nodes in a graph, then the

```
REFINE($G$)
  node ← 0
  while node < $G$.num_nodes do
    algorithm [node]. apply($G$, node)
    node ← node +1
  return $G$

OPTIMIZE($G$)
  $S$ ← create_stack()
  best_versions ← create_list ()
  $S$.push(⟨$G$, 0⟩)
  while not $S$.is_empty() do
    ⟨$G$, node⟩ ← $S$.pop()
    if node < $G$.num_nodes then
      $S$.push(⟨$G$, node+1⟩)
      for t in optimizations do
        if t .matches($G$, node) then
          $G'$ ← $G$.copy()
          t .apply($G'$, node)
          $S$.push(⟨$G'$, node +1⟩)
    else
      ADD-TO-SEARCH-SPACE($G$, best_versions)
  return best_versions
```

Figure 4: The REFINE and OPTIMIZE algorithms.

algorithm calls ADD-TO-SEARCH-SPACE, shown Figure 7, which decides whether the graph should be added to the best_versions list. ADD-TO-SEARCH-SPACE is described in Section 3.2.3.

The OPTIMIZE algorithm applies one optimization to each node, in the order in which the nodes are created. This ordering explores most combinations but is not exhaustive in some situations. We are currently working on changes to make the algorithm exhaustive.

3.2.1 Graph Refinement

Refinement steps are responsible for reducing high-level matrix and vector operations into loops and scalar operations. Refinement steps introduce new nodes to represent loads, stores, and reduction operations, and they introduce a special kind of subgraph to represent a generalized form of loop over independent operations. Depending on the architecture and the nesting of the loops, a generalized loop can be translated into a parallel loop, a sequential loop, or even a vector instruction.

Generalized loops are created based on the information in Table 1 in the column titled **Result Type**. For example, in the `add` row, the result type is $O{<}\tau_l + \tau_r{>}$. A loop is generated for each container type (one in this case). In the implementation, we associate extra information with each container type, such as its size and storage format, which is needed to generate the loop. The body of the loop is informed by the container's element type, in this case $\tau_l + \tau_r$, which means that the body of the loop adds the elements of the left and right operands. To access elements, we insert nodes to handle load and store operations. Figure 5 shows an example of applying two refinements to a matrix addition according to the `add` algorithm. Each refinement adds two load nodes, one store node, and a subgraph (surrounded in dotted lines) to represent a loop.

3.2.2 Optimization

The two methods of loop fusion in use by the compiler are shown in Figure 6. Figure 6(a) shows the graph transformation for the case when two independent loops access the same data. This transfor-

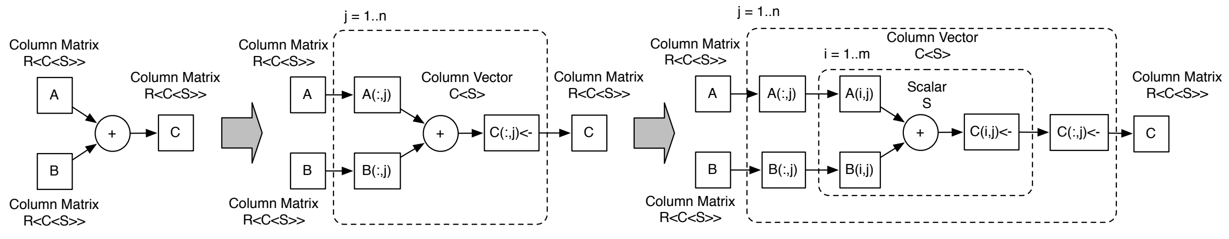
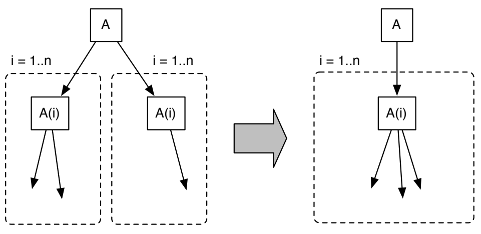
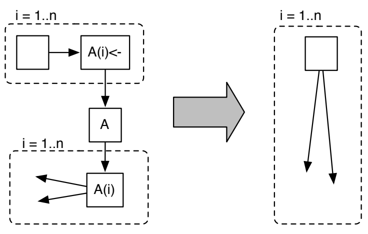


Figure 5: Matrix addition implemented through two refinement steps.



(a) Interleave independent operations



(b) Fuse loops in a pipeline

Figure 6: Loop fusion transformations applied by the compiler.

mation replaces two traversals of the data with a single traversal of the data. Figure 6(b) shows the graph transformation for fusing a set of pipelined loops. This fusion saves two traversals of the intermediate array and removes the need for the intermediate array altogether. (This is known as array contraction [7] and minimizing array materializations [44].) The pipelining optimization is applicable to operation nodes whose **Pipe** column entry is "yes" in the linear algebra knowledge base (see Table 1).

The BTO compiler does not yet perform loop tiling, and therefore does not achieve high performance for matrix multiplication or other level 3 BLAS operations. Achieving high-performance on level 3 operations is not a high priority in our work because the level 3 operations are handled quite well by others [23, 49]; there is little benefit from combining sequences of operations when the operations, like matrix multiplication, have a high ratio of computation to memory access. That said, we plan to add loop tiling because it can achieve modest performance improvements for some level 1 and level 2 operations.

3.2.3 *Search Space Pruning*

With our current set of optimizations, the compiler generates between 1 and 648 optimization combinations for each kernel in our suite of benchmarks. Empirically testing 648 versions is possible but, as we add more optimizations, this number will grow quickly, resulting in a large search space. We therefore include an analytic evaluation step that is accurate enough to remove the need for empirically testing many of the versions in the search space. The algorithm for pruning the search space is shown in Figure 7. We convert the graph to an abstract syntax tree (for convenience) and then pass the tree to the cost estimator described in Section 4. If its cost is lower than the most costly of the best versions so far, we replace the most costly version with the new version. The size to which the set of best versions is allowed to grow is specified by the user.

3.2.4 *Translation to C++*

After the analytic model has pruned the search space, we generate code for the best candidates and empirically evaluate their performance. The BTO compiler generates C++ and then compiles the C++ to machine code using the platform's native compiler. The only non-C feature of C++ that we use is references, so it would be a simple matter to retarget the code generator to C or Fortran.

```
ADD-TO-SEARCH-SPACE($G$, best_versions, $k$)
  $C \leftarrow$ cost(graph_to_tree($G$))
  if best_versions . size () $< k$ then
    best_versions . insert ($\langle G, C\rangle$)
  else
    $\langle G', C'\rangle \leftarrow$ best_versions.max()
    if $C < C'$ then
      best_versions .remove($\langle G', C'\rangle$)
      best_versions . insert ($\langle G, C\rangle$)
```

Figure 7: Pruning based on cost estimation.

The BTO compiler currently does not attempt any lower-level optimizations such as software pipelining or vectorization but relies instead on the native compiler to do so. For computation-bound kernels, or kernels without opportunities for loop fusion, this omission results in sub-optimal performance. In future work, we plan to also include lower-level optimizations in the compiler.

To translate a dataflow graph to C++, the compiler generates one loop for each subgraph and generates the obvious expressions to carry out the scalar operations within each loop. The order in which the loops, and scalar operations within the loops, appear in the output is determined by topological sorting the dataflow graph.

4. ANALYTIC PREDICTION

To efficiently differentiate between optimization choices, we designed and implemented an analytic model. Section 4.1 describes how we predict the amount of data accessed from each memory structure in a machine. We validate these predictions by presenting measurements from hardware performance counters, hand checking the calculations, and instrumenting the generated code to record memory accesses. To estimate overall performance, the analytic model converts data access predictions into execution time as described in Section 4.2. We validate the predicted performance by comparing to the actual performance.

4.1 Predicting Data Accesses

To increase the speed of our model, we restrict it to calculating only the most distinguishing factors. We assume that memory structures are fully associative and have a line size equal to the word size used in the calculation. The model assumes a consecutive access pattern because that is what our compiler produces. The model treats TLBs and caches identically, with the TLB having a size equal to its number of entries times page size. We do not model latency, cache line size, or cache associativity as these factors do not have a significant impact on the relative performance predictions. In general, we assume a "warm cache": if the working set is smaller than cache, we count all memory accesses as cache hits. For larger working sets, we do not count the first miss that brings a datum into cache in situations where that datum will be reused $O(n)$ times or more, where n is the size of a vector or matrix.

Towards defining equations for the number of accesses to each memory structure, we establish several auxiliary notions. Let x range over memory structures in a machine. We write $\mathit{prev}(x)$ for the next smaller memory structure than x. The function prev is undefined (equal to $\perp$) for the smallest memory structure (typically L1 cache). The size of a memory structure x is written $\mathit{size}(x)$.

To represent all memory accesses within a loop L, we use a multiset of addresses $R(L)$. Each address occurs once in $R(L)$ for each time it is accessed in loop L (not including subloops). Let d range over memory addresses. We write $R(L)(d)$ for the number of occurrences of d in $R(L)$. We write $L_1 \leq L_2$ when L_1 is either the same loop as L_2 or nested somewhere within L_2. For a loop L, the working set of the loop, written $WS(L)$, is the number of unique data accesses in the loop (including sub-loops).

$$WS(L) = |\{d \mid 0 < \sum_{L' \leq L} R(L')(d)\}|. \tag{1}$$

The reuse distance of data element d in loop L, written $RD(d, L)$, is the number of unique data accesses between two accesses to d during the execution of loop L.

Now we define the hits to a memory structure x in loop L, written $H(x, L)$, and we define the accesses to x in loop L, written $A(x, L)$. These two multisets are mutually recursive but the base case of A does not rely on H.

$$H(x, L)(d) = \begin{cases} A(x, L)(d) & \text{if } WS(L) \leq \mathit{size}(x) \text{ or} \\ & (RD(d) \leq \mathit{size}(x) \\ & \text{and } R(L)(d) > 1) \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

$$A(x, L) = \begin{cases} R(L) & \text{if } \mathit{prev}(x) = \perp \\ A(\mathit{prev}(x), L) - H(\mathit{prev}(x), L) & \text{otherwise.} \end{cases} \tag{3}$$

The number of accesses to memory structure x in loop L (not counting sub-loops) is $|A(x, L)|$.

4.1.1 Implementation

The model takes as input an abstract syntax tree and a machine. The first step in using the model is to convert the compiler's dataflow graph to an abstract syntax tree. The tree contains three types of nodes: loops, statements, variables. Each node for a loop contains information about the variables on which it iterates, how many times the loop is run and pointers to all variables accessed within the loop. Each statement node contains all the variables that it references. A variable node includes the variable's name, the number of different iterates that are used to access the variable (if the variable is an array) and the iterates' names. The top node in an abstract syntax tree is always one that does not iterate on any variables. It performs a single iteration to allow the combination of multiple independent loops for analysis.

A machine is represented in a structure that contains its name, the number of memory structures it contains, and pointers to those memory structures. Each memory structure contains the amount of data it can hold or address, and a bandwidth, all expressed in bytes.

For each memory structure except the smallest, we perform a depth-first traversal of the abstract syntax tree, evaluating Equation 1 until a loop with a working set smaller than the memory structure is found. Then accesses to the next larger memory structure are computed using Equation 3 beginning with the next higher loop in the tree until the root is reached. Accesses are stored per variable with each instance of a variable having a distinct miss count.

In the implementation of the model, we trade accuracy for speed. We only calculate accesses, hits, and reuse distances at the first element of each array, which forfeits some accuracy as reuse distances sometimes change throughout arrays. This simplification only affects our predictions for data sets near cache boundaries.

4.1.2 Evaluation

To ensure the correctness and accuracy of the implementation, we first compare the accesses computed by the implementation of the analytic model to those resulting from a by-hand calculation of Equation 3. The greatest possible difference between the two is twice the maximum number of variables in a statement (an array only counts once) times the wordsize of a data element. This difference results from the model's not enforcing the ordering of variables within a statement. It is small compared to both cache size and reuse distance. In contrast, the prediction of the points at which matrices and vectors no longer fit into a memory structure are in exact agreement.

Comparing the reuse distances calculated by the model to reuse distances recorded by instrumenting code shows that the model predicts reuse to within the same constant as the constant for memory accesses, described above.

Finally, we compare our model's predictions of memory reads to the actual number of reads measured by hardware performance counters. For each memory structure and loop, we divide the number of accesses by the memory structure's line or page size, written $LS(x)$, to obtain the number of cache lines or page table walks needed for that memory structure, written $LA(x)$:

$$LA(x) = \lceil A(x, L)/LS(x) \rceil. \tag{4}$$

Figure 8 shows the predicted and actual memory miss results for the ATAX kernel. (Misses to a memory structure are equivalent to accesses from the next larger structure.) A separate graph is used for each memory structure (See Table 3.) The predicted misses for the L1 and L2 caches are accurate to within 1% except near cache boundaries. In those cases, conflict misses play an important role and expose the difference between the set associativity of the actual caches and the full associativity assumed by the model.

The TLB is fully associative so it does not experience conflict misses. The predicted misses for the TLB are accurate to within 10% for large matrices on the Intel Core 2 and the AMD Opteron.

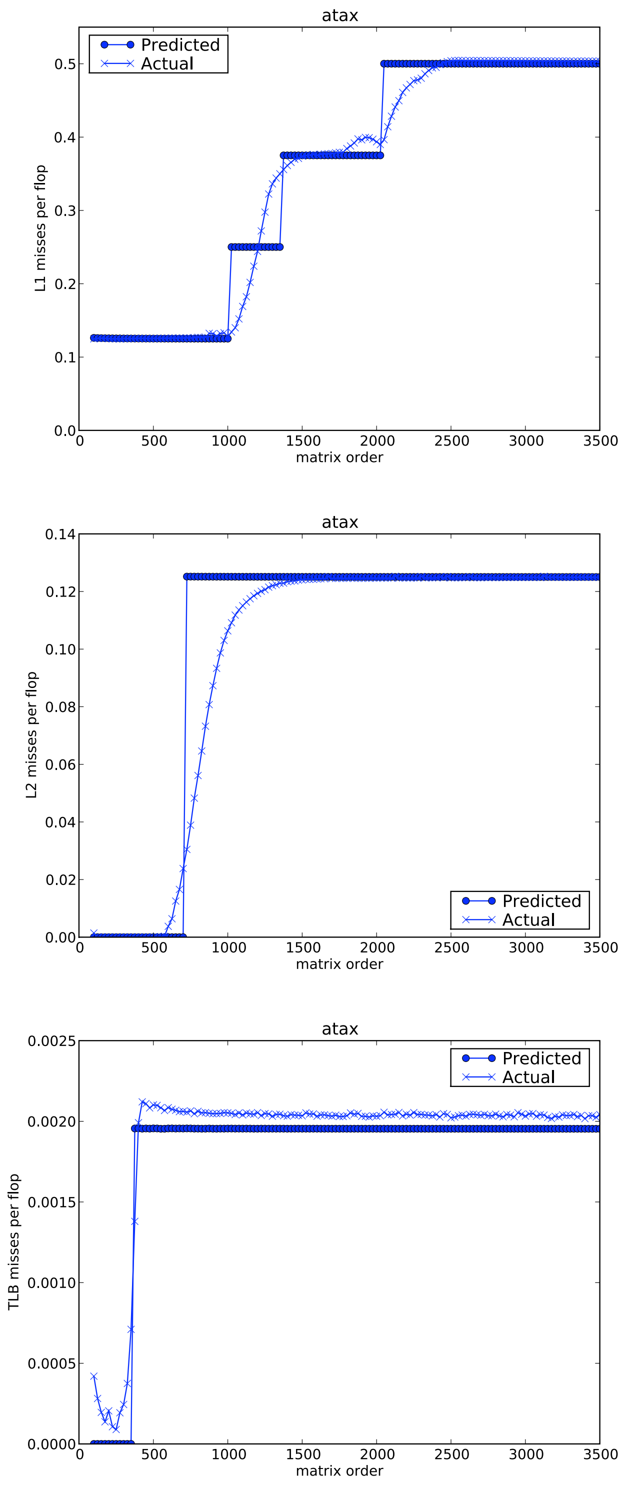


Figure 8: Memory predictions vs. measured values for ATAX on an Intel Core 2.

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

Table 2: Kernel specifications.

Processor	Speed	Mem	L1	L2	TLB
Intel Core 2	2.4 GHz	4 GB	32 KB	4 MB	256
AMD Opteron	2.6 GHz	3 GB	64 KB	1 MB	40/512

Table 3: Specifications of the test machines. For TLB, we list the number of entries.

4.2 Cost Function

To convert memory predictions to a single value for comparing optimization choices we define a cost function. The cost function takes as input the data accesses computed by our model (Section 4.1.1) and the bandwidth, $B(x)$, between each memory structure x and the CPU. We obtain the bandwidths using TRIAD from the Stream benchmark [33].

If L is an inner loop, the *cost* is computed as follows.

$$cost(L) = \max\{A(x, L)/B(x) \mid \text{for all } x\}. \qquad (5)$$

We use the largest value of $A(x, L)/B(x)$ because that represents the bottleneck that limits performance.

We write $child(L)$ for each of the loops directly nested within loop L. If L is an outer loop, the *cost* is computed as follows

$$cost(L) = \max\{A(x, L)/B(x) \mid \text{for all } x\} + \sum_{c \in child(L)} cost(c). \quad (6)$$

When applied recursively to the top of the abstract syntax tree, Equation 6 produces a runtime cost estimate.

In Figures 9 and 10, we compare the runtime estimates from the cost function with actual runtimes of the compiler-generated codes on the Intel Core 2. Each graph in Figure 9 shows the results for the two versions of ATAX produced by the compiler. In each figure, the lines are numbered according to how much fusion was applied to that version of the kernel, with number 1 corresponding to no fusion.

Figure 9 shows that our predictions for large matrices and vectors are accurate: predicted and actual performances are nearly identical. For smaller matrices, our predictions are less accurate for two reasons. First, the model implementation does not currently take the L1 cache into account, so when L1 bandwidth is the bottleneck, the predictions are off. Also, the model does not take the cost of arithmetic or other computations into account, so the model over-predicts performance when computation is the bottleneck. At cache boundaries, our predictions abruptly change while the compiler produced versions follow smooth curves. The abrupt changes happen because the cost model uses the memory model's predictions. Therefore, jumps in performance in Figure 9 correspond to the the jumps in memory predictions in Figure 8.

Figure 10 compares our predictions to actual performance for the 648 versions of GEMVER produced by the BTO compiler, at matrix order 3000. It shows that as the actual performance increases, the predicted performance, for the most part, increases as well. The model over-predicts performance in all cases, though this inaccuracy is less important than the performance difference between versions. When kernels require temporary storage, the first write to a temporary array produces ten times the expected TLB misses. If we replace the TLB predictions with the actual number of TLB misses, the resulting costs are extremely accurate.

The last observation from Figures 9 and 10 is that our cost function always finds the best kernel. All of the findings on the Intel Core 2 hold on the Opteron as well, except for the GEMVER kernel, where the best version is ranked second by the cost function.

5. EXPERIMENTAL EVALUATION

In this section, we present several performance studies. We first describe our test environment and then present the performance of the code generated by our compiler when using empirical testing for the entire search space. We then show that pruning the space using the analytic model drastically reduces compile time without significantly decreasing the performance of the generated code.

5.1 Test Environment and Methodology

We evaluated the compiler on the eleven kernels listed in Table 2 and on two computers, the Intel Core 2 and AMD Opteron listed in Table 3. The selection of kernels tells a complete story regarding the performance of our compiler. First, we selected several kernels with opportunities for loop fusion to demonstrate situations in which our compiler shines. Of these, the BiCGK kernel appears in the biconjugate gradient method, ATAX in the computation of normal equations, GEMVER, DGEMVT, AXPYDOT, and WAXPBY appear in the updated BLAS [13], and VADD and MADD are kernels that we created. Next, we selected GESUMMV from the updated BLAS to show a kernel with some opportunities for fusion, but where fusion does not make big impact. Lastly, we selected the DGEMV kernel from the BLAS, in which there is no opportunity for loop fusion.

We compiled the C++ code generated by our compiler using Intel's icc compiler with -O3 and vectorization options. We ran the experiments on matrix orders ranging from 100 to 10,000 at intervals of 100 and on vector dimensions ranging from 10,000 to 1,000,000 at intervals of 10,000. We compared our compiler's performance to several BLAS implementations: GotoBLAS [22], Netlib [35], Intel's MKL [28], and AMD's ACML [5].

For this comparison, we implement each kernel as a sequence of calls to BLAS. For example, to implement GEMVER, we make two calls to DGER, two calls to DCOPY, and two calls to DGEMV. All tests are run a minimum of five times to provide statistically sound numbers. The values reported in tables and graphs are averages over all trials. We compute standard deviations, but, in most cases, they are less than 2%, so we do not include error bars in the graphs. The only exceptions are for experiments with working sets smaller than cache.

5.2 Performance of Generated Code

The results in this section show that the BTO compiler achieves

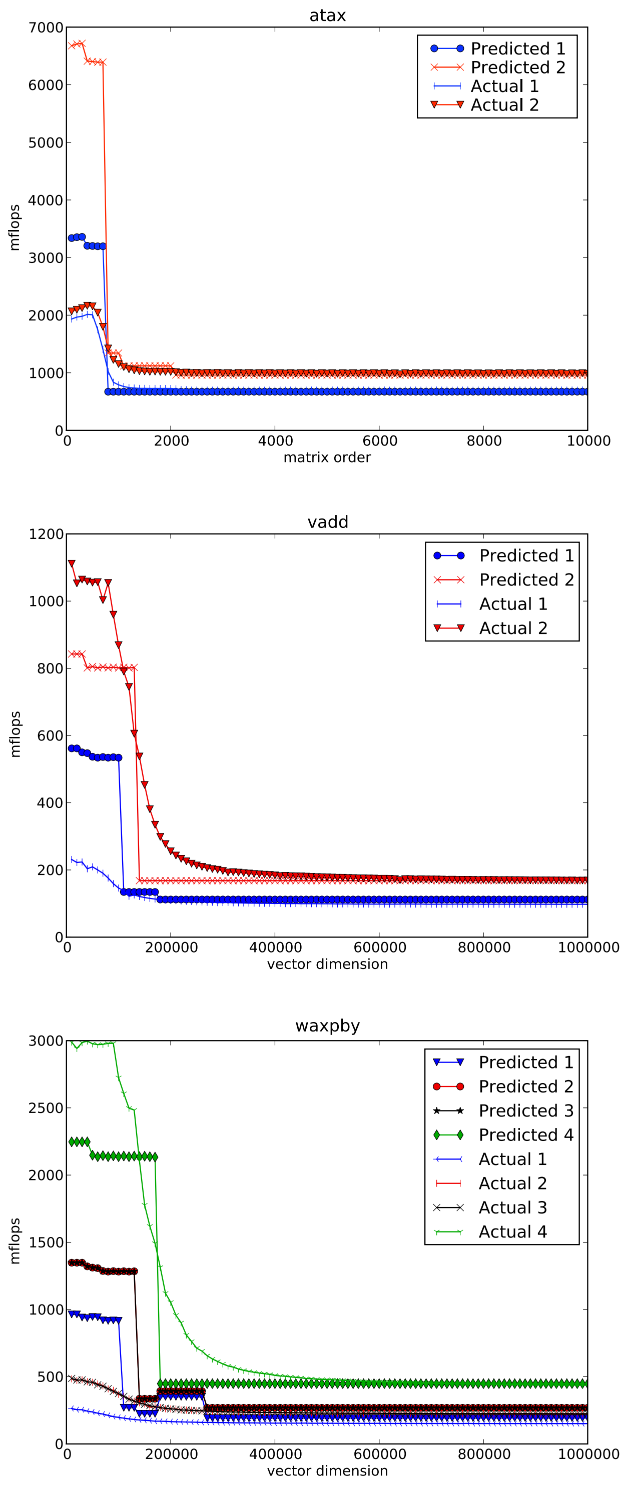


Figure 9: Predicted vs. actual runtime of three kernels on an Intel Core 2.

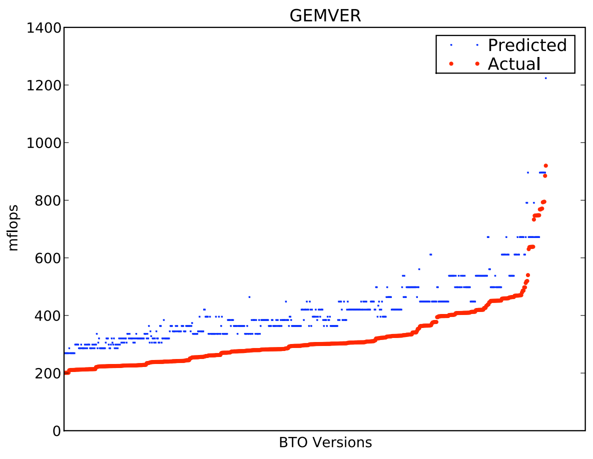


Figure 10: Predicted vs. actual runtime of the 648 versions of GEMVER produced by the BTO compiler on a Core 2.

higher performance than BLAS-based implementations when a kernel is memory bound and that loop fusion across subroutine boundaries reduces memory traffic. In other situations, the BTO compiler usually achieves slightly lower performance because it relies on the native compiler for low-level optimizations and it does not perform other optimizations such as loop tiling.

The graphs in Figure 11 show the performance, on an Intel Core 2, of four BTO-generated kernels compared to BLAS-based implementations. The graphs in Figure 11 show that the BTO versions are 21% to 137% faster for level 2 kernels (ATAX, GEMVER) when the matrix order is 1000 or larger. For level 1 kernels (VADD, WAXPBY), the BTO-generated code is 60% to 88% faster. The performance difference for both the level 1 and 2 kernels is due to increased data reuse in the L2 cache. Table 4 shows L2 cache misses on the Intel Core 2 for BTO-generated code and the BLAS-based implementations. The BTO code has between 35% and 62% the number of L2 misses of the BLAS-based implementations.

Other kernels where loop fusion significantly reduces memory traffic are BiCGK, DGEMVT, MADD, and AXPYDOT (see Table 4). The performance increases for these kernels is summarized in Tables 5 and 6, which compare the results of BTO to the best performing BLAS library for two representative sizes.

There are kernels where the BTO compiler performs loop fusion and does not achieve speedups over BLAS alternatives. For example, the BLAS-based GESUMMV is faster than the BTO-generated code even though our compiler can fuse two DGEMV calls. We do not see memory (Table 4) or performance (Tables 5 and 6) gains for GESUMMV because fusion reduces memory accesses by only $O(n)$, which is dominated by the overall $O(n^2)$ memory accesses, where n is the matrix order. Further, we rely on the native C++ compiler for low level optimizations such as vectorization and instruction scheduling, but the native compilers do not optimize as well as the hand tuning done by Goto and the Intel and AMD teams. We also see slightly more L2 misses in the BTO version of GESUMMV than in the MKL version (Table 4), which we conjecture could be due to tiling in the MKL BLAS. The lack of low level optimizations in the BTO compiler also explains the lower performance for small matrix orders on the ATAX and GEMVER kernels. For sizes less than 1000, when all data fits in cache, memory traffic is not a bottleneck: vectorization and instruction scheduling in the inner loop become the primary factors in performance.

The DGEMV kernel is representative of kernels with no fusion opportunities (beyond the obvious fusion already within DGEMV). In such situations, the reliance of BTO on the native compiler makes an even bigger difference. The the BTO-generated DGEMV is 17% slower than MKL's for large matrices on an Intel Core 2 (Table 5) and 39% slower than the GotoBLAS on an AMD Opteron (Table 6). The reasons for the lower performance are the same as discussed above for GESUMMV.

For kernels such as ATAX and GEMVER, one might expect even larger performance improvements than what we observe. For these kernels, loop fusion saves memory accesses but introduces another inefficiency. Both kernels read a column or row of the matrix from memory and then perform a dot product. Then the column or row is read from cache and a DAXPY is performed. However, while the column or row is read from cache, the memory bus is idle. Because moving data through the memory bus is bottleneck of these kernels, leaving it idle reduces the speedup achieved through fusion. This problem can be solved by a form of vector-scale software pipelining, which we plan to integrate into a future edition of the BTO compiler.

5.3 Efficiency of Hybrid Analysis

This section demonstrates how our hybrid analytic/empirical approach reduces the overall compile-time without decreasing the performance of the generated code. In Table 7, we show results for each algorithm for two different matrix orders or vector dimensions (denoted size in the table) on an Intel Core 2. The first column of the table gives the kernel name followed by the number of differently-optimized versions produced by the compiler. The column headed **Best Mflops** shows the best performance from the compiler's empirical search while **Model's Top 1%** describes the versions with predicted performance within 1% of the memory model's fastest prediction. **Range** shows the performance range as compared to the **Best** column and **Count** shows the number of versions in that top 1%. For most of the kernels, the predicted top 1% achieve within 10% of the actual best performance, demonstrating that the analytic model is indeed picking out the best versions. The only exception is AXPYDOT, where one of the versions from the predicted top 1% only achieves 75% the performance of the best. We are looking into this discrepancy in the analytic model.

When there is little performance change from version to version, as in GESUMMV, the analytic model is not able to reduce the set for empirical testing. The **Range** column shows that the 12 versions vary by a maximum of 8.3%. All 12 versions perform well, so the next edition of our compiler will perform further pruning in such situations and only empirically test a small percentage of the best-performing versions.

Table 8 shows the time taken to compile these kernels. **Model Time** is the time to analyze all versions for two different sizes of data (sizes are the same as shown in Table 5). The two columns under **Empirical Time** show the time taken to empirically test the same two sizes of data using the model's top 1% and to empirically test all versions respectively. The two columns under **Total Time** show the total compile time, both with pruning (and only empirically testing the top 1%) and without pruning (empirically testing all versions). In situations where the model predicts only one version in the top 1%, the compiler does not perform empirical testing. For BiCGK, note that there is only one version in the top 1% for matrix size 1000 but two for 10,000.

The table shows that, by testing all versions predicted to run within 1% of the best predicted version, we find the best actual version. The amount of time it takes to achieve these results is under fifteen seconds for all but GESUMMV. Also, using the model

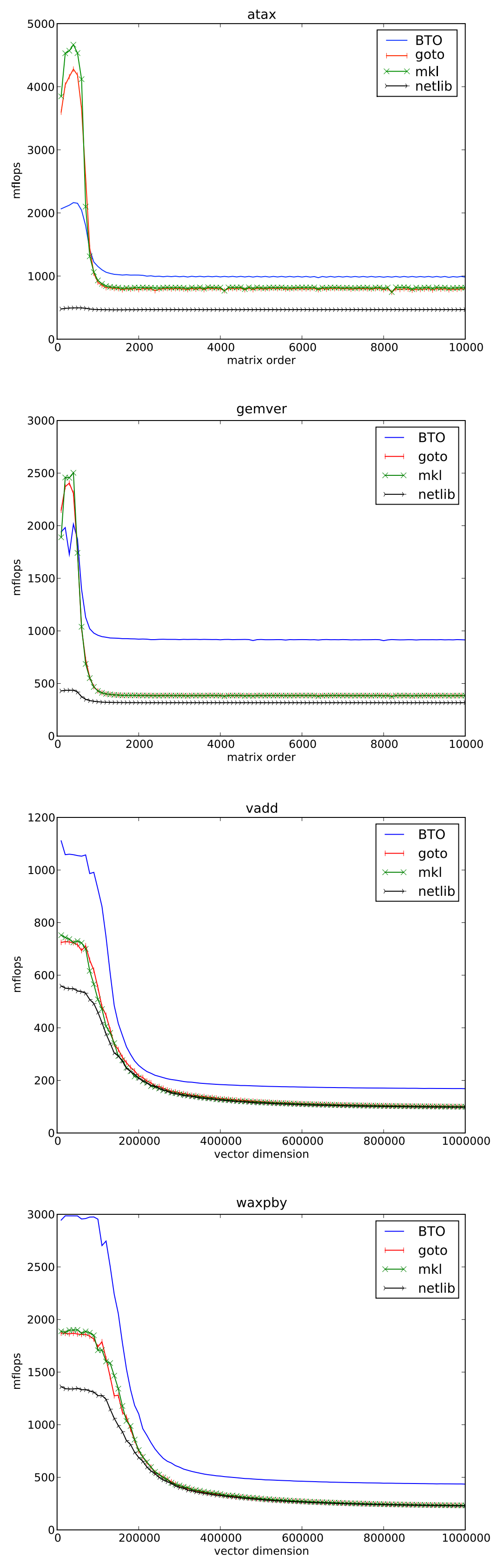


Figure 11: Performance of BTO vs. BLAS on Core 2.

Kernel	Size	BTO	MKL
ATAX	1000	106223.8	211139.2
	10,000	12529786.2	25068564.8
BiCGK	1000	105802.2	212285.0
	10,000	12527997.4	25070363.0
DGEMV	1000	106583.6	106458.8
	10,000	12533659.6	12534375.2
DGEMVT	1000	107256.2	211784.6
	10,000	12539799.2	25068111.8
GEMVER	1000	245457.8	667241.0
	10,000	25086039.2	74461605.0
GESUMMV	1000	249571.2	248366.6
	10,000	25067173.6	25057697.2
MADD	1000	375176.8	491180.8
	10,000	38788905.2	52858139.0
AXPYDOT	10,0000	3746.6	3108.2
	1,000,000	501346.6	742450.2
VADD	100,000	2523.4	3740.2
	1,000,000	501609.2	736532.0
WAXPBY	100,000	213.8	604.8
	1,000,000	374910.6	601202.2

Table 4: L2 misses on Intel Core 2 for BTO and MKL.

	Matrix Size 1000			Matrix Size 10,000		
Kernel	**BLAS**	**BTO**	**Speedup**	**BLAS**	**BTO**	**Speedup**
GEMVER	428	958	124%	385	914	137%
BiCGK	939	1381	47%	820	1232	50%
MADD	75	110	47%	73	107	47%
DGEMVT	932	1174	26%	821	990	21%
ATAX	935	1156	24%	817	990	21%
DGEMV	934	757	-19%	818	680	-17%
GESUMMV	816	760	-7%	817	735	-10%
	Vector Size 100,000			**Vector Size 1,000,000**		
Kernel	**BLAS**	**BTO**	**Speedup**	**BLAS**	**BTO**	**Speedup**
WAXPBY	1809	2898	60%	232	435	88%
VADD	506	925	83%	99	168	70%
AXPYDOT	995	1571	58%	221	328	48%

Table 5: Performance (in Mflops) of BTO vs best BLAS on an Intel Core 2. (MKL was always best.)

	Matrix Size 1000			Matrix Size 10,000		
Kernel	**BLAS**	**BTO**	**Speedup**	**BLAS**	**BTO**	**Speedup**
GEMVER	314 (G)	641	104%	311 (G)	715	130%
MADD	52 (G)	70	35%	72 (A)	102	42%
BiCGK	512 (G)	596	16%	515 (G)	578	12%
DGEMVT	502 (G)	646	29%	512 (G)	545	6%
ATAX	513 (G)	705	37%	516 (G)	542	5%
GESUMMV	553 (G)	515	-7%	737 (G)	555	-25%
DGEMV	528 (G)	336	-36%	545 (G)	330	-39%
	Vector Size 100,000			**Vector Size 1,000,000**		
Kernel	**BLAS**	**BTO**	**Speedup**	**BLAS**	**BTO**	**Speedup**
VADD	95 (A)	107	13%	81 (A)	112	38%
AXPYDOT	167 (A)	204	22%	169 (A)	213	26%
WAXPBY	237 (A)	280	18%	234 (A)	285	22%

Table 6: Performance (in Mflops) of BTO vs best BLAS on an Opteron. G is for the GotoBLAS and A is for ACML.

Kernel (versions)	Size	Best Mflops	Model's Top 1%	
			Range	Count
ATAX (2)	1000	1160	100%	1
	10,000	972	100%	1
BiCGK (3)	1000	1364	96.2-100%	2
	10,000	1232	100%	1
DGEMV (4)	1000	753	99.2-100%	4
	10,000	680	98.6-100%	4
DGEMVT (8)	1000	1174	100%	1
	10,000	990	100%	1
GEMVER (648)	1000	959	100%	1
	6000	918	100%	1
GESUMMV (12)	1000	759	92.6-100%	12
	10,000	735	91.7-100%	12
AXPYDOT (4)	100000	1571	73.2-100%	2
	1,000,000	329	87.5-100%	2
VADD (2)	100000	857	100%	1
	1,000,000	169	100%	1
WAXPBY (4)	100000	2981	100%	1
	1,000,000	437	100%	1

Table 7: Search space pruning results for Intel Core 2. *Best* shows the best possible the compiler is capable of generating without pruning. For the predicted top 1%, *Range* shows the performance range as compared to the *Best* column and *Count* shows the number of versions included in that 1%.

Kernel	Model Time (sec)	Empirical Time (sec)		Total Time (sec)	
		Top 1%	All	With Pruning	Without
ATAX	0.002	-	8.87	0.002	8.87
BiCGK	0.002	0.274	12.08	0.276	12.08
DGEMV	0.004	12.98	12.98	12.98	12.98
DGEMVT	0.011	-	38.29	0.011	38.29
GEMVER	2.256	-	3844.2	2.256	3844.2
GESUMMV	0.016	76.17	76.17	76.19	76.17
AXPYDOT	0.003	0.66	1.43	0.66	1.43
VADD	0.001	-	0.67	0.001	0.67
WAXPBY	0.003	-	1.44	0.003	1.44

Table 8: *Model Time* shows the time to run the model on all versions for two sizes. For *Empirical Time*, *Top* 1% column shows the time to run the top 1% as ranked by the model and *All* shows the time for all versions. Empirical test times include repetition to get a statistically sound performance number. *Total Time* shows the total compile time.

dramatically reduces the amount of time it takes to find the best version while adding an insignificant cost when it fails to differentiate between versions. Tests on the AMD Opteron produced the same findings with one exception: for GEMVER, the model does not find the best version. It does however find a version within 15% of the best for a matrix size of 1000 and within 5% of best for a matrix size of 10,000. Thus, there is room for improvement in our modeling of the AMD Opteron.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present a compilation framework that transforms linear algebra specifications into C++ after enumerating combinations of two loop fusion decisions. We also describe an analytic performance model and show that it can accurately distinguish between significant performance differences in the compiler-produced code. We show that, by using the model, the compiler can focus empirical testing and greatly reduce compilation time without significantly decreasing performance of the generated code.

Our future plans for the compiler include increasing the kinds of optimizations it can perform, including automatic parallelization, loop tiling, software pipelining, and array padding. As these optimizations are added, we plan to update the memory model as needed to predict their effects. Also, we plan to create a symbolic version of the memory model so that we can quickly identify regions (with respect to matrix order and vector size) that have similar memory behavior. This will enable a reduction in the model's execution time and will make possible the use of different techniques (such as loop tiling) for different matrix orders. Additionally, we will explore whether an adaptive memory model would help the compiler to differentiate more finely between routines. A model improved in this manner would account for more hardware and algorithm factors such as cache associativity and varying reuse distance.

Acknowledgments

The work of J. Siek and G. Belter was supported by NSF CAREER Award 0846121 and by a gift from LogicBlox, Inc. The work of E. Jessup and I. Karlin was supported by NSF grant CCF-0830458.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: http://dx.doi.org/10.1109/CGO.2006.37.

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

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

[4] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y.-J. J. Wu. PLAPACK: parallel linear algebra package design overview. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–16, New York, NY, USA, 1997. ACM. ISBN 0-89791-985-8. doi: http://doi.acm.org/10.1145/509593.509622.

[5] AMD. AMD core math library (ACML). http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx, 2009.

[6] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, Washington, DC, USA, 1990. IEEE Computer Society. ISBN O-69791-412-O.

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

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

[9] P. Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, University of Texas at Austin, August 2006.

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

[11] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzaran, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–57, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1122981.

[12] 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.

[13] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/567806.567807.

[14] U. Bondhugula. *Effective Automatic Parallelization and Optimization Using the Polyhedral Model*. PhD thesis, The Ohio State University, August 2008.

[15] T. A. Budd. An APL compiler for a vector processor. *ACM Trans. Program. Lang. Syst.*, 6(3):297–313, 1984. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/579.357248.

[16] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: http://dx.doi.org/10.1109/CGO.2005.10.

[17] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag. ISBN 3-540-41090-2.

[18] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/77626.79170.

[19] A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytical models and empirical search: A hybrid approach to code optimization. In *18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[20] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 328–343, London, UK, 1992. Springer-Verlag. ISBN 3-540-55422-X.

[21] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21:703–746, 1999.

[22] K. Goto. GotoBLAS. `http://www.tacc.utexas.edu/resources/software/#blas`, 2007.

[23] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3), 2008.

[24] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD'99*. Elsevier, 1999. URL `citeseer.ist.psu.edu/gropp99towards.html`.

[25] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4): 422–455, 2001. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/504210.504213.

[26] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, February 2005.

[27] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software*, 34(3):14:1–14:33, 2008.

[28] Intel. Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, 2007.

[29] K. Kennedy, B. Broom, A. Chauhan, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2), February 2005.

[30] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 7–1–7–22, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238851.

[31] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, New York, NY, USA, 1981. ACM Press. ISBN 0-89791-029-X. doi: http://doi.acm.org/10.1145/567532.567555.

[32] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-853-3. doi: http://doi.acm.org/10.1145/263699.263719.

[33] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[34] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 434–443, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-164-X. doi: http://doi.acm.org/10.1145/305138.305230.

[35] Netlib. Netlib repository. http://www.netlib.org/, 2007.

[36] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, USA, third edition, 2003.

[37] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–100, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375594.

[38] W. Pugh and T. Shpeisman. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing: 11th International Workshop, LCPC'98*, volume 1656 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 1998.

[39] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, 2004. ISSN 1094-3420. doi: http://dx.doi.org/10.1177/1094342004041291.

[40] A. Qasem. *Automatic Tuning of Scientific Applications*. PhD thesis, Rice University, July 2007.

[41] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. Annotating user-defined abstractions for optimization. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, 2006.

[42] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[43] L. D. Rose, K. Gallivan, E. Gallopoulos, B. A. Marsolf, and D. A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 269–288, London, UK, 1996. Springer-Verlag. ISBN 3-540-60765-X.

[44] D. J. Rosenkrantz, L. R. Mullin, and H. B. H. III. On materializations of array-valued temporaries. In *Languages and Compilers for Parallel Computing: 13th International Workshop*, August 2000.

[45] F. Russell, M. Mellor, P. Kelly, and O. Beckmann. An active linear algebra library using delayed evaluation and runtime code generation, 2006. to appear.

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

[47] 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)*, April 2008.

[48] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science, pages 223–230. Springer-Verlag, 1998.

[49] 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.

[50] D. B. Whalley. Tuning high performance kernels through empirical compilation. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 89–98, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2380-3. doi: http://dx.doi.org/10.1109/ICPP.2005.77.

[51] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics. ISBN 0-89871-228-9.

[52] 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.

[53] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 2005.

[54] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM. ISBN 1595931678. doi: 10.1145/1088149.1088168. URL `http://portal.acm.org/citation.cfm?id=1088168`.

[55] 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.