

Memory Analysis and Tuning of Composed Linear Algebra Kernels

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

ABSTRACT

The performance of many linear algebra routines on modern processors is limited by data retrieval from main memory. In this paper, we demonstrate how to compose multiple linear algebra operations into single routines to reduce memory reads. A full analysis of the positive and negative memory effects of these composed routines is presented. Additionally, we examine how to combine other memory tuning techniques with composition to eliminate the negative memory effects composition sometimes creates.

1. INTRODUCTION

Linear algebra calculations are performed in a wide range of scientific and engineering disciplines. They are often the most time consuming part of simulations used to solve problems in the diverse fields of atmospheric science [?], quantum physics [?], and structural engineering [?]. The high cost of linear algebra means that reducing its runtime can have a large effect on overall routine performance [?, ?].

To reduce the runtime of a linear algebra calculation, one must identify the limiting factor of performance [?]. Once the bottleneck is identified, code can be rewritten to reduce its impact through a variety of program tuning techniques [?, ?, ?, ?]. Profiling tools such as those described in [?, ?, ?] can be used to identify the computer structure that is limiting performance of a routine and the proper tuning technique to decrease its effect.

When analyzing and tuning a program, it is important to understand how the memory hierarchy of a computer is organized. When executing a program to perform a calculation, a computer must first read data from memory into the processor where the calculation occurs [?]. The speed at which data are moved from memory to the processor is limited by the bandwidth, the amount of data that can be moved in a given time. Due to the Von Neumann bottleneck [?], the amount of data that can be moved from memory to the processor per unit time is much less than the amount of data the processor can perform calculations on in the same unit of time. The speed disparity is increasing as the speed of processors increases faster than the speed of bandwidth between processors and memory [?]. Caches, which store data closer to the processor for quicker access, allow algorithms that reuse data to overcome the bottleneck [?]. However, the execution of algorithms that are heavily dependent on memory reads due to limited or no data reuse, for example, matrix-vector multiplication, only see speed increases proportional to memory bandwidth increases and not the much faster processor speed increases [?].

To increase the speed of memory bound matrix-vector multiplies,

two or more calculations that access the same data from memory can be composed into one routine [?, ?]. Composed calculations which take two operations that can be performed separately and merge them into one routine, can see performance gains proportional to the reduction in memory traffic. However, combining multiple calculations into one can lead to bad memory access patterns such as non-consecutive reads and an increase in the amount of data that must fit in cache for good routine performance. In this paper, we discuss two different methods to combine two dense matrix-vector calculations into one routine. In one routine the multiplies are independent of each other and in the other the multiplies are dependent on each other. The negative effects of composing each are shown along with methods to minimize these effects.

In section 2, we examine how to compose two independent matrix-vector multiplies and show how cache blocking can be used to minimize the effect of extra vectors needing to reside in cache for efficient performance. In section 3, we show how to compose two dependent matrix-vector multiplies and how software pipelining can be used to keep the bus from memory to the processor constantly busy. Finally, in section 4, we present conclusions about memory efficient linear algebra routines, followed by the future direction of our research in section 5.

2. CREATING AN INDEPENDENT COMPOSED ROUTINE

Independent calculations occur when the result of each calculation does not affect the result of the other. When independent calculations occur in close proximity to each other in an algorithm, they can be combined. An example of two such matrix-vector multiplies are $Ax = r$ and $A^T y = s$. Both calculations occur in the bi-conjugate gradient method [?]. In this section, we demonstrate how to compose this calculation and how to use cache blocking to maintain the efficiency of the composed calculation for large sized matrices.

2.1 Composing $Ax = r, A^T y = s$

Since the two matrix vector multiplies in this calculation are independent, they can be merged into a single loop through a performance optimization technique called loop merging [?] as shown in figure 1. The left algorithm is the uncomposed routine and the right algorithm is the composed routine. The resulting composed routine, therefore, only needs to read the matrix A from memory once to perform both calculations.

Tests run on a 2.4 GHz Intel Xeon processor with 2GB of memory and a 512KB level 2 cache show the large speedups that result from composing $Ax = r$ and $A^T y = s$. All programs were com-

```

for i = 1:n
  for j = 1:n
    r(j)+ = A(j,i) * x(i)
  for i = 1:n
    for j = 1:n
      s(i)+ = A(j,i) * y(j)

```

Figure 1: Composing $Ax = r, A^T y = s$

piled using the Intel Fortran compiler ifort with the optimization flag -O3 turned on. Performance of the routines presented in this paper are compiler independent as tests run using different compilers show similar speedups. In figure 2, the composed routine is compared to two uncomposed routines: an un-optimized baseline routine and the highly optimized GotoBLAS. Square matrices were used and we are most interested in results of matrices of size 400 and larger as these do not fit in cache and, therefore, require all reads of the matrix to come from memory. We measure the throughput, the amount of work performed per unit time, in millions of floating point operations per second(MFlops) to compare routine performance. The left graph shows that for matrices larger than cache the composed algorithm produces a 90% increase in throughput over the baseline algorithm and 50% to 80% improvement over the optimized algorithm for matrix sizes between 400 and 10000. The right graph shows the reduction in memory reads produced by composition. Cache misses were measured using the Performance Application Programming Interface (PAPI). Memory reads were cut in half from just over .15 per floating point operation to just over .07 per floating point operation. Each level 2 cache miss requires a read to memory and, therefore, shows the effects of composition on reducing memory reads.

2.2 Cache Blocking to Prevent Performance Declines

In figure 2, the composed algorithm’s performance begins to decline for matrices of size 10000 and larger. The decline occurs as the vectors used in the calculation become too large to fit in cache and must be read from memory. The uncomposed algorithm accesses two vectors with each iteration of the outside loop, while the composed algorithm accesses three vectors. The three vectors are column i of A , r and y . In accessing three vectors per outer loop the amount of data that must fit in cache for good performance is increased and this explains the performance drop of the composed routine for large matrix sizes. To reduce the amount of data that must fit within the cache at once, cache blocking can be used.

Cache blocking breaks a matrix into multiple pieces such that the smaller pieces fit into cache [?, ?]. In doing so, data can be used in multiple calculations each time they are read from memory. In the case of matrix-vector multiplication, since each element of the matrix is used only once, the result and multivectors are the only structures where cache blocking can produce data reuse. In figure 3, we show how a matrix-vector multiply with column-wise access of the matrix is split into two pieces when cache blocking is used. The product $Ax = b$ can be computed in two steps. First, the top half of the vector b_1 is computed by multiplying the block row A_1 by the vector x . The block row A_1 is read in a column at a time do to the column major Fortran storage used. Then b_2 is computed in the same manner from the block row A_2 . Thus, only half of A and b need to fit into cache at any one time. To cache block $Ax = r, A^T y = s$ we split the matrix A into two block rows

and the vectors r and y in half.

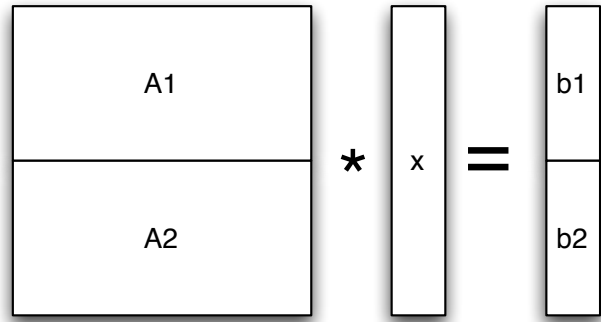


Figure 3: Diagram of how to cache block a column major matrix-vector multiply.

As shown in figure 4 cache blocking increases the performance of the composed routine to within 10% of what it was without cache blocking. The gap in performance can be attributed to cache blocking creating non-consecutive memory reads, which can have a high latency, and disrupt the memory prefetcher [?]. Cache blocking as expected reduces the number of memory reads as seen by the decrease in level 2 cache misses also shown, in figure 4. It should be noted for smaller matrices cache blocking produces a slower algorithm than just composing due to the non-consecutive memory reads introduced by cache blocking.

3. CREATING A DEPENDENT COMPOSED ROUTINE

Dependent calculations occur when the partial result of one calculation must be complete before the other calculation can be started. An example of one such calculation is $AA^T x = b$, which occurs when using Kleinberg’s algorithm to find authorities in hyper-linked environments [?]. It is important to note that the same technique used to compose $AA^T x = b$ can be used to compose the computation $A^T Ax = b$ [?], which is used in linear programming and linear least squares [?, ?].

3.1 Composing $AA^T x = b$

Composing this calculation requires accounting for the dependence between the two matrix-vector multiplies. The matrix-vector product $A^T x$ is computed as inner products of the rows of A^T with the vector x . Each inner product results in one element of the vector t . The matrix-vector product $At = b$ is then computed via a linear combination of the columns of A with the elements of t as coefficients. Thus, the second matrix-vector product cannot begin until at least one element of the vector t has been computed. As each new element of t is computed, one more column of A can be added to the linear combination. Because the dot product with that column is computed independently of the linear combination, the column is retrieved from cache once for each matrix-vector product.

Tests run on a 2.4 GHz Intel Xeon processor with 4GB of memory and a 4MB level 2 cache show large speedups from composition. In figure 6 square matrices were once again used and comparisons were made using ifort to compile the code with the -O3

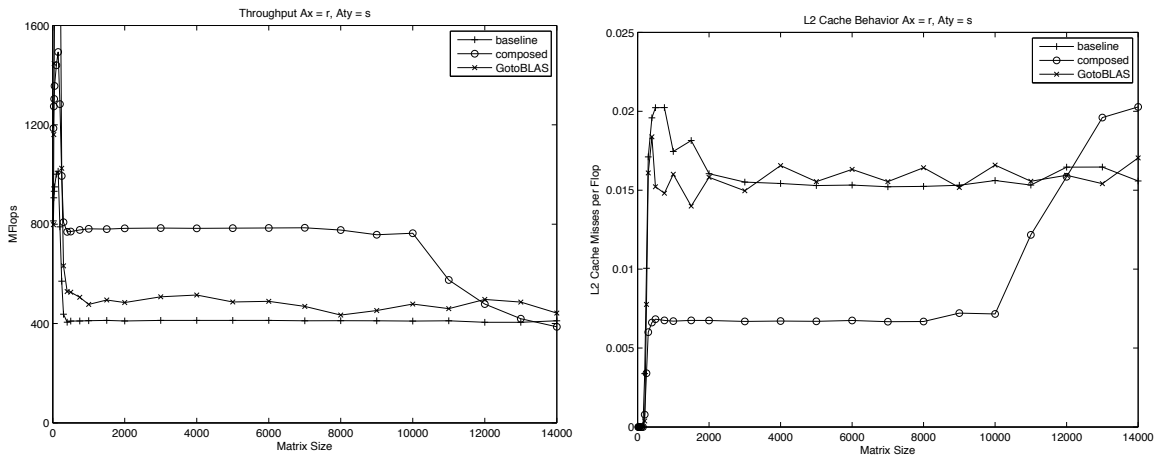


Figure 2: $Ax = r, A^T y = s$ throughput and memory reads.

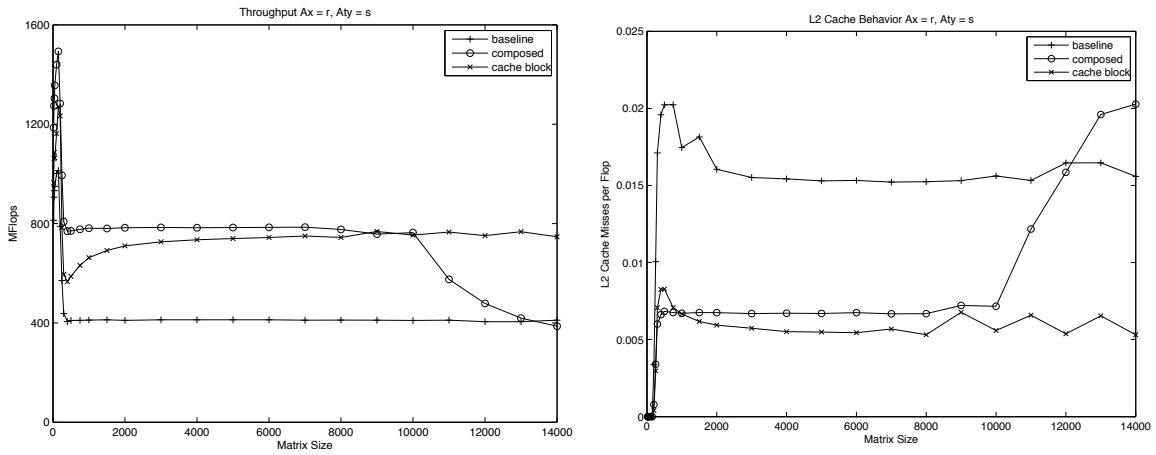


Figure 4: $Ax = r, A^T y = s$ cache blocking results.

```

for  $i = 1:n$ 
  for  $j = 1:n$ 
     $t(i) += A(j, i) * x(j)$ 
  for  $i = 1:n$ 
    for  $j = 1:n$ 
       $b(i) += A(j, i) * t(j)$ 

```

Figure 5: Composing $AA^T x = b$

option turned on as in section 2. For this machine, the matrices begin to become larger than cache at sizes greater than 1000 and for these matrices the composed calculation produces an increase in throughput of approximately 40% over the baseline. Comparisons to GotoBLAS are not included due to it only being about 5% faster than the baseline. Once again the right graph shows level 2 cache misses, which equate to memory reads.

3.2 Software Pipelining to Improve Memory Access Patterns

When using the composed algorithm in figure 5, each column of A is read once and then immediately read a second time. The second read is from cache and, therefore, quicker than the first read from memory. However, while this second read is occurring no new data is being transferred over the bus between memory and the processor. Since the ability to move data between memory and the processor is what limits the overall speed of the routine, having the memory bus idle for any period of time reduces the overall performance of the composed routine. To prevent the memory bus from becoming idle, we use software pipelining to interleave the second read of one column with the first read of the next so a new column is always being read from memory.

Software pipelining is used to interleave operations where a dependency exists. It reorders dependent operations such that they are overlapped in code [?]. The calculation $b[i] = b[i - 1] + c[i - 1]$, $c[i] = c[i - 1] + c[i - 2]$, and $a[i] = b[i + 1] + c[i + 1]$ can be performed using software pipelining. The operation is pipelined by calculating $b[i]$, $c[i]$, and $a[i - 1]$ at each iteration of a loop rather than computing b and c in one loop and then a in another. The software pipelined algorithm reduces the number of times that vectors b and c need to be read in from memory by one. In this paper, we do not use software pipelining to reduce memory reads but rather to retrieve data from memory for the operation $AA^T x = b$ more efficiently. The algorithm to perform this is shown in figure 7. The middle two for loops in the figure are where the pipelining occurs. Of note is the first statement within these loops that is bringing in new data from memory with every execution, therefore, keeping the memory bus constantly busy.

```

for  $j = 1:n$ 
   $t(1) = A(j, 1) * x(j)$ 
for  $i = 2:n$ 
  for  $j = 1:n$ 
     $t(i) = A(j, i) * x(j)$ 
     $b(j - 1) = A(j - 1, i) * t(i - 1)$ 
  for  $j = 1:n$ 
     $b(j) = A(j, n) * t(n)$ 

```

Figure 7: How to software pipeline $AA^T x = b$

As shown in figure 8 software pipelining increased the throughput of the composed routine by up to 20% over the non-pipelined rou-

tine. Performance gains of over 60% also occur when compared to the baseline. The number of level 2 cache misses are not shown as the number of misses is unaffected by software pipelining, as pipelining did not affect total number of memory reads it just reorganized them in a more efficient manner.

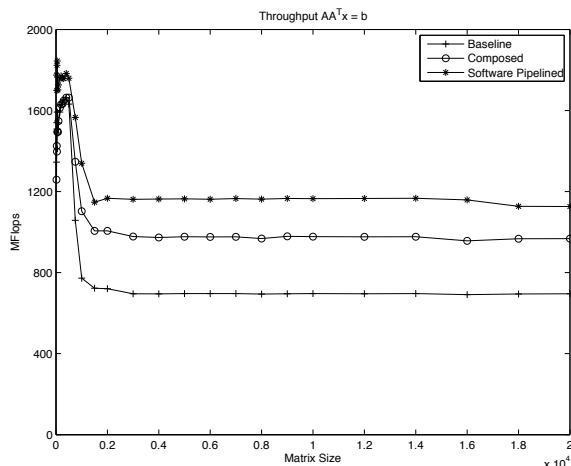


Figure 8: $AA^T x = b$ software pipelining results.

4. CONCLUSIONS

Composed routines significantly reduce the amount of data that linear algebra routines need to read from memory. When linear algebra calculations are memory bound, any reduction in memory traffic significantly increases routine speed. Composition reduces memory traffic and therefore increases routine speed. However, it can change memory access patterns, introduce latency bound non-consecutive reads and increase the amount of data that needs to fit in cache for efficient performance. Other tuning techniques can be combined with composition to reduce or eliminate these memory effects, creating even faster routines.

5. FUTURE WORK

In this paper, we present the memory effects caused by creating two different composed linear algebra routines. After presenting each routine we show how to overcome the bad memory costs introduced when creating each composed routine. We plan to continue this work by extending our domain to sparse routines and by analyzing multi-core routines. With all our work, we will use other performance tuning techniques to overcome negative memory access patterns introduced by composition.

6. ACKNOWLEDGMENTS

Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research, a grant from the IBM Shared University Research (SUR) program and the Intel Corporation.

7. REFERENCES

- [1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.
- [2] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an

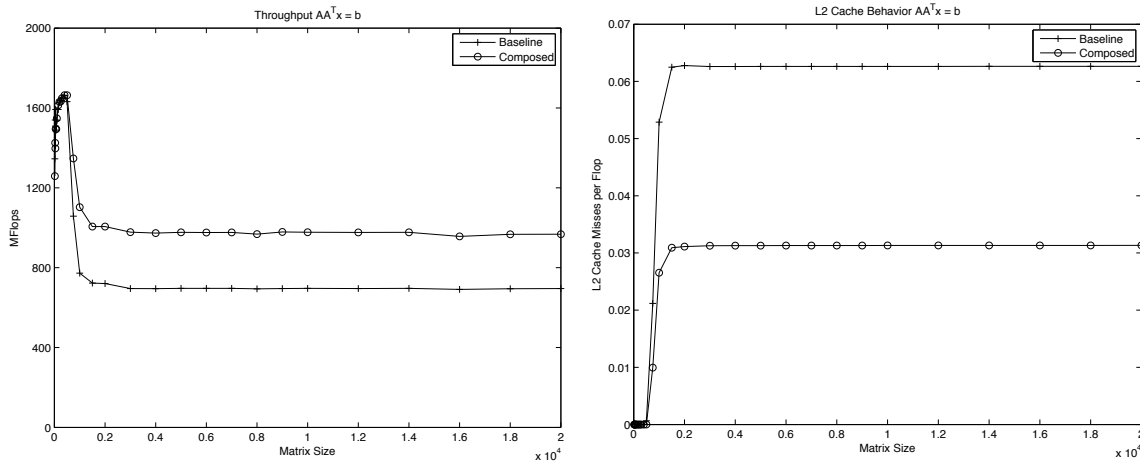


Figure 6: $AA^T x = b$ throughput and memory reads.

- unstructured mesh CFD application. *Supercomputing, ACM/IEEE 1999 Conference*, pages 69–81, Nov 1999.
- [3] J. Backus. Can programming be liberated from the Von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [4] A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of GMRES. *SIAM J. Sci. Comput.*, 27:1608–1626, 2006.
- [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*. SIAM, second edition, 1994.
- [6] 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 11th International Conference on Supercomputing*, pages 340–347, New York, NY. ACM Press.
- [7] T.-H. Chen and C. C.-P. Chen. Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative methods. In *Proceedings of Design Automation Conference*, pages 559–562, 2001.
- [8] J. M. Dennis. *Automated memory analysis: improving the design and implementation of iterative algorithms*. PhD thesis, University of Colorado, Boulder, CO, 2005.
- [9] G. Gao, R. Olson, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *In Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, New Haven, CT, Aug. 2004.
- [10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.
- [11] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. Technical Report LAPACK Working Note 174, November 2005.
- [12] D. Kincaid and W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole, third edition, 2002.
- [13] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46:604–632, 1996.
- [14] M. S. Lam, E. E. Rothber, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Palo Alto, CA, Apr. 1991.
- [15] 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, and G. Yang. NEESgrid: A distributed collaboratory for advanced earthquake engineering experiment and simulation. In *13th World Conference on Earthquake Engineering*, Vancouver, B.C, Canada, Aug 2004. Paper NO. 1674.
- [16] TACC. Gotoblas. <http://www.tacc.utexas.edu/resources/software/#blas>, 2007.
- [17] G. Vidal. Efficient simulation of one dimensional quantum many-body systems. In *Physical Review Letters*, 93(4):1–4, 2004.
- [18] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, pages 1–35, Baltimore, MD, November 2002.
- [19] R. Vuduc, A. Gyulassy, J. Demmel, and K. Yelick. Memory hierarchy optimizations and performance bounds for sparse $A^T A * x$. Technical Report UCB/CS-03-1232, University of California, Berkeley, February 2003.
- [20] W. Wang and D. P. O’Leary. Adaptive use of iterative methods in interior point methods for linear programming. Technical Report 3560, College Park, MD, 1995.
- [21] 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.
- [22] M. Xue, K. K. Droegemeier, and V. Wong. The advanced regional prediction system (ARPS) - a multiscale nonhydrostatic atmospheric simulation and prediction model. part I: model dynamics and verification. *Meteorology and Atmospheric Physics*, 75:161–193, 2000.