Generating Empirically Optimized Numerical Software from MATLAB Prototypes

Elizabeth Jessup,[1] Boyana Norris,[2] Albert Hartono,[3] and Jeremy Siek[4]

[1]University of Colorado at Boulder Dept. of Computer Science, Boulder, CO 80309
jessup@cs.colorado.edu
[2]Argonne National Laboratory Mathematics and Computer Science Division, Argonne, IL 60439
norris@mcs.anl.gov
[3]Ohio State University Dept. of Computer Science and Engineering, Columbus, OH 43210
hartonoa@cse.ohio-state.edu
[4]University of Colorado at Boulder Dept. of Electrical and Computer Engineering, Boulder, CO 80309
jeremy.siek@colorado.edu

September 19, 2008

Abstract

The growing demand for higher levels of detail and accuracy in results means that the size and complexity of scientific computations is increasing at least as fast as the improvements in processor technology. Programming scientific applications is hard, and optimizing them for high performance is even harder. The development of optimized codes requires extensive knowledge, not only of the costs of floating-point arithmetic but also of memory access issues and compiler optimizations. Experiments show that the complexity of this hardware-software system means that performance is difficult to predict fully. Therefore, computational scientists are often forced to choose between investing too much time in tuning code or accepting performance that is significantly lower than the best achievable performance on a given architecture.

In this paper, we describe the first steps toward a fully automated system for the optimization of the matrix algebra kernels that are a foundational part of many scientific applications. To generate highly optimized code from a high-level MATLAB prototype, we will follow a three-step approach. To begin, we have developed a compiler that converts a MATLAB script into simple C code. We then use the source transformation tool PLuTo to optimize that code for coarse-grained parallelism and locality simultaneously. The result is an automatically generated code containing loop transformations and OpenMP annotations for exploiting multicore parallelism. Finally, we employ the empirical performance tuning system Orio to generate many tuned versions of the same operation using different optimization techniques, such as loop unrolling and memory alignment. Orio performs an automated empirical search for selecting the

best among the multiple optimized code variants.

We discuss performance results showing that the code generated by using our system significantly outperforms not only the original simple C code but also code based on source BLAS, ATLAS-optimized BLAS, and Intel MKL routines.

1   Introduction

The development of high-performance numerical codes is challenging because performance is determined by complex interactions between the algorithm, data structure, programming language, compiler, and computer architecture. Scientists seeking high performance are thus required to master advanced concepts in computer science and carry out intricate programming tasks in addition to managing the scientific content of their work. They must either invest substantial time in tuning their software or accept low performance. In either case, the productivity of the scientist degrades.

Historically, the research community has pursued two separate paths toward the goal of making software run at near-peak levels. The first of these builds on research into compilers and their associated technologies. The holy grail of compilation research is to take an arbitrary code as input and produce optimal code as output for a given language and hardware platform. Despite the immense effort that has been poured into that approach, its success has been limited both by practical constraints (users will not tolerate compile times that extend into several days) and by the amount of detailed information the compiler can obtain about the software to be compiled and the hardware on which it is to execute [31, 53, 54]. The second path has been to identify kernel routines that constitute the dominant costs of a wide variety of applications. When such kernels can be identified and agreed upon by the members of the community, programmers with the required level of technical knowledge can concentrate on producing these optimized kernel libraries for architectures of interest. A prime example is the high-performance BLAS libraries produced by a combination of hardware vendors, independent software vendors, and researchers. Developers who write their codes calling these routines can achieve high performance across all supported architectures. The efforts in high-performance library development aptly demonstrate, however, that the research into compiler technologies has not been fully successful. If compilers could take the relatively simple code even for basic tasks such as a dense matrix-matrix product and produce optimal code, this second approach would be unnecessary. Like compiler optimization, the kernel library-oriented approach has significant limitations. Support for a variety of architectures is the most serious problem because

2

the optimizations necessary to achieve near-peak performance are by nature not portable. Lack of portability is particularly problematic because processor designs are constantly changing, and constant updates are nearly impossible to maintain by hand. It is a sad fact of the computing industry that by the time highly-optimized code is available for a given architecture, that architecture is generally well on its way to obsolescence.

We are working on a combination of the two approaches designed to overcome some of their shortcomings. This paper described our three-pronged approach toward the development of an automatically tuned library for matrix algebra computations. In Section 4.1, we describe the compiler we have developed that converts a MATLAB script into simple C code. In Section 4.3, we demonstrate how to employ the empirical performance tuning system Orio in conjunction with the PLuTo tool to generate many tuned versions of the same operation using different optimization techniques, such as loop unrolling and memory alignment. The result is an automatically generated code containing loop transformations and OpenMP annotations for exploiting multicore parallelism. In Section 5, we provide performance results showing that the code generated by using our system significantly outperforms not only the original simple C code but also code based on source BLAS, ATLAS-optimized BLAS, and Intel MKL routines.

2   Background

Existing optimizing MATLAB compilers, such as the MaJIC MATLAB compiler, include limited local optimizations for matrix expressions, but do not perform optimizations such as loop fusion across multiple operations as we do with the tools described in this paper. The telescoping languages project [33, 32, 31] uses techniques such as strength reduction, vectorization, and procedure specialization to optimize MATLAB scripts, but do not generate reusable optimized linear algebra routines as described in this paper.

The most common approach to tuning numerical codes is for an expert to transform the source manually, unrolling loops, blocking for multiple levels of cache, and inserting prefetch instructions. The pitfalls of this approach are well-understood [24]. It requires a significant amount of time and effort. Optimizing code for one particular platform may in fact make it less efficient on other platforms and often makes it complex and hard to understand or maintain. An alternative is the use of tuned libraries of key numerical algorithms (e.g., BLAS [17] and LA-PACK [1] for dense linear algebra, SPARSPAK [23] for sparse linear algebra, and BLITZ++ [47] to support scientific computing in C++, or GPULib by Tech-X, an open source library for accelerating numerical computations on Graphics Processing Units [26], to name just a few).

Such libraries are difficult to develop, and they generally cannot adapt to execution context: as we note in Section 3, a series of calls to library routines can incur high memory costs.

Specialized code generators circumvent the high costs of manual code generation . They include tools for basic dense linear algebra operations (ATLAS [52], PhiPAC [7,9]), sparse linear algebra (OSKI [48]), FFTs [21, 40]), stencil-based operations [30], and tensor computations (TCE [6]) among others. It is often impossible to predict the performance of code precisely on modern computer architectures. Thus, many of these specialized code generators exploit search strategies to identify the best (or nearly best) code for a particular choice of problem parameters and machine. Such autotuning tools are not general but rather each is valuable in one specific problem domain. While not all applications can be written in terms of available libraries, the performance achieved with autotuning tools does convincingly demonstrate the advantage of source-to-source transformation over optimizing compilers.

The best performance today is obtained through approaches like that taken by PERI [38] which involves a labor intensive rewrite of portion of the code by a group of performance tools experts in collaboration with application scientists. Maintaining performance of the code as the application and hardware evolves means that the tuning process may need to be repeated. We propose to investigate ways to capture this performance tuning expertise in a way that can be integrated into the application. We will thus enable performance portability and ensure that the improvements to the application are long-lasting.

A number of source or binary transformation tools for performance-improving optimizations exist. LoopTool [20], developed at Rice University, supports annotation-based loop fusion, unroll-and-jam, skewing and tiling. A new tool, POET [55] also supports a number of loop transformations. POET offers a complex template-based syntax for defining transformations in a language-independent manner. Other recent research efforts whose goal, at least in part, is to enable optimizations of source code to be augmented with performance-related information include the X language [16] (a macro C-like language for annotating C code), the Broadway [37] compiler, and various meta-programming techniques [46,50,34,13]. PLuTo [12,11] is a source-to-source transformation tool for optimizing sequences of nested loops. PLuTo employs a polyhedral model of nested loops, where the dynamic instance (iteration) of each statement is viewed as an integer point in a well-defined space, called the statement's polyhedron. Combined with a characterization of data dependences, this representation allows the construction of mathematically correct complex loop transformations. The transformations target both improved cache locality and parallelism.

3 Composed Operations

Codes based on matrix algebra are generally constructed as a sequence of calls to the Basic Linear Algebra Subprograms (BLAS) [18, 36, 17] and similar sparse matrix libraries [42, 22]. Writing programs in this way promotes readability and maintainability but can be costly in terms of memory efficiency. Specifically, the retrieval of a large order matrix at each routine call can have a profound effect on performance even when highly tuned implementations of the BLAS (e.g [8, 51, 29, 19, 25]) are used.

The authors and other researchers have observed that it can be much more efficient to call a single specialized routine that performs multiple operations than to make successive calls to separate BLAS [2, 10, 44, 3, 4, 15, 28, 27, 49]. Single routines that carry out more than one linear algebra operation are known as composed BLAS. As an example, consider the pair of matrix-vector products $q = Ap$, $s = A^T r$, where A is a matrix and p, q, r, and s are vectors, that comprise the computational bottlenecks of the BiConjugate Gradient Method [5, 43] and of the GEMVER kernel examined in Section 4.1. These two operations can be implemented as a pair of calls to the BLAS routine `GEMV` or they can be rewritten as a composed BLAS consisting of the doubly nested loop encompassing both matrix-vector multiplies. In the former case, the matrix A is accessed twice while, in the latter, is it accessed only once. Our preliminary results indicate that this loop fusion leads to a routine that delivers 90% more megaflops than does a pair of calls to the highly optimized Goto BLAS [25] `GEMV` routines for large matrix orders.

Composed routines are the focus of the work presented here, but much of what we discuss generalizes to a much broader array of computations. Extending our approach will be the subject of future work.

4 Our Approach

To generate highly optimized code from a MATLAB prototype, we will follow a three-step approach, illustrated in Figure 1. To begin, we have developed a compiler that converts a MATLAB script into simple C code. We then (optionally) use the source-to-source automatic parallelization tool PLuTo [12] developed at Ohio State University to optimize for coarse-grained parallelism and locality simultaneously. Using the results of the PLuTo analysis, we insert annotations into the C code, which are then processed by our extensible annotation system Orio to generate many tuned versions of the same operation using different optimization parameters. Orio then performs an empirical search to select the best among the multiple
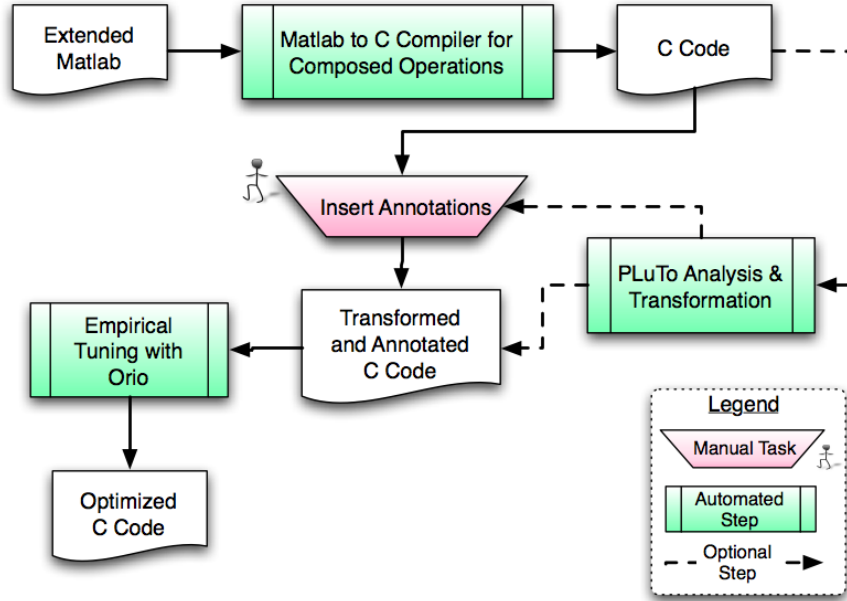
optimized code variants.

Figure 1: Overview of the code generation and tuning process.

4.1 A MATLAB Compiler

We have developed a compiler that translates MATLAB scripts into C code. Figure 2 gives an overview of the compilation process. The MATLAB kernel specification is parsed into a high-level intermediate representation in the form of a dataflow graph. This dataflow graph is then iteratively processed until all of the implementation choices have been made. The compilation process consists of three phases: analysis, lowering, and optimization, which are together iterated until all of the implementation decisions have been made. The graph is then translated into C code.
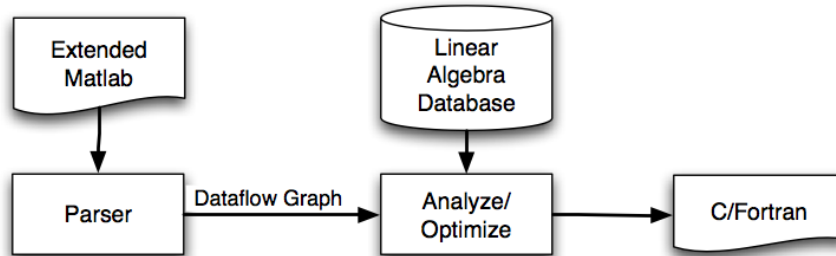

Figure 2: Overview of the MATLAB compiler.

4.1.1 The Dataflow Graph

An example dataflow graph for the GEMVER kernel, defined below, is given in figure 3.

$$A \leftarrow A + u_1 v_1^T + u_2 v_2^T$$
$$x \leftarrow \beta A^T y + z$$
$$w \leftarrow \alpha A x$$

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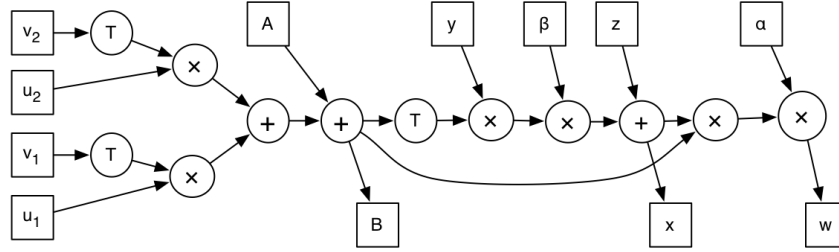

Figure 3: Dataflow graph for the GEMVER kernel.

4.1.2 Analyze the Dataflow Graph

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

7

Going one more step downstream, there is an addition with A, which was annotated in Figure 9 to be column major. At this point it is clear that the outer-products should be computed in column-major order.

We do not hard code this knowledge of basic linear algebra in the compiler algorithm itself, but instead use a data-driven approach and store information regarding how to implement basic linear algebra in a separate database, which we refer to as the *linear algebra database*. This separation allows us to add new matrix formats, operations, and basic linear algebra algorithms without changes to the compiler algorithm.

The analysis algorithm makes implementation choices using the most-constrained first strategy (also called minimum remaining values) from the artificial intelligence literature [41]. The compiler chooses the node with the fewest matching implementations (in the linear algebra database) and assigns a matching implementation to the node. If there is more than one match, the prototype compiler picks the first. Once we add backtracking we will push each alternative onto the stack to be explored later. Once the choice is made, the kind and storage format labels for the operation node itself and for the input nodes are updated with the labels specified in the linear algebra database for the chosen algorithm. The algorithm then goes on to find the next node with the fewest matching implementations and repeats. In this phase, only the name of the algorithm is assigned to the operation node. The details of the implementation are not specified until the refinement phase.

4.2   Refine the Dataflow Graph

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

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

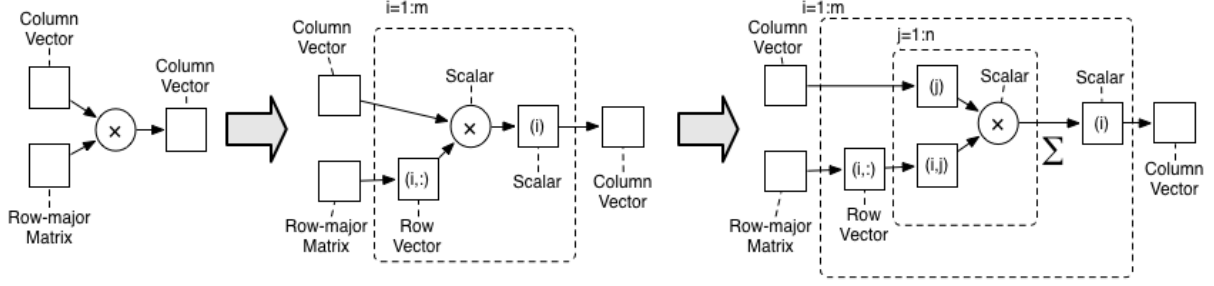Σ indicates that the sum of all of the iterations is computed.

Figure 4: The refinement of a matrix-vector product to a set of inner products and the refinement of inner products to scalar multiplications and additions.

4.2.1 Optimize the Dataflow Graph

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

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

4.2.2 Translate the Dataflow Graph to C

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

4.3 Orio

Orio [39] is an empirical tuning tool that takes as input annotated C code and generates multiple tuned versions of the annotated code. Orio does not parse the application code; rather, it relies on annotations expressed as structured comments, which contain a simplified expression of the computation, as well as directives on what transformations to apply.

Figure 5 illustrates the tuning process using Orio. In the simplest case, Orio can be used to improve code performance by performing source-to-source transformations, such as loop unrolling and memory alignment optimizations. First, the input to Orio is C code, which contains syntactically structured comments that contain both the computation to be tuned, as well as various performance-tuning directives. Orio scans the annotated input code and extracts all annotated regions. Each annotated region is then passed to the code transformation modules for potential optimizations. The code generator then produces the final code incorporating the newly generated optimized C code corresponding to the annotation. The original C code is not parsed (only the annotation comments are processed) and is preserved in the new version. The user has the option of selecting either the original or the tuned code by using a compile-time macro.
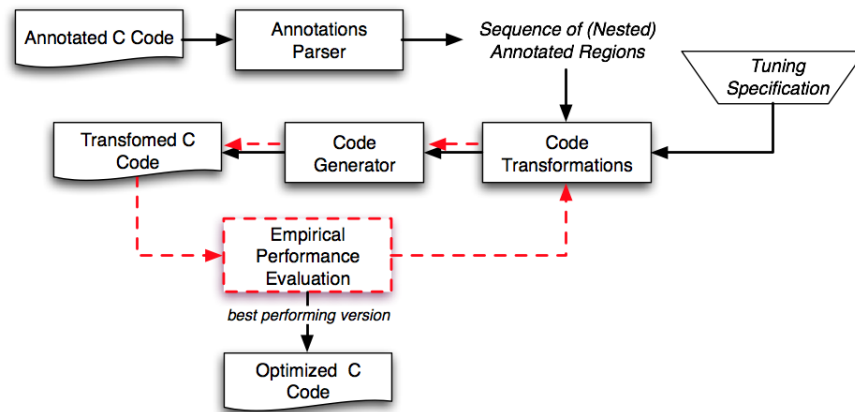

Figure 5: Overview of the Orio empirical tuning process.

Orio can also be used as an automatic performance tuning tool. The system uses its code transformation modules and code generator to generate an optimized code version for each distinct combination of performance parameters. Then the optimized code versions are executed and their performance measured, which is subsequently compared to the performances of other previously tested code variants. After iteratively evaluating all code variants under consideration, the best-performing code is selected as the final output of Orio. The tuning specifications, written by users in the form of annotations (either embedded in the source code or in separate files), are parsed and used by Orio to guide the search and tuning process. These specifications include important information such as the used compilers, the search strategy, the program transformation parameters, the input data sizes, etc.

```
void vadd(int n, double *y, double *x1,
                double *x2, double *x3)
{
    int i;

/*@ begin Align (x1[],x2[],x3[],y[]) @*/
/*@ begin Loop (
  transform Unroll(ufactor=20, parallelize=True)
    for (i=0; i < n; i++)
      y[i] = x1[i] + x2[i] + x3[i];
        ) @*/

    for (i=0; i < n ; i++)
        y[i] = x1[i] + x2[i] + x3[i];

/*@ end @*/
/*@ end @*/

}
```

```
spec align_unroll {
 def build
 {
   arg build_command = 'mpixlc -O3 -qstrict -lm';
 }
 def performance_counter
 {
   arg method = 'basic timer';
   arg repetitions = 10000;
 }
 def performance_params
 {
   param UF[] = range(1,20);
 }
 def input_params
 {
   param SIZE = @THESIZE@;
   decl int n = SIZE;
   decl double x1[n] = random;
   decl double x2[n] = random;
   decl double x3[n] = random;
   decl double y[n] = 0;
 }
 def search
 {
   arg algorithm = 'Exhaustive';
   arg time_limit = 20;
   arg run_command = 'cqsub -n 64 -t 10 -q short ';
   arg num_processes = 64;
 }
}
```

Figure 6: Orio example: annotated C source code (left) and a tuning specification for the Blue Gene/P (right).

We use the VADD operation, $x = w+y+z$, as a simple example of the annotations and tuning specification used by Orio. Figure 6 shows the C code with annotation comments indicating that Orio should perform memory alignment and loop transformations. In addition to simple source transformation annotations in the example above, Orio supports a number of other transformations, including loop unroll and jam, composite optimizations, and architecture-

11

specific directives (e.g., for generation of calls to SIMD intrinsics on the Blue Gene/P). On the right-hand side of Figure 6, a separate tuning specification contains the information required for creating complete executable tests and running them, including variable types, initializations (e.g., random), the search heuristic (in this case, exhaustive search is indicated, other options are random, simplex, and simulated annealing), and execution environment details. When parallel resources are available, Orio executes multiple variants in the same parallel job (the granularity is indicated by the `num_processes` field in the search portion of the tuning specification. At present the tuning specification must be created manually. When Orio is used in conjunction with compiler tools, such as the MATLAB compiler described in this paper, it should be possible to generate most of this tuning specification automatically.

4.4 Search Space Exploration and Evaluation

The set of all possible combinations of implementation and optimization choices is called the search space [35, 45, 14]. Although the dataflow graphs are quite small in our setting (usually less than 100 nodes), it is still important to prune the search space to avoid an exponential compilation time. Our prototype MATLAB compiler already performs some pruning: the optimization rewrite rules only apply under certain circumstances. Orio provides a number of different search heuristics to narrow the search space for near-optimal performance, including random, simplex, and simulated annealing searches. The search algorithm is one of the user-defined values in the Orio tuning specification.

5 Experimental Results

In this section we present performance results for a number of composed BLAS operations. The experiment described in Section 5.1 were performed on Blue Gene/P, while the remaining experiments described in this section were all performed on an Intel Xeon 2.80 GHz Intel(R) E5462 dual quad-core workstation with a 1600Mhz front-side bus and 16GB DDR2 FBDIMM RAM, running Ubuntu 8.04.

For the Xeon experiments, we used the Intel icc compiler (version 10.1) with the `-O3` `-parallel` optimization flags. We generated and tested the following different implementations of each operation:

- C from MATLAB: C code generated by the MATLAB compiler described in Section 4.1.

- BLAS: A version implemented using multiple BLAS calls and linked with the default

BLAS library available on Ubutnu 8.04.

- Intel MKL: The same code as the BLAS version, but linked to the Intel MKL libraries.

- ATLAS: The same code as the BLAS version, but linked to the ATLAS libraries tuned for this machine.

- Orio (Seq.): The C code generated by the MATLAB compiler annotated with sequential-only optimization directives.

- Orio (Par.) : The C code generated by the MATLAB compiler annotated with both sequential and parallel optimization directives. This version was compiled with the `-O3 -openmp` options.

5.1 VADD

The VADD operation is simply the addition of three arrays, storing the result in a fourth: $x = w + y + z$. We optimized the performance for the VADD operation on the Blue Gene/P at Argonne National Laboratory using the IBM XLC compiler version 9.0. We measured the performance for two scenarios: using one core per processor and using all cores. In the single-core case, the codes were compiled with the following options: `-O3 -qstrict -qarch=450d -qtune=450 -qhot -qsmp=noauto`. For the multicore case, the only difference in the compilation was that for the non-Orio versions, the `-qsmp=auto` option was used.

```
VADD
in
  w : vector,
  y : vector,
  z : vector
out
  x : vector
{
  x = w + y + z
}
```
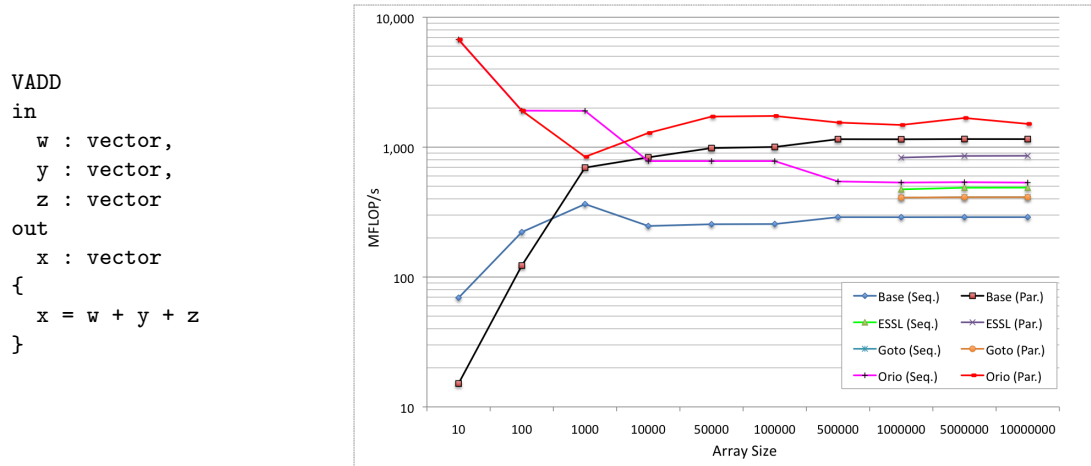

Figure 7: Annotated MATLAB code (left) and MFLOP/s (right) for the VADD operation using different code versions.

The "Base" label designates the simple loop C implementation generated by the Matlab compiler, which was simply compiled with the options mentioned earlier and executed. We tested the performance of the IBM ESSL library and Goto BLAS for the two calls to `daxpy`

13

used in the BLAS version of the code. Finally, we annotated the simple C loop version with memory alignment and loop unrolling directives, and then used Orio to select the best tuned version for the single-core and multicore scenarios. We observe that even for a very simple operation, such as the vector addition above, the compiler alone is unable to obtain the same level of performance as the empirically tuned versions. Furthermore, as expected, the BLAS performance was worse than that of the single loop implementation, however, selecting the correct compiler options was crucial for achieving good performance of the base version.

5.2 ATAX

The extended Matlab code for the ATAX operation, $y = A' * (A * x)$, and the corresponding performance results are shown in Figure 8. The Orio-tuned version that incorporates PLuTo-generated loop fusion optimizations and Orio unrolling and parallelization directives achieves the best performance for most problem sizes, outperforming the Intel MKL version by a factor of 2 to 5.7 and the compiler-optimized C version by a factor of 4 to 7.

```
ATAX
in
  A : row matrix, x : vector
out
  y : vector
{
  y = A' * (A * x)
}
```
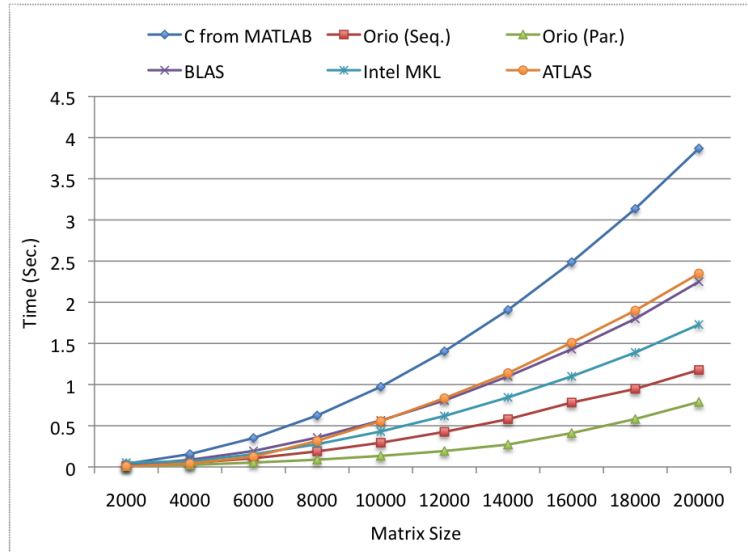

Figure 8: Annotated MATLAB code (left) and wall-clock time in seconds (right) for the ATAX operation using different code versions.

5.3 GEMVER

The extended MATLAB code for the GEMVER operation and the corresponding performance results are shown in Figure 9. As in the ATAX example, the parallel Orio version achieves the best performance, although in this case the sequential Orio version performance almost as well.

14

```
GEMVER
in
  A : column matrix,
  u1 : vector, u2 : vector,
  v1 : vector, v2 : vector,
  a : scalar, b : scalar,
  y : vector, z : vector
out
  B : column matrix,
  x : vector, w : vector
{
  B = A + u1 * v1' + u2 * v2'
  x = b * (B' * y) + z
  w = a * (B * x)
}
```
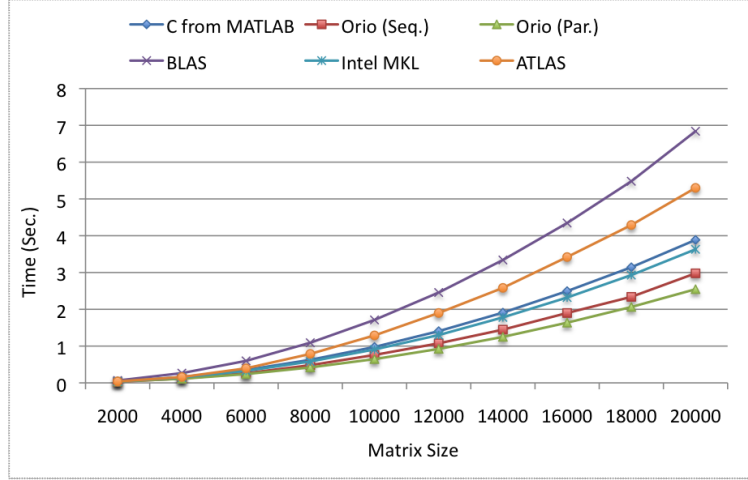

Figure 9: Annotated MATLAB code (left) and wall-clock time in seconds (right) for the GEMVER operation using different code versions.

5.4 GESUMMV

The extended Matlab code for the GESUMMV operation, $y = a * (A * x) + b * (B * x)$, and the corresponding performance results are shown in Figure 10.

```
GESUMMV
in
  A : row matrix,
  B : row matrix,
  x : vector,
  a : scalar,
  b : scalar
out
  y : vector
{
  y = a * (A * x) + b * (B * x)
}
```
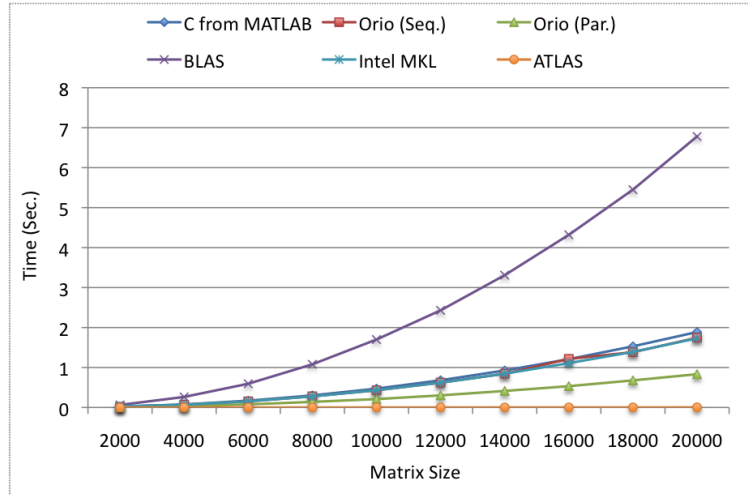

Figure 10: Annotated MATLAB code (left) and wall-clock time in seconds (right) for the GESUMMV operation using different code versions.

5.5 BiCG Kernel

The extended Matlab code for the BiCG Kernel operation, $q = A * p$, $s = A' * r$, and the corresponding performance results are shown in Figure 10. For this operation, the PLuTo analysis did not result in performance improvement, thus we are showing the results obtained

15

only through Orio transformations, which included vectorization, scalar replacement, and unroll and jam. Again the best performance was exhibited by the parallel Orio version, while all the BLAS versions performed worse than the compiler-optimized C loop version.
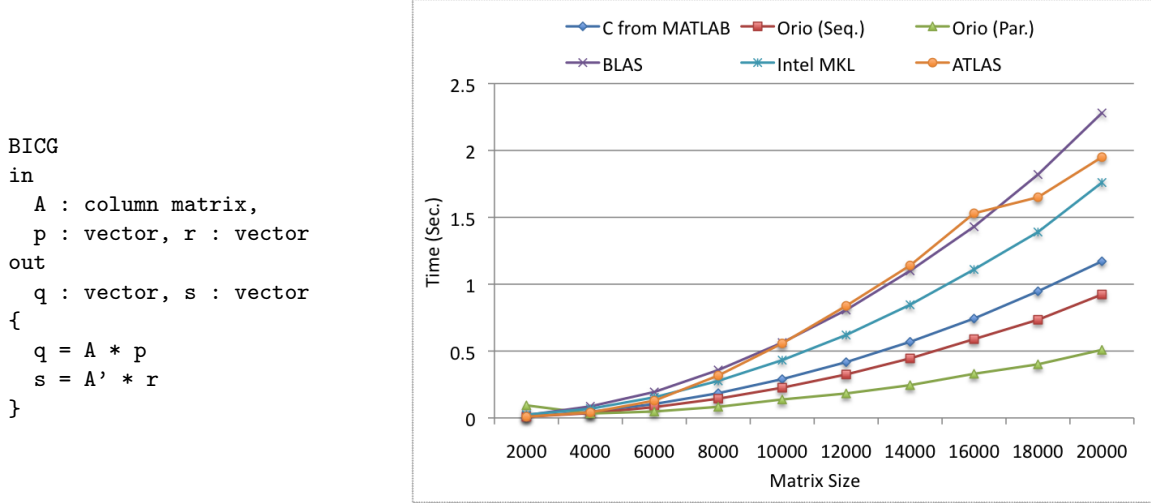
```
BICG
in
  A : column matrix,
  p : vector, r : vector
out
  q : vector, s : vector
{
  q = A * p
  s = A' * r
}
```


Figure 11: Annotated MATLAB code (left) and wall-clock time in seconds (right) for the BiCG kernel operation using different code versions.

6   Conclusions and Future Work

The positive initial results from our three-pronged approach to generating tuned linear algebra routines motivate several future lines of investigation. At present there are a number of manual steps involved in obtaining a compiled tuned linear algebra library from the high-level extended MATLAB descriptions. We will next delve into each manual step and create the software that automates it and in effect serves as the "glue" between the different tools. This would enable application developers to easily produce custom linear algebra libraries tuned for specific architectures. We will also investigate extending the supported functionality of our MATLAB compiler beyond composed BLAS operations, thus benefiting a wider spectrum of applications.

Ultimately, the tuning process should result in more than one version of the computation since different versions are optimal for different problem sizes.

We plan to use a fast analytic cost function to prune out regions of the search space that will not produce competitive implementations. The cost function is based on a model of the computer architecture, but does not necessarily account for all the details. Once the search space is narrowed, we will use empirical testing with representative data sets to determine the exact performance of the alternatives and choose the best one.

References

[1] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, second ed., 1995.

[2] T. ASHBY, A. KENNEDY, AND M. O'BOYLE, *Cross component optimisation in a high level category-based language*, in Euro-Par 2004 Parallel Processing: 10th International Euro-Par Conference, vol. 3149 of Lecture Notes in Computer Science, Pisa, Italy, August 2004, Springer-Verlag GmbH, p. 654.

[3] A. H. BAKER, J. M. DENNIS, AND E. R. JESSUP, *An efficient block variant of GMRES*, SIAM J. Sci. Comput., 27 (2006), pp. 1608–1626.

[4] A. H. BAKER, E. R. JESSUP, AND T. MANTEUFFEL, *A technique for accelerating the convergence of restarted GMRES*, SIAM Journal on Matrix Analysis and Applications, 26 (2005), pp. 962–984.

[5] R. BARRETT, M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, AND H. V. DER VORST, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.

[6] G. BAUMGARTNER, A. AUER, D. E. BERNHOLDT, A. BIBIREATA, V. CHOPPELLA, D. COCIORVA, X. GAO, R. J. HARRISON, S. HIRATA, S. KRISHNAMOORTHY, S. KRISHAN, C.-C. LAM, Q. LU, M. NOOIJEN, R. M. PITZER, J. RAMANUJAM, P. SADAYAPPAN, , AND A. SIBIRYAKOV, *Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models*, in Proceedings of the IEEE, vol. 93, IEEE, Feb 2005, pp. 276–292.

[7] J. BILMES, K. ASANOVIC, C.-W. CHIN, AND J. DEMMEL, *Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology*, in International Conference on Supercomputing, 1997, pp. 340–347.

[8] 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, New York, NY, USA, 1997, ACM Press, pp. 340–347.

[9] J. BILMES, K. ASANOVIĆ, J. DEMMEL, D. LAM, AND C. CHIN, *The PHiPAC WWW home page*, 2006. http://www.icsi.berkeley.edu/~bilmes/phipac.

[10] 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 (2002), pp. 135–151.

[11] U. BONDHUGULA, M. BASKARAN, S. KRISHNAMOORTHY, J. RAMANUJAM, A. ROUNTEV, AND P. SADAYAPPAN, *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*, in International Conference on Compiler Construction (ETAPS CC), Apr. 2008.

[12] U. BONDHUGULA, A. HARTONO, J. RAMANUJAM, AND P. SADAYAPPAN, *PLuTo: A practical and fully automatic polyhedral program optimization system*, in Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ, June 2008.

[13] S. CHIBA, *A metaobject protocol for C++*, in ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Oct 1995, pp. 285–299.

[14] K. D. COOPER, A. GROSUL, T. J. HARVEY, S. REEVES, D. SUBRAMANIAN, L. TORCZON, AND T. WATERMAN, *ACME: Adaptive compilation made efficient*, in LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, NewYork, NY, 2005, ACM, pp. 69–77.

[15] J. M. DENNIS, *Automated memory analysis: improving the design and implementation of iterative algorithms*, PhD thesis, University of Colorado, Boulder, CO, USA, 2005.

[16] S. DONADIO, J. BRODMAN, T. ROEDER, K. YOTOV, D. B. RTHOU, A. COHEN, M. J. GARZARÁN, D. PADUA, AND K. PINGALI, *Language for the compact representation of multiple program versions*, in Proceedings of Languages and Compilers for Parallel Computing (LCPC¡D5¿05), no. 4339 in Lecture Notes in Computer Science, Springer-Verlag, Germany, 2006, pp. 136–151.

[17] J. J. DONGARRA, J. D. CROZ, I. DUFF, AND S. HAMMARLING, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[18] J. J. DONGARRA, J. D. CROZ, S. HAMMARLING, AND R. J. HANSON, *An extended set of Fortran basic linear algebra subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

[19] *Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL.* http://www.ibm.com/systems/p/software/essl.html, 2007.

[20] R. FOWLER, G. JIN, AND J. MELLOR-CRUMMEY, *Increasing temporal locality with skewing and recursive blocking*, in Proceedings of SC01: High-Performance Computing and Networking, Nov. 2001.

[21] M. FRIGO, *A fast Fourier transform compiler*, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), Atlanta, GA, 1999.

[22] A. GEORGE AND E. NG, *A brief description of SPARSPAK Waterloo sparse linear equations package*, SIGNUM Newsl., 16 (1981), pp. 17–20.

[23] ——, *A new release of SPARSPAK: the Waterloo sparse matrix package*, SIGNUM Newsl., 19 (1984), pp. 9–13.

[24] S. GOEDECKER AND A. HOISIE, *Performance optimization of numerically intensive codes*, Software Environments & Tools, 12 (2001).

[25] K. GOTO AND R. VAN DE GEIJN, *High-performance implementation of the level-3 BLAS*, Tech. Rep. TR-2006-23, The University of Texas at Austin, Department of Computer Sciences, 2006.

[26] *Gpulib home page.* http://GPULib.txcorp.com, 2008.

[27] W. D. GROPP, D. K. KAUSHIK, D. E. KEYES, AND B. F. SMITH, *High-performance parallel implicit CFD*, Parallel Computing, 27 (2001), pp. 337–362.

[28] 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 (2008). to appear.

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

[30] S. KAMIL, K. DATTA, S. WILLIAM, L. OLIKER, J. SHALF, AND K. YELICK, *Implicit and explicit optimizations for stencil computations*, in Proceedings of MSPC'06 October 22, 2006, San Jose, CA, USA, 2006.

[31] K. KENNEDY, *Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems*, in Proceedings of IPDPS 2000, May 2000. CD-ROM Proceedings.

[32] K. KENNEDY, B. BROOM, A. CHAUHAN, R. FOWLER, J. GARVIN, C. S KOELBEL, C. MCCOSH, AND J. MELLOR-CRUMMEY, *Telescoping languages: A system for automatic generation of domain languages*, Proceedings of the IEEE, 93 (2005), pp. 387–408.

[33] K. KENNEDY ET AL., *Telescoping languages project description.* `http://telescoping.rice.edu`, 2006.

[34] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Meta-Object Protocol*, MIT Press, Cambridge (MA), 1991.

[35] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. OBoyle, *Combined selection of tile sizes and unroll factors using iterative compilation*, in Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society, 2000.

[36] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–323.

[37] C. Lin and S. Z. Guyer, *Boradway: A compiler for exploiting the domain-specific semantics of software li braries*, Proceedings of the IEEE, 93 (2005), pp. 342–357.

[38] R. Lucas et al., *PERI: Performance Engineering Research Institute*. `http://peri-scidac.org`, 2008.

[39] B. Norris, A. Hartono, and W. Gropp, *Annotations for productivity and performance portability*, in Petascale Computing: Algorithms and Applications, Computational Science, Chapman & Hall / CRC Press, Taylor and Francis Group, 2007, pp. 443–462. Preprint ANL/MCS-P1392-0107.

[40] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. . Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, *SPIRAL: Code generation for DSP transforms*, in Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Platform Adaptation, vol. 93, Feb 2005, pp. 216–231.

[41] S. Russell and P. Norvig, *Articial Intelligence: A Modern Approach*, Prentice Hall, Inc., 2nd ed., 2003.

[42] Y. Saad, *SPARSKIT: A basic tool-kit for sparse matrix computations*, University of Minnesota, Department of Computer Science and Engineering.

[43] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[44] K. S. Stanley, *Execution Time of Symmetric Eigensolvers*, PhD thesis, EECS Department, University of California, Berkeley, 1998.

[45] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, *Compiler optimization-space exploration*, in CGO 03: Proceedings of the international symposium on Code generation and optimization, Washington, DC, USA, 2003, IEEE Computer Society, p. 204215.

[46] T. Veldhuizen, *Expression templates*, C++ Report, 7 (1995), pp. 26–31.

[47] T. L. Veldhuizen, *Blitz++: The library that thinks it is a compiler*, in Modern Software Tools for Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser (Springer-Verlag), Boston, 1997.

[48] R. Vuduc, J. Demmel, and K. Yelick, *OSKI: A library of automatically tuned sparse matrix kernels*, in Proceedings of SciDAC 2005, vol. 16 of Journal of Physics: Conference Series, Institute of Physics Publishing, June 2005, pp. 521–530.

[49] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick, *Memory hierarchy optimizations and performance bounds for sparse $A^T Ax$*, Tech. Rep. UCB/CSD-03-1232, EECS Department, University of California, Berkeley, 2003.

[50] D. Weise and R. Crew, *Programmable syntax macros*, in SIGPLAN Conference on Programming Language Design and Implementation, 1993, pp. 156–165.

[51] 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), Washington, DC, USA, 1998, IEEE Computer Society, pp. 1–27.

[52] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000. `www.netlib.org/lapack/lawns/lawn147.ps`.

[53] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman, London, 1989.

[54] ——, *High Performance Compilers for Parallel Computers*, Addison-Wesley, 1996.

[55] Q. Yi, K. S. and<A0> Haihang You<A0>and Richard Vuduc, and D. Quinlan, *POET: Parameterized optimizations for empirical tuning*, in Proceedings of the Parallel and Distributed Processing Symposium, 2007, IEEE, Mar 2007, pp. 1–8.