Generating Empirically Optimized Composed Matrix Kernels from MATLAB Prototypes

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

[1] Argonne National Laboratory
norris@mcs.anl.gov
[2] Ohio State University
hartonoa@cse.ohio-state.edu
[3] University of Colorado at Boulder
{elizabeth.jessup,jeremy.siek}@colorado.edu

Abstract. The development of optimized codes is time-consuming and requires extensive architecture, compiler, and language expertise, therefore, computational scientists are often forced to choose between investing considerable time in tuning code or accepting lower performance. 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 define a three-step approach. To begin, we have developed a compiler that converts a MATLAB script into simple C code. We then use the polyhedral optimization system PLuTo to optimize that code for coarse-grained parallelism and locality simultaneously. Finally, we annotate the resulting code with performance-tuning directives and use 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 to select the best among the multiple optimized code variants. We discuss performance results on two architectures.

Key words: MATLAB, code generation, empirical performance tuning

1 Introduction

The development of high-performance numerical codes is challenging because performance is determined by complex interactions among the algorithm, data structure, programming language, compiler, and computer architecture. Scientists seeking high performance are thus required to master advanced concepts

* The work of this author was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

** The work of this author was supported by the National Science Foundation under grants no. CCF-0430646 and CCF-830458.

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 scientists degrades.

Historically, the research community has pursued two separate paths toward the goal of making software run at near-peak levels. The first path builds on research into compilers and their associated technologies. One of the main goals of compilation research is to take an arbitrary code as input and produce optimal code as output for a given language and hardware platform. The success of this approach has been limited by a number of factors: (i) optimal mappings between the computation graph and the hardware are expensive (often NP-complete) to compute; (ii) potentially useful information that could aid optimization cannot be represented in general-purpose languages such as C and Fortran; and (iii) user control of compiler optimizations is limited and varies from compiler to compiler, and (iv) apart from differences in execution time, it is difficult to evaluate the effectiveness of different compiler optimizations.

When compilers alone cannot achieve the desired performance, another path to performance optimization is to identify kernel routines that dominate the execution time of a wide variety of applications. An example is the high-performance Basic Linear Algebra Subprograms (BLAS) libraries [1] 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, but are also subject to the limitations of the library (e.g., portability and the types of operations available).

This paper describes a combination of the two approaches designed to overcome some of their shortcomings. We describe our initial efforts toward the development of software infrastructure for *generating* automatically tuned libraries for matrix algebra computations. In Section 2 we briefly discuss relevant prior and current research efforts. In Section 3 we describe our MATLAB-to-C compiler and the empirical performance tuning system Orio and its use in conjunction with the PLuTo tool suite to generate and empirically evaluate many tuned versions of the C code generated by the MATLAB compiler. In Section 4 we provide performance results on two architectures. In Section 5 we conclude with a brief summary.

2 Background

Existing optimizing MATLAB [2] compilers, such as the MaJIC MATLAB compiler [3], 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 [4] uses techniques such as strength reduction, vectorization, and procedure specialization to optimize MATLAB scripts but does 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 [5]: 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, for example, BLAS [6] and LAPACK [7] for dense linear algebra.

Specialized code generators circumvent the high costs of manual code generation. They include tools for basic dense linear algebra operations (ATLAS [8], PhiPAC [9]), and sparse linear algebra (OSKI [10]) among others. While these libraries target a specific kernel, our approach aims at enabling the definition of arbitrary kernels involving dense matrix linear algebra. It is often impossible to predict precisely the performance of code 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. Most existing autotuning tools are not general but focus on a specific domain or algorithm.

A number of source or binary transformation tools for general performance-improving optimizations exist. LoopTool [11], developed at Rice University, supports annotation-based loop fusion, unroll-and-jam, skewing, and tiling. A relatively new tool, POET [12], also supports a number of loop transformations. POET offers a complex template-based syntax for defining transformations in a language-independent manner (but currently only C++ is supported). PLuTo [13] 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 Optimizing Composed BLAS Operations

Codes based on matrix algebra are generally constructed as a sequence of calls to the BLAS and similar sparse matrix libraries [14]. 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 profoundly affect performance even when highly tuned implementations of the BLAS (e.g., [15]) are used.

A much more efficient approach is to call a single, specialized routine that performs multiple operations, rather than to make successive calls to separate BLAS routines (e.g., see [16]). 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 represent the computational bottlenecks of the biconjugate gradient method (BiCG) [17] and of the GEMVER kernel examined in Section 3.1. 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 a doubly nested loop encompassing both matrix-vector products. In the former case, the matrix A is accessed twice, whereas in the latter it is accessed only once. Our preliminary results in prior research and in the work reported in this paper indicate that loop fusion leads to a routine that delivers significantly better performance than does a pair of calls to the best optimized BLAS `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.

To generate highly optimized code from a MATLAB prototype of the composed BLAS operation, we 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 [18]. After generating the C code from the high-level MATLAB prototype, we (optionally) use the source-to-source automatic parallelization tool PLuTo [13] 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.

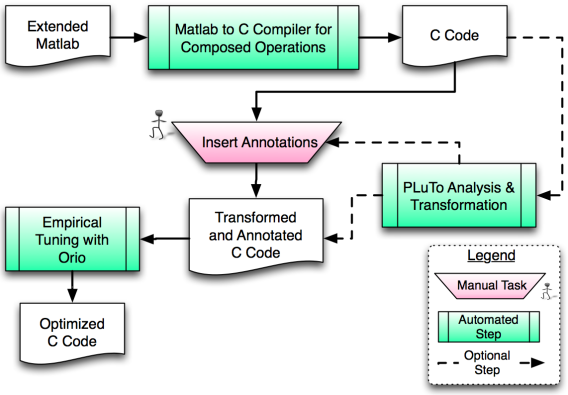


Fig. 1. Code generation and tuning process.

In the remainder of this section we describe each of the tools developed by the authors of this paper, namely, the MATLAB-to-C compiler [18] and the Orio empirical tuning tool [19, 20].

3.1 A MATLAB Compiler

Figure 2 gives an overview of the MATLAB-to-C compilation process [18]. The MATLAB kernel specification is parsed into a high-level intermediate representation in the form of a dataflow graph, in which each node represents a parameter (e.g., a scalar, matrix, or vector variable) of the kernel or an operation.

This dataflow graph is then iteratively processed until all of the implementation choices have been made. The compilation process consists of three phases – analysis, refinement, and optimization – that are together iterated until all of the implementation decisions have been made. The graph is then translated into C code.

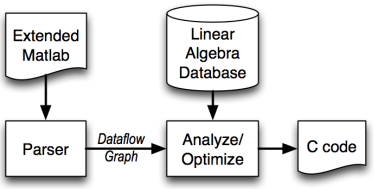


Fig. 2. MATLAB-to-C compiler.

Here we briefly describe the analysis, refinement, and optimization of the dataflow graph; these are discussed in more detail in [18]. During the analysis phase, all types of intermediate nodes are computed and assigned. The algorithm choice and storage format determination are computed simultaneously. Consider, for example, the GEMVER kernel computation: $A \leftarrow A + u_1 v_1^T + u_2 v_2^T;\ x \leftarrow \beta A^T y + z;\ w \leftarrow \alpha A x$. The multiplication of u_1 and v_1^T can be implemented by iterating over rows first or over columns first, depending 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 can choose either option as long as we make the same choice for both outer products.

The information on implementation possibilities for basic linear algebra operations is not hard-coded in the compiler; rather, this data is stored in a database, called 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 known as minimum remaining values) [21]. The compiler chooses the node with the fewest matching implementations (in the linear algebra database) and assigns an algorithm name to the node. If there is more than one match, the prototype compiler picks the first. This process is repeated with all remaining nodes in the 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 that also contains an iteration strategy for traversing the elements of the matrix or vector. In the optimization step, we apply conditional rewrite rules to optimize the dataflow graph, for example merging two subgraphs when they share a common operand. This rule is responsible for fusing the loops of the two matrix-vector products in the GEMVER kernel. The final step performed by the MATLAB compiler when the graph cannot be refined further is the generation of C code. The generator outputs a C loop for each subgraph based on a topological sort of the graph.

3.2 Orio

Orio [19, 20] is an empirical tuning tool that takes annotated C code as input, generates multiple transformed versions of the annotated code, and empirically

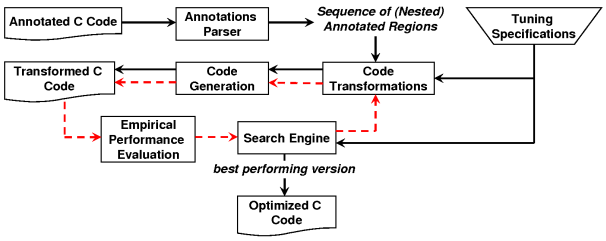


Fig. 3. Overview of the Orio empirical tuning process.

evaluates the performance of the generated codes, ultimately choosing the best-performing version to use in production runs.

Figure 3 illustrates the tuning process implemented in Orio. The input to Orio is C code containing semantic comments that describe both the computation and various performance-tuning directives. Orio first extracts all annotation code regions by parsing the marked-up input code. Each annotated region is then passed to code transformation module and code generator for potential optimizations. Next the transformed C code with various incorporated optimizations corresponding to the specified annotations is produced. The source-to-source transformation system of Orio generates an optimized code version for each distinct combination of performance parameter values. Each generated code variant is then executed and its performance cost evaluated. After iteratively testing all code variants, the best-performing code is selected as the final output of Orio. Because the search space of all possible optimized code variants can be exponentially large, the search engine implements a number of search heuristics (i.e., random, simplex, and simulated annealing) to effectively narrow the search for near-optimal performance. The tuning specifications, written by users in the form of annotations, contains information required for guiding the transformation and tuning process.

Figure 4 shows an annotation example used by Orio to empirically optimize VADD operation on Blue Gene/P. The annotations contain performance hints that instruct Orio to perform memory alignment optimization, loop unrolling, and multicore parallelization (using OpenMP). In addition to these simple optimizations, Orio supports other transformations such as loop blocking, loop permutation, scalar replacement, array copy optimization, and some architecture-dependent optimizations. The right-hand side of Figure 4 shows separate tuning specifications used for building and running executable tests, including performance parameter values, execution environment details, input variable information, and the search algorithm. Orio also supports parallel search when parallel resources are available. In this example, the parallel Orio driver simultaneously executes 64 code variants in the same parallel job. The commands used by Orio to submit a parallel job and to query its status are also specified in the tuning specifications. At present, users must create the tuning specifications manually. When Orio is used in conjunction with compiler tools, such as the MATLAB compiler described in this paper, it should eventually be possible to automatically generate the tuning specifications.

```
void vadd(int n, double *y, double *x1,
          double *x2, double *x3) {
/*@ begin PerfTuning(
  import spec vadd_tune_spec;
) @*/

 register int i;

/*@ begin BGP_Align(y[],x1[],
                    x2[],x3[]) @*/
/*@ begin Loop(
 transform Unroll(ufactor=UF,
                  parallelize=PAR)
 for (i=0; i<=n-1; i++)
   y[i] = x1[i] + x2[i] + x3[i];
) @*/

 for (i=0; i<=n-1; i++)
   y[i] = x1[i] + x2[i] + x3[i];

/*@ end @*/
/*@ end @*/
/*@ end @*/
}
```

```
spec vadd_tune_spec {
 def build {
  arg build_command = 'mpixlc -O3 -qstrict -lm';
  arg batch_command = 'qsub -n 64 -t 10';
  arg status_command = 'qstat';
  arg num_procs = 64;
 }
 def performance_params {
  param UF[] = range(1,32);
  param PAR[] = [True, False];
 }
 def input_params {
  param N = [10,100,1000,10**4,10**5,10**6,10**7];
 }
 def input_vars {
  decl int n = N;
  decl double y[N] = 0;
  decl double x1[N] = random;
  decl double x2[N] = random;
  decl double x3[N] = random;
 }
 def search {
  arg algorithm = 'Exhaustive';
 }
}
```

Fig. 4. Orio example: Annotated C source code (left) and tuning specification excerpt for the Blue Gene/P (right).

4 Experimental Results

We evaluated our approach by running experiments on an Intel Xeon workstation and the Blue Gene/P at Argonne. The Intel machine has dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB) with 2 GB RAM, running Ubuntu 8.04. Intel C compiler (v10.1) was used with `-O3` option (and `-parallel`/`-openmp` for automatic/manual parallelization, respectively). Each node of the Blue Gene/P has four 850 MHz PowerPC 450 processors with a dual floating-point unit and 2 GB total memory per node, running a proprietary operating system. On the Blue Gene/P, we used IBM XLC compiler (v9.0), with `-O3 -qstrict -qarch=450d -qtune=450 -qhot` options (and `-qsmp=auto`/`-qsmp=noauto` for automatic/manual parallelization, respectively).

Table 1 lists the composed BLAS operations used in our experiments, along with their input and output variables. Vectors are typeset in lowercase with an overhead arrow. Scalars and matrices are represented as lowercase and uppercase letters, respectively. A regular uppercase denotes a row matrix, whereas a bold uppercase symbolizes a column matrix. The extended MATLAB expression that corresponds to each operation can be seen in the last column of Table 1.

The performance results (in MFLOP/s) of tuning the VADD operation on the Blue Gene/P are given in Figure 5(a). The "Base" label designates the C implementation generated by the MATLAB compiler. We also tested the performance of an implementation that calls `DAXPY` twice using available BLAS libraries. Finally, we tuned the simple C loop version using Orio, with the performance annotations previously shown in Figure 4. In this experiment, we measured the performance for both the sequential and parallel scenarios. Even for a very sim-

Table 1. Composed BLAS operations used in our experiments.

Name	Input	Output	Operation
VADD	$\overrightarrow{w}, \overrightarrow{y}, \overrightarrow{z}$	$\overrightarrow{x}$	$\overrightarrow{x} = \overrightarrow{w} + \overrightarrow{y} + \overrightarrow{z}$
ATAX	$A, \overrightarrow{x}$	$\overrightarrow{y}$	$\overrightarrow{y} = A' * (A * \overrightarrow{x})$
GEMVER	$\mathbf{A}$,a,b, $\overrightarrow{u_1}$,$\overrightarrow{u_2}$,$\overrightarrow{v_1}$,$\overrightarrow{v_2}$, $\overrightarrow{y}, \overrightarrow{z}$	$\mathbf{B}$, $\overrightarrow{x}, \overrightarrow{w}$	$\mathbf{B} = \mathbf{A} + \overrightarrow{u_1} * \overrightarrow{v_1}' + \overrightarrow{u_2} * \overrightarrow{v_2}'$ $\overrightarrow{x} = b * (\mathbf{B}' * \overrightarrow{y}) + \overrightarrow{z}$ $\overrightarrow{w} = a * (\mathbf{B} * \overrightarrow{x})$
BiCG Kernel	$\mathbf{A}, \overrightarrow{p}, \overrightarrow{r}$	$\overrightarrow{q}, \overrightarrow{s}$	$\overrightarrow{q} = \mathbf{A} * \overrightarrow{p}$ $\overrightarrow{s} = \mathbf{A}' * \overrightarrow{r}$

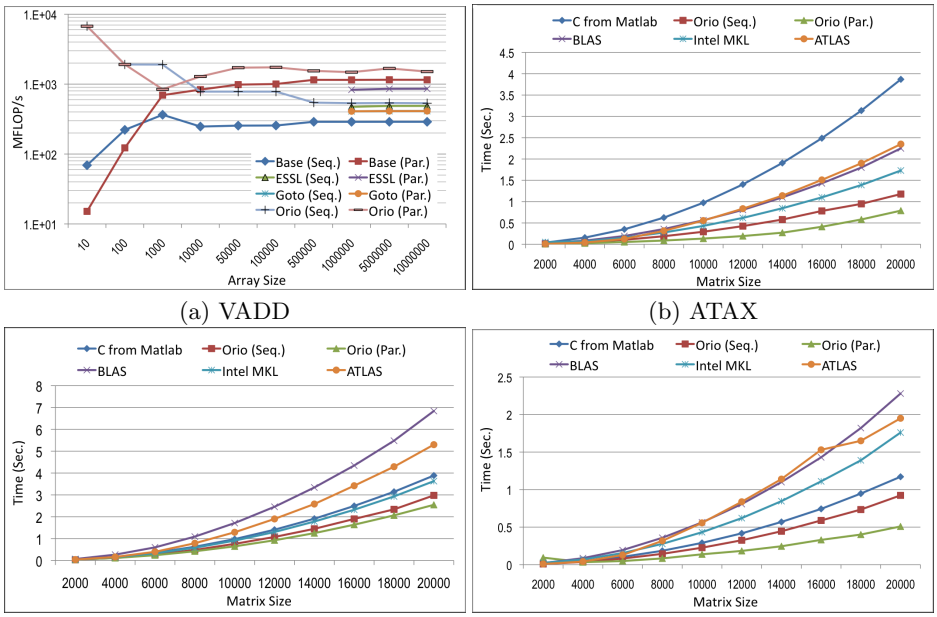
ple operation such as vector addition the compiler alone is unable to obtain the same level of performance as the empirically tuned versions. Furthermore, as expected, the BLAS implementation does not exploit locality and thus performed worse than the single-loop implementation.

The experiments of the remaining operations were performed on the multicore Intel Xeon. Included in these experiments are performance numbers for six code variants: the C code generated by the MATLAB compiler ("C from MATLAB"), three BLAS-based implementations that use Intel MKL, ATLAS, and the default BLAS library on Ubuntu 8.04, and the sequential and parallel code variants tuned by Orio ("Orio (Seq.)" and "Orio (Par.)", respectively).

The Xeon performance results (in seconds) of ATAX are shown in Figure 5(b). The Orio-tuned version that incorporates PLuTo-generated loop fusion optimizations and Orio 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. The optimizations performed by both Orio versions include scalar replacement, vectorization, and loop unroll/jam.

Figure 5(c) shows the performance (in seconds) of the GEMVER operation on the Xeon workstation. Here we used the same PLuTo and Orio optimizations as for the ATAX example. Similarly, the parallel Orio version achieved the best performance, although in this case the sequential Orio version performs almost the same, suggesting that the compiler was not able to parallelize the code very effectively. For this operation, substantial performance differences exist between among the different BLAS versions, with the Intel MKL version achieving performance close to that of the simple compiler code.

The performance (in seconds) for the BiCG kernel operation is shown in Figure 5(d). For this operation, the PLuTo analysis did not result in performance improvement. Thus we are showing the results obtained only through Orio transformations, which included vectorization, scalar replacement, and loop unroll/jam. Again the best performance was achieved by the parallel Orio version, while all the BLAS versions performed worse than the compiler-optimized C loop version.



(a) VADD (b) ATAX

(c) GEMVER (d) BiCG Kernel

Fig. 5. Performance results for several composed BLAS operations.

5 Conclusions

We have described an approach to generating tuned linear algebra libraries from high-level annotated MATLAB code that involves a suite of tools to (1) translate the MATLAB code to C, (2) analyze the resulting loops and identify locality and parallelism-enhancing optimizations using PLuTo, and (3) annotate the resulting C code with syntactic performance directives and use Orio to generate multiple optimized versions and empirically select the one with the best performance. Preliminary results from experiments with several composed BLAS operations show that the optimized code generated by this suite of tools significantly outperforms the versions using tuned BLAS and aggressive compiler optimizations.

The positive initial results from our approach to generating tuned linear algebra routines motivate several future lines of investigation, including closer integration between the tools handling the different steps of the process and more automation at each step.

References

1. Dongarra, J.J., Croz, J.D., Duff, I.S., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. **16** (1990) 1–17
2. MathWorks: MATLAB - The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`

3. Menon, V., Pingali, K.: High-level semantic optimization of numerical codes. In: Proceedings of the 13th International Conference on Supercomputing, New York, ACM Press (1999) 434–443
4. Kennedy, K., et al.: Telescoping languages project description. `http://telescoping.rice.edu` (2006)
5. Goedecker, S., Hoisie, A.: Performance optimization of numerically intensive codes. Software Environments & Tools **12** (2001)
6. Dongarra, J.J., Croz, J.D., Duff, I., Hammarling, S.: A set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Softw. **16**(1) (1990) 1–17
7. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. Second edn. SIAM, Philadelphia, PA (1995)
8. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2) (2001) 3–35
9. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In: International Conference on Supercomputing. (1997) 340–347
10. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005. Volume 16 of Journal of Physics: Conference Series., Institute of Physics Publishing (June 2005) 521–530
11. Fowler, R., Jin, G., Mellor-Crummey, J.: Increasing temporal locality with skewing and recursive blocking. In: Proceedings of SC01: High-Performance Computing and Networking. (November 2001)
12. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: Parameterized optimizations for empirical tuning. In: Proceedings of the Parallel and Distributed Processing Symposium, 2007, IEEE (March 2007) 1–8
13. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: 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)
14. Saad, Y.: SPARSKIT: A basic tool kit for sparse matrix computations. University of Minnesota, Department of Computer Science and Engineering. (1990)
15. Goto, K., van de Geijn, R.: High-performance implementation of the level-3 BLAS. Technical Report TR-2006-23, The University of Texas at Austin, Department of Computer Sciences (2006)
16. Gropp, W.D., Kaushik, D.K., Keyes, D.E., Smith, B.F.: High-performance parallel implicit CFD. Parallel Computing **27** (2001) 337–362
17. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
18. Jessup, E., Karlin, I., , Siek, J.: Build to order linear algebra kernels. In: Proceedings of the IEEE International Symposium on Parallel and Distributed (IPDPS 2008), IEEE (2008) 1–8
19. Norris, B., Hartono, A., Gropp, W.: Annotations for productivity and performance portability. In: Petascale Computing: Algorithms and Applications. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group (2007) 443–462
20. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio. In: Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, Rome, Italy (2009) (to appear).
21. Russell, S., Norvig, P.: Articial Intelligence: A Modern Approach. 2nd edn. Prentice Hall, Inc. (2003)