Understanding Memory Effects in the Automated Generation of Optimized Matrix Algebra Kernels

Elizabeth R. Jessup, Ian Karlin, Erik Silkensen

Department of Computer Science, University of Colorado, 430 UCB, Boulder, CO 80309

Geoffrey Belter, Jeremy Siek

Department of Electrical, Computer, and Energy Engineering, University of Colorado, 425 UCB, Boulder, CO 80309

Abstract

Efficient implementation of matrix algebra is important to the performance of many large and complex physical models. Among important tuning techniques is loop fusion which can reduce the amount of data moved between memory and the processor. We have developed the Build to Order (BTO) compiler to automate loop fusion for matrix algebra kernels. In this paper, we present BTO’s analytic memory model which substantially reduces the number of loop fusion options considered by the compiler. We introduce an example that motivates the inclusion of registers in the model. We demonstrate how the model’s modular design facilitates the addition of register allocation to the model’s set of memory components, improving its accuracy.

Keywords: matrix algebra, memory efficiency, automatic tuning

1. Introduction

Matrix algebra plays a role in evaluating physical models from such diverse areas as acoustic scattering, aerodynamics, combustion, global circulation, radiation transport, and structural analysis [1, 2, 3, 4]. The programs used to solve these problems are often large and complex, meaning that methods for implementing them efficiently are of broad importance. A variety of methods have been employed for the tuning of matrix algebra software, e.g. [5, 6, 7, 8, 9, 10, 11, 12]. Among them is loop fusion, a technique for reducing the amount of data moved from memory to the processor during program execution. Loop fusion has been the subject of much recent study [13, 14, 15, 16], and the new standard for the Basic Linear Algebra Subprograms (BLAS) [17] recognizes four commonly used routines that benefit from its application.
Matrix algebra software is often constructed as sequences of BLAS calls, and adjoining calls offer opportunities for fusion. However, there are two significant issues in such tuning. First, it can be difficult to know when it will be advantageous. Loop fusion can substantially reduce data access and thereby lead to significant speedups. Yet, in some circumstances, it can be detrimental to performance. Figure 1 shows loop fusion applied to the GEMVER kernel of the updated BLAS [17]. When a column of matrix $A$ remains in cache throughout an iteration of the loop, the fused implementation reads that column only once from main memory during that iteration. When just two vectors fit in cache, though, fusing the first two scaled vector additions forces the column of $A$ to be evicted meaning that it must be reread when accessed again. This example demonstrates that creating an efficient fused routine requires careful consideration of both algorithm and memory subsystem.

A second issue is the effort required to implement loop fusion. While rewriting codes to join loops is not generally a complex programming task, repeating it many times in search of the optimal arrangement is time consuming. At the same time, it is not possible to anticipate which BLAS will appear together, and, therefore, it is not practical to create a library of all possible fused BLAS combinations. To enable the optimization of arbitrary linear algebra kernels, we created the Build to Order (BTO) compiler [18] which automates loop fusion for sequences of matrix algebra operations. BTO takes in matrix and vector arithmetic statements expressed in annotated MATLAB and produces a tuned implementation of those statements in C. At its most basic, the compiler explores the search space of all possible combinations of two variants of loop fusion, determining the most efficient organization via empirical search. To improve on this approach, we introduced an analytic memory model that quickly estimates the costs of the routines generated by the compiler and, from those costs, identifies the best performing subset of them. The very fastest routine is distinguished via empirical search.

The first version of the analytic model based its predictions on cache and TLB misses. In this paper, we describe the model and explain how we discovered that register allocation should also be included. We show how we added register effects and demonstrate how the model’s predictive power improved as a result. The paper is organized as follows. In Section 2, we review related work. In Section 3, we describe how the model works within the BTO compiler at a high level and present typical predictions produced by the model. In Section 4, we present a code where the best version produced by the compiler is not the one the original model without registers chooses. In studying this discrepancy, we leverage the compiler’s ability to generate all possible routines in order to stress that model and reveal a need for it to account for registers. In Section 5, we show how we add in a new feature that predicts the register allocations of native compilers, e.g., gcc and icc. The resulting improvement in prediction of routines performance by the model is then presented.

2. Related Work

Techniques for identifying the best of a collection of routines include empirical search, analytic models and hybrid approaches using both methods. Successes have been reported for all three approaches. For example, both PHiPAC [12] and ATLAS [19, 20] use empirical search to determine the optimal combination of optimizations for matrix-matrix multiplication. Yotov et al. [21] show that analytic models can deliver similar performance in less time than empirical search for matrix-matrix multiplication. A follow up study [22] shows that models can be used to narrow the parameters and ranges for which empirical search is used while still producing high performance codes. Other
applications of empirical analysis include loop fusion decisions [23], optimizations and prefetching [24], and loop nesting [25]. Analytic models have been used to determine capacity misses for perfectly nested loops [26], for array padding [27], and for register allocation [28]. A similar approach to our model is the equations described in [29] which provide high accuracy cache miss predictions at large expense by means of reuse distances. Recent methods using both analytic modeling and empirical search have shown promise for obtaining good results in reasonable time. In [30], Chen et al. use an analytic model to identify, empirically test, and tune candidate implementations of matrix computations. To find the optimal tile size for matrix multiplication, Epshteyn et al. [31] adapt their analytic model based on empirical results using explanation-based learning. These hybrid methods take advantage of analytic models to guide them in the right directions and empirical search to more fully explore the areas identified by the model.

3. Analytic Model Overview, Purpose and Place in Compiler

To efficiently estimate the runtimes of the code variants produced by our compiler, we use an analytic model. In this section, we give a high level description of how the model works and its function within the BTO compiler. Results showing typical model behavior are presented along with explanation. A more detailed description of the model and analysis of its predictive ability can be found in [18].

In its first step, the model calculates reuse distances for each data structure at all loops. The reuse distance of a data structure is the number of unique data accesses between two accesses to an element of the data structure. The reuse distances are then used to determine the memory structure from which the data are read. Next, the total amount of data read from each memory structure for each loop is calculated along with the cost of moving that data to the processor. The memory structure with the largest cost in each loop is the one limiting performance of that loop. The overall routine cost for a loop that contains other loop(s) is assigned to be the maximum of its own cost and the cost of its components.

Runtime predictions are then passed to the compiler where they are ordered. The compiler keeps all routines with predicted times within a user-specified percentage of the best predicted time and empirically tests them to identify the fastest one which is output as a C kernel. In order to limit the number of routines that must be empirically tested, the model must be able to distinguish between routines with large differences in memory traffic cost but does not need to predict performance exactly.

Measured (actual) and predicted megaflops ratings are presented for two versions of the calculation $y = AA^T x$ produced by the BTO compiler are shown in Figure 2. The tests were run on an AMD Opteron with the specifications shown in Table 1. The figure shows that the model properly ranks the two versions across all matrix sizes. We have tested the model more thoroughly across a much broader range of problems and have confirmed that this figure’s results are typical. In a tradeoﬀ between precision and speed, the model assumes that caches are fully associative, leading to stairsteps in the predicted curves. The measured curves are typically smooth.

4. Improving the Analytic Model

We now present an analysis of memory bound matrix-vector multiplies that suggests ways to improve the analytic model. In particular, this analysis aids the development of the compiler’s memory model, enabling improvements that reduce the number of loop fusion options considered. This cutting is important because the number of possible routines grows rapidly with kernel complexity, making exhaustive testing expensive. We also present hardware performance counter data that show that we need to consider register allocation in the memory model.

We consider a sequence of matrix-vector multiplies. For example, we define a routine DGEMV2 which multiplies vectors $u_0$ and $u_1$ in turn by a matrix $A$ as shown in Figure 3. The annotated MATLAB provided in this figure serves as input to the compiler which generates all possible loop fusion combinations for the pair of matrix-vector multiplies.
Figure 4 shows three of these possibilities: no loop fusion, only outer loops fused, and all loops fused. These three choices range from the least complex (no fusion) to the most complex (fully fused) with one intermediate selection (outer loops).

\[
\text{DGEMV2}
\]

\begin{verbatim}
 in
   u0 : vector, u1 : vector,
   A : row matrix
 out
   v0 : vector, v1 : vector
 { 
   v0 = A * u0
   v1 = A * u1
 }
\end{verbatim}

Figure 3: DGEMV2

In order to evaluate the analytic model, we ran a series of experiments on sequences of one to eight successive matrix-vector multiplications for the fully fused and all outer loops fused combinations. The number of repetitions of matrix-vector multiplication in a test is denoted \( nvecs \). For example, \( nvecs = 2 \) for DGEMV2. All of the tests were run using the Intel icc compiler on the AMD Opteron described earlier.

As shown in Figure 5, the fully fused variant outperforms the outer loops one for \( nvecs < 5 \). For \( nvecs \geq 5 \), fusing outer loops is the more efficient choice. To explain this crossover, we instrumented the code to track memory traffic. Figure 6 compares L1 cache misses per flop measured for the two routines over the range of \( nvecs \) values \( 1–8 \). The fully fused routine suffers fewer L1 cache misses per flop than does the outer loops routine for all values of \( nvecs \). Both routines have nearly identical numbers of L2 cache misses and TLB misses per flop for all values of \( nvecs \): on plots of L2 and TLB misses versus \( nvecs \), the lines for the two routines are indistinguishable. Thus, the relative performance differences between the two cannot be explained by these three memory effects.

The cause of the crossover is found even higher in the memory hierarchy. An examination of the assembly code reveals that the crossover results from register spilling. Register spilling occurs whenever there are more variables in play than there are registers. In this case, the register allocator must save and restore spilled values to and from the L1 cache, incurring additional memory access costs. The assembly for an inner loop of the fused outer loops variant presented in Figure 7 is the same for all values of \( nvecs \). In contrast, Figure 8 shows that the assembly for the fully
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)

\( v_0[i] += A[i][j] \times u_0[j] \) for (j = 0; j < n; j++)
for (i = 0; i < n; i++)

\( v_1[i] += A[i][j] \times u_1[j] \) for (j = 0; j < n; j++)
for (i = 0; i < n; i++)

(a) No Fusion (b) All Outer Loops Fused (c) All Loops Fused

Figure 4: Three possible loop fusion options

Figure 5: The measured performances of the fully fused and outer loops fused routines compared with the predicted performance of the fully fused routine.

The fused version adds move instructions as nvecs increases from four to five to six. These increased instructions account for the degradation in performance of the fully fused version.

5. Accounting for Registers

To be able to account for register usage in the model, the number of registers was determined, the bandwidth between the level 1 cache and processor calculated, and the model code modified to account for how the native compiler allocates registers. The following section first describes how to incorporate registers into the model and then shows how adding them improves performance prediction.

5.1. The Changes Needed to Add in Registers

The first step in including registers to the model is to represent the registers as a memory structure. To do so on the Opteron system described in Section 3, we first determined that there are eight general purpose registers. However, because one is reserved for the stack pointer and another for the base pointer, only six registers can be used for general purpose computation. Then we determined the bandwidth between the level 1 cache and the processor by means of the DAXPY benchmark in STREAM2 [32] and stored it as the cost of register misses.

The next step involved modifying the code to account for the fact that registers are not allocated in a least recently used fashion. Instead, the native compilers attempt to allocate registers in a manner that reduces the number of reads into registers. To figure out which variables remain in registers, the following heuristics are used in the model to mimic the native compiler’s allocation. The iterate of an inner loop is stored in a register. A variable that is accessed within an inner loop more than once is stored in a register if one is available. Finally, when a register is available, it is
Figure 6: L1 misses per flop for the fused outer loops and fully fused versions of multiple matrix-vector multiplications. nvecs is number of multiplies.

L0: fdl (%eax,%edi,8)
    fmull (%esi,%edi,8)
    faddp %st,%st(1)
    fstl (%ebx,%edx,8)
    add $0x1,%edi
    cmp %ecx,%edi
    jl L0

Figure 7: Outer Loops Assembly

allocated by the model to the first instance of a variable that accesses data that never change in subsequent interactions of the current loop.

5.2. The Effect of Registers on Predictions

The result of adding registers into the model is shown in Figure 9 for fully fused routines. The graph shows that the model with registers predicts a decrease in performance beginning at nvecs = 5, which matches the behavior of the measured performance, while the model without registers, as noted in Section 4, predicts increased performance. At nvecs = 5, not all vectors in the inner loop of the fully fused calculation shown in Figure 4 remain in a register. The resulting growth in L1 cache misses causes performance to be bound on traffic from the L1 cache. Additionally, each increase of nvecs beyond 5 raises the ratio of L1 reads to computation and so reduces the performance predicted by the model.

We also compared both models’ predictions for the best routine for values nvecs = 1 to 6 to the results of exhaustively testing all routines for those values of nvecs. Complete tests for nvecs = 7 and 8 were not attempted due to their long runtimes: we estimate that accurately testing the more than 150,000 versions the compiler produces for nvecs = 8 could take ten or more days. Instead, we only ran tests for fused outer loops and fully fused versions for those two values of nvec. In all cases, the model with registers predicts that fusing all outer loops provides the best version. Also, inner loops should be fused together in groups of three or four when possible. Actual tests up to nvecs = 6 demonstrate that fusing inner loops in groups of three produces the best performance. The model predicts and experimental evidence from tests run shows that extra inner loops should be allocated in a way that keeps the number of groups of inner loops at a minimum and groups of 3 or 4 at a maximum.
For \( nvecs = 1 \) to 3, both versions of the model predict the best routine. When \( nvecs = 4 \), both models’ best predicted routine was the second best performing routine found by exhaustive search. In that case, the best predicted routine was less than 5% slower than the optimal routine found by exhaustive search. The latter optimal routine had all outer loops fused. It also had three inner loops fused together and one left alone. For \( nvecs = 5 \) to 6, the model with registers predicts the optimal routine and best predicted routine to be similarly performing. Empirical testing by the compiler finds the best routine in these cases. Using the version of the model without registers, the BTO compiler cannot find the optimal routine without empirically testing hundreds of routines. With registers included in the model, the accuracy of the model’s predictions and the performance of the routine chosen by the compiler improved.

6. Future Work

To further the improvement of our system we plan to introduce memory of past decisions. Since many linear algebra kernels have similar operations, having the compiler remember what decision it made in the past will allow it to trim branches from its decision tree. Past decisions could come from routines run by a user or from routines run at install time to profile the system’s response to various fusion combinations. Trimming away full search branches will become especially important as we explore how to create parallel routines and add other optimizations such as cache blocking to our compiler. In addition, we are improving the model to algebraically determine when a data structure no longer fits within a cache. The new model will not have to be run on every problem size of interest but rather once for all problem sizes. This improvement will allow us to quickly reason about blocking decisions and create problem size dependent kernels. Finally, the predicted performance of our tested routines is not the same as measured. Currently, we use a single benchmark to predict the performance of all routines. The benchmark was chosen for its overall good
performance across a suite of test examples. Using different benchmarks based on the properties of the routines being analyzed could improve predictions.


[30] C. Chen, J. Chame, M. Hall, Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy, in:
