A Statistical Approach to Reducing an Optimization Search Space

Ian Karlin
University of Colorado at Boulder
Department of Computer Science,
430 UCB, Boulder, CO, 80309
Ian.Karlin@colorado.edu

Erik Silkensen
University of Colorado at Boulder
Department of Computer Science,
430 UCB, Boulder, CO, 80309
Erik.Silkensen@colorado.edu

Elizabeth R. Jessup
University of Colorado at Boulder
Department of Computer Science,
430 UCB, Boulder, CO, 80309
Elizabeth.Jessup@colorado.edu

Geoffrey Belter
University of Colorado at Boulder
Department of Electrical, Computer,
and Energy Engineering 425 UCB,
Boulder, CO, 80309
Geoffrey.Belter@colorado.edu

Thomas Nelson
University of Colorado at Boulder
Department of Computer Science,
430 UCB, Boulder, CO, 80309
Thomas.Nelson@colorado.edu

Pavel Zelinsky
University of Colorado at Boulder
Department of Computer Science,
430 UCB, Boulder, CO, 80309
Pavel.Zelinsky@colorado.edu

Jeremy G. Siek
University of Colorado at Boulder
Department of Electrical, Computer,
and Energy Engineering 425 UCB,
Boulder, CO, 80309
Jeremy.Siek@colorado.edu

ABSTRACT

The performance of many scientific calculations is limited by the cost of data movement inside linear algebra kernels. We developed a compiler called Build to Order (BTO) that tunes kernels to reduce data movement through the memory hierarchy resulting in large speedups. Our compiler accepts annotated MATLAB code and outputs C code with varying levels of loop fusion. In this paper, we present an empirical analysis of the fusion of vector operations with matrix operations. We show using statistical techniques that fusing vectors accessed only once by an operation is not profitable unless doing so enables the fusion of matrix operations. However, vectors accessed multiple times can be profitable to fuse. We discuss the implications of these results to the enumeration and analysis phases of the BTO compiler, and explain how we plan to use them to reduce the runtime of these phases.

Keywords

Performance analysis, Linear algebra, Auto-tuning

1. INTRODUCTION

Linear algebra calculations contribute significantly to the runtime of applications in various scientific fields, such as acoustic scattering, computational fluid dynamics, earthquake engineering, and structural analysis [1, 9, 10, 16]. Data movement often limits the performance of the linear algebra calculations used in these programs [1]. Tuning these linear algebra calculations to reduce data movement through the memory hierarchy often results in a corresponding decrease in routine runtime [8, 13].

One technique frequently used to reduce memory traffic of linear algebra routines is loop fusion, which combines multiple loops of calculations that access the same data into one [12]. Combining loops that access the same data increases the spatial and temporal locality of data accesses. For linear algebra calculations, the result is routines that can perform over 100% faster than comparable unfused routines [18].

While fused routines create significant speedups, producing tuned kernels for all routines that could benefit from fusion is impractical because the number of kernels that would benefit is immense. Also, constantly changing architectures require updating these kernels often since the optimal amount of fusion varies on different architectures.

Auto-tuning is a way to solve the issues of updating linear algebra kernels for ever-changing machines. It has been applied to such other mathematical kernels as fast Fourier transforms [11] and matrix multiplication [4]. Auto-tuned codes often perform more efficiently than hand-optimized versions of the same calculation [19].

There are several ways to automatically create efficient fused linear algebra calculations. Vuduc et al. produce efficient versions of three different fused sparse matrix-vector calculations within the Optimized Sparse Kernel Interface (OSKI) [17]. PLUTO [6] performs fusion using a general purpose compiler within the polyhedral model for C programs. Qasem has devised a method for general purpose Fortran codes [15].

We use a different approach to generate fused kernels. Our compiler Build to Order (BTO) takes in a subset of MATLAB, applies fusion to the kernels and outputs optimized kernels in C [2]. By using MATLAB as input, we are able to generate our own loops that traverse data structures optimally. Also, the higher level input is easier for a user to learn and use. Finally, by using higher level input, we are able to optimize more aggressively as we have a more global view of the calculation being performed.

Our approach differs from other automatic fusion techniques in another meaningful way. The BTO compiler enumerates all legal combinations of loop fusion where the loops being fused share at least one common data element. It then compares the performance of varying amounts of loop fusion using a hybrid analytic/empirical search strategy. Within the hybrid approach, a memory model first analyzes the memory traffic produced by each version of the routine and then predicts a runtime for that version. The best identified versions from the model are then empirically tested with the best selected to be output in C.

One drawback of examining all versions to be fused is that the enumeration of all possible ways to perform a loop fusion calculation is NP-Complete [7]. For serial calculations within our compiler exploring the entire search space has been practical. As we add other optimizations, that need to be enumerated in combination with loop fusion, such as blocking, and the generation of shared memory parallel codes, the search space continually increases. At some point it may become infeasible to explore the entire search space.

In this paper, we show that for certain calculations we can shrink the search space of routines considered while not sacrificing performance of the generated routine. We perform tests on various routines and architectures and show that, in certain cases, the fusion of vector operations with matrix operations does not ever statistically significantly impact routine performance. For every vector operation removed from the search space the number of versions of a routine enumerated is reduced in half.

The rest of this paper is organized as follows. In section 2, we describe the BTO compiler and explain how it converts input MATLAB into efficient C. In section 3, we show how memory predictions often occur in groups. An analysis of the impacts of vector operations within matrix operations is presented in section 4. The analysis includes the circumstances that the fusion of vector operations need not be considered. Section 5 includes how we plan to remove certain vector operations from the enumerated search space in the BTO compiler. Finally Section 6 presents conclusions and future work.

2. BUILD TO ORDER (BTO) COMPILER

The BTO compiler is a system that takes in a subset of annotated MATLAB and produces optimized kernels in C [2]. Its primary purpose is to create memory-efficient linear algebra kernels by reducing data traffic through the memory hierarchy. To limit memory traffic, the compiler uses two forms of loop fusion. Another technique, data partitioning, enables two additional features: cache blocking, which can further reduce data movement and shared memory parallel codes [3]. BTO ensures the creation of efficient routines by exploring the entire search space of potentially profitable parallelization and optimization decisions.

A secondary goal of the project is ease of use. Ease of use is accomplished by automating the creation of efficient linear algebra routines from an accessible high level input.

The BTO compiler works in phases. In the first phase, it parses the input MATLAB and generates a data-flow graph of that input with loops represented. Next, it performs the refinement phase where high level matrix and vector operations are turned into loops and scalars. During the refinement phase a data partitioning algorithm creates loops that are used to create shared memory parallel code or cache blocks. Data partitioning is applied to a single operation in a calculation and then propagated to other data structures that share a dependency with the partitioned operation. Once data partitioning decisions are complete, the compiler then performs graph lowering to generate loops.

Next, the optimization phase applies loop fusion to the input routine. First it enumerates all potentially profitable combinations of two forms of loop fusion, interleaving and pipelining, that can be applied to the input routine. A fusion opportunity is potentially profitable when the loops share at least one data structure. Interleaving involves fusing loops of two independent operations. In this case, any data accessed by both operations are read once. Pipelining fuses two operations where one operation consumes the result of another. Pipelining reduces the number of data traversals and removes the need for an intermediate array to store the result of the first operation.

The optimization phase produces multiple versions of the input routine. Each version differs from all others in at least one way. The ways they can differ are the amount of fusion, parallelization, the number of cache blocks and the sizes of the blocks. These versions are then passed to the analytic phase.

In the evaluation phase, all versions of a routine are tested using a two step process. First, the analytic memory model is run on all versions, producing a sorted list of predicted runtimes. Then the best routines are empirically tested with the fastest generated into C code. The interaction between the two steps in the evaluation phase is user-controllable through runtime options. The user can select a maximum amount of time that the compiler should spend empirically searching through routines. Also, the user can specify that only those routines that the model predicts are within a certain threshold of the best predicted routine are empirically tested. The two options can be combined.

After the evaluation phase has identified the best version, the compiler outputs the code for it. A user can then make calls to the produced kernel within their own program.

3. SIMILAR RANKING OF ROUTINES

The enumeration of routines to be considered and the testing of those routines using hybrid search dominate the runtime of the BTO compiler. The model predicts small runtime differences between routines where vector operations are fused with matrix operations and the unfused variants. For example, for the GESUMMV calculation shown in Table 1, the compiler enumerates twelve possible versions with different amount of fusion. This calculation performs two matrix vector multiplies with the vector x multiplied by the matrices A and B. The results of the multiplications are then multiplied by the scalars α and β before being summed and stored in y. The model produces

Table 1. Test Routines: Greek Letters Represent Scalars, Lower Case Letters Vectors, and Upper Case Letters Matrices

Routine Name	Calculation
DGEMVT	$x = \beta A^T y + z$ $w = \alpha A x$
GEMVER	$B = A + u_1 v_1^T + u_2 v_2^T$ $x = \beta B^T y + z$ $w = \alpha B x$
GESUMMV	$y = \alpha A x + \beta B x$

Table 2. Machine Specifications

Processor	Speed	Mem	L1	L2	L3
Intel Core 2	2.4 GHz	4 GB	32 KB	4 MB	
Intel Core i7	2.8 GHz	4 GB	32 KB	256 KB	8 MB
AMD Opteron	2.6 GHz	3 GB	64 KB	1 MB	
Power PPC G5	2.3 GHz	8 GB	32 KB	512 KB	

predictions for the Core 2 system shown in Table 2 that all differ by less than 1%. Actual performance differences for the best and worst of these versions are less than 3%.

Also, when we graph the actual and predicted runtimes for the GEMVER calculation in Table 1 for the Core 2 system, we notice that many of the predicted and actual runtimes of routines are nearly identical. For many pairs of routines with near identical performance and predictions, the only difference is the fusion of a vector operation with a matrix operation. If this were always the case, the creating and testing the fusion of vector operations with matrix operations in the BTO compiler would be unnecessary.

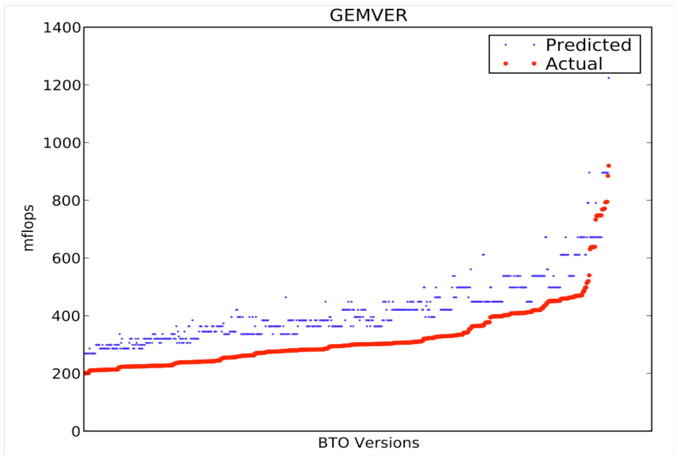

Figure 1: Predicted vs. Actual Runtime of the 648 Version of GEMVER Produced by the BTO Compiler

4. DETEMERNING WHICH VECTOR OPERATIONS MATTER

In this section, a vector operation refers to the fusion of loops where each loop accesses the same vector. For the unfused GESUMMV calculation shown in Figure 2, there are three sets of loops that contain vector operations that can be fused. Loops 1 and 2 can be fused with loops 4 and 5 to reduce the number of accesses to the x vector, where each element is accessed n times. Loops 3 and 6 can both be fused with loop 7 reducing the number of accesses to each element of the t1 and t2 vectors by one.

To determine whether fusing vector operations with matrix operations significantly impacts performance, we ran a series of tests. The test results were then analyzed to determine the significance of fusing vector operations. In this section. we first describe the environment, routines and methodology used to perform tests. We then present the results of these experiments including a statistical analysis of the results when needed.

```
for $i = 1$ to $n$ do                      // Loop 1
    for $j = 1$ to $n$ do                  // Loop 2
        $t_1(i) = t_1(i) + A(i,j)x(j)$
for $i = 1$ to $n$ do                      // Loop 3
    $t_1(i) = \alpha t_1(i)$
for $i = 1$ to $n$ do                      // Loop 4
    for $j = 1$ to $n$ do                  // Loop 5
        $t_2(i) = t_2(i) + B(i,j)x(j)$
for $i = 1$ to $n$ do                      // Loop 6
    $t_2(i) = \beta t_2(i)$
for $i = 1$ to $n$ do                      // Loop 7
    $y(i) = t_1(i) + t_2(i)$
```

Figure 2. Unfused GESUMMV calculation.

4.1 Test Environment and Methodology

To determine the impact of fusing vector operations with matrix calculations, we ran the three calculations in Table 1 on the four machines in Table 2. All tests were compiled using gcc with the –O3 compiler flag turned on. The DGEMVT and GEMVER kernels were chosen from the updated Basic Linear Algebra Subprograms (BLAS) [5] and contain vector operations where the vector is accessed only once. The GESUMMV operation was chosen because it contains vector operations where the vector is both accessed once and multiple times. For DGEMVT there are two sets of loops that can be fused that contain vector operations. For the GEMVER and GESUMMV calculations, there are respectively four and three sets of loops that containing vector operations that can be fused. All routines were chosen because they occur in important numerical linear algebra routines such as Householder Bidiagonlization [13].

Routines were run five times for each test of interest and performance differences less than 3% were considered small enough not to be significant. Any differences greater than 3% were subjected to statistical analysis to determine if the differences were statistically significant at a 95% confidence level. Student's paired T-test [14] was used to compare the results. Using one directional tests, our null hypothesis was that the results were identical and, unless the p value from running the comparison was less than .05, we accepted the hypothesis. A one directional test means that we only consider when fusion improves performance. If fusion negatively impacted the performance of a routine, in a statistically significant manner, then the hypothesis would be accepted.

4.2 Results and Analysis

The middle column of Table 3 shows the number of ways to fuse each routine with all vector operations considered. In all cases, we compared the performance of fusing and not fusing each operation by keeping all other fusion decisions the same and only changing the loop we were interested in. When a single pair of loops had a performance difference of more than 3%, we then used Student's paired T-test to determine if the differences were significant. The paired T-test was run for all pairs where the only difference between each routine in a pair was the fusion of the same vector operation.

For the DGEMVT and GEMVER calculations, the fusion of vector operations never significantly impacted routine performance. On the Core 2, PowerPC and i7 machines, the performance differences were always less than 2%. On the Opteron, differences were larger. Using a paired t-test analysis on the Opteron for the four pairs of interest resulted in p values of 0.14 to 0.68 for all pairs of interest, meaning that there was not a statistically significant difference in runtime for any loop pair of interest at the 95% confidence level.

For the GESUMMV calculation on the Core 2, i7 and PowerPC, system, differences in performance was always less than 3% for all fusion possibilities. On the Opteron system, however, performance differences were greater for vectors accessed more than once. In this case, runtime differences of over 30% resulted as shown in Figure 2. These differences were statistically significant. For vectors accessed only once, they were not significant with p values of 0.27 and 0.087.

From these experiments, we conclude that the fusion of vector operations where each element of the vector is accessed many times can significantly increase performance and must be considered when fusing loops. We also conclude that vector operations that only access elements only once do not significantly impact performance when fused and can be removed from the search space.

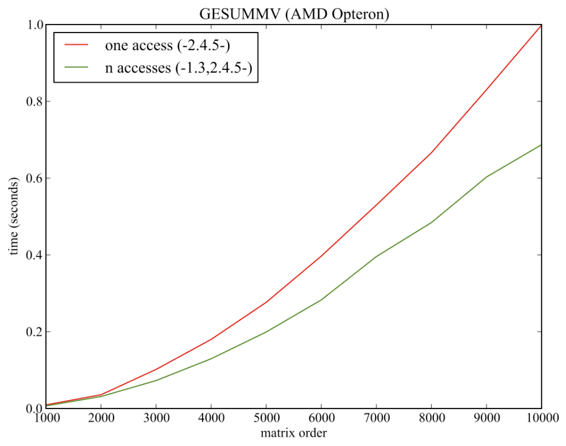

Figure 3. Runtime of fusing a vector operation where the vector is accessed n times on an Opteron system.

5. REMOVING VECTOR OPERATIONS

For each routine we tested there are two vector operations where all elements in the vector are accessed once. Removing the fusion of these operations with each other and with matrix operations from the space of considered routines results in a 75% reduction in the number of routines to be tested as shown in Table 3. For most routines, being able to eliminate a vector operation from the search space reduces the number of routines to be consider by approximately one half.

Table 3. Search Space of Routines

Routine Name	With Vector Operations	Without Vector Operations
DGEMVT	8	2
GEMVER	648	162
GESUMMV	12	3

To perform this reduction within the compiler, we have two strategies to test. In each strategy, we would need to write code to determine the relative cost of various operations. One option is to always fuse vector operations when enumerating routines and then unfuse them when testing in the model empirically. Another option is to only fuse vector operations with matrix operations when fusing the vector operation enables the fusion of matrix operations.

6. CONCLUSIONS AND FUTURE WORK

Within the BTO compiler, the largest part of the runtime is spent enumerating and comparing different versions of a routine. The difference between some of these versions is the fusion of vector operations with matrix operations. In the case where the vector operation accesses each element of the vector once and is fused with a matrix operation, there is never a statistically significant performance improvement from the fusion. By removing the consideration of fusing vector operations when the vectors are accessed only once with matrix operations we will significantly reduce the amount of time spent enumerating and searching in the BTO compiler.

To continue this work, we plan to perform an analysis of whether matrix-vector operations significantly impact the runtime of calculations when fused with matrix-matrix operations. In this case, each routine accesses the same amount of data. However, for matrix-matrix operations access each element of the matrix many times while matrix-vector operations access each element once.

7. ACKNOWLEDGEMENTS

The work of E. R. Jessup, I. Karlin, E. Silkensen and P. Zelinsky is supported by National Science Foundation award CCF 0830458. The work of J. G. Siek and G. Belter is supported by CAREER: Bridging the Gap Between Prototyping and Production. NSF Grant Number 0846121. The work of T. Nelson is supported by Algorithms and Software for Communication Avoidance and Communication Hiding at the Extreme Scale (CACHE) Institute, which is supported by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research. Argonne National Laboratory is operated for the DOE by UChicago Argonne, LLC, under Contract No. DE-AC02-06CH11357.

8. REFERENCES

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. Supercomputing, ACM/IEEE 1999 Conference, pages 69–81, Nov 1999.

[2] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In SC ’09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–12, New York, NY, USA, 2009. ACM.

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

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In Proceedings of the 11th International Conference on Supercomputing, pages 340–347, New York, NY, 1997. ACM Press.

[5] 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, R. C. Whaley, An updated set of basic linear algebra subprograms (BLAS), ACM Trans. Math. Softw. 28 (2) (2002) 135–151.

[6] 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 CC'08/ETAPS'08: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] A. Darte. On the complexity of loop fusion. Parallel Computing, 26:149–157, 1999.

[8] J. M. Dennis and E. R. Jessup. Applying automated memory analysis to improve iterative algorithms. SIAM Journal on Scientific Computing, 29(5):2210–2223, 2007.

[9] C. Farhat, A.Macedo, M.Lesoinne, A two-level domain decomposition method for the iterative solution of high frequency exterior Helmholtz problems, Numerische Mathematik 85 (2000) 283–308.

[10] M. Field, Optimizing a parallel conjugate gradient solver, SIAM J. Sci. Stat. Comput. 19(1998) 27–37.

[11] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005). In Special Issue on Program Generation, Optimization, and Platform Adaptation.

[12] G. Gao, R. Olson, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, pages 281–295, New Haven, CT, Aug. 2004.

[13] 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.

[14] R. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, New York, 1991. Wiley.

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

[16] B. Spencer Jr., T. Finholt, I. Foster, C. Kesselman, C. Beldica, J. Futrelle, S. Gullapalli, P. Hubbard, L. Liming, D. Marcusiu, L. Pearlman, C. Severance, G. Yang, NEESgrid: A distributed collaboratory for advanced earthquake engineering experiment and simulation, in: 13th World Conference on Earthquake Engineering, Vancouver, B.C, Canada, 2004, paper No. 1674

[17] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series, 16:521–530, June 2005.

[18] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and performance bounds for sparse AT Ax. In ICCS 2003: Workshop on Parallel Linear Algebra, Melbourne, Australia, June 2003.

[19] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In Proceedings of 1998 ACM/IEEE Conference on Supercomputing (CDROM), pages 1–27, Washington DC, 1998. IEEE Computer Society.