Improving Performance of Hypermatrix Cholesky Factorization⋆

José R. Herrero, Juan J. Navarro

Computer Architecture Department, Universitat Politècnica de Catalunya,
Jordi Girona 1-3, Mòdul D6, E-08034 Barcelona, (Spain)
{josepr,juanjo}@ac.upc.es

Abstract. This paper shows how a sparse hypermatrix Cholesky factorization can be improved. This is accomplished by means of efficient codes which operate on very small dense matrices. Different matrix sizes or target platforms may require different codes to obtain good performance. We write a set of codes for each matrix operation using different loop orders and unroll factors. Then, for each matrix size, we automatically compile each code fixing matrix leading dimensions and loop sizes, run the resulting executable and keep its Mflops. The best combination is then used to produce the object introduced in a library. Thus, a routine for each desired matrix size is available from the library. The large overhead incurred by the hypermatrix Cholesky factorization of sparse matrices can therefore be lessened by reducing the block size when those routines are used. Using the routines, e.g. matrix multiplication, in our small matrix library produced important speed-ups in our sparse Cholesky code.

1 Introduction

The Cholesky factorization of a sparse matrix is an important operation in the numerical algorithms field. This paper presents our work on the optimization of the sequential algorithm when a hypermatrix data structure is used.

1.1 Hypermatrix representation of a sparse matrix

Sparse matrices are mostly composed by zeros but often have small dense blocks which have traditionally been exploited in order to improve performance [1]. Our application uses a data structure based on a hypermatrix (HM) scheme [2]. The matrix is partitioned recursively into blocks of different sizes. The HM structure consists of several (*N*) levels of submatrices. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated as such. Null pointers in pointer matrices indicate that the corresponding subblock does not have any non-zero elements and is therefore

⋆ This work was supported by the Ministerio de Ciencia y Tecnología of Spain and the EU FEDER funds (TIC2001-0995-C02-01)

http://people.ac.upc.edu/josepr/

unnecessary, both for storage and computation. Figure 1 shows a sparse matrix and its corresponding hypermatrix with 2 levels of pointers.

The main potential advantages of a HM structure w.r.t. 1D data structures, like the Compact Row Wise structure, are: the ease of use of multilevel blocks to adapt the computation to the underlying memory hierarchy; and the operation on dense matrices.

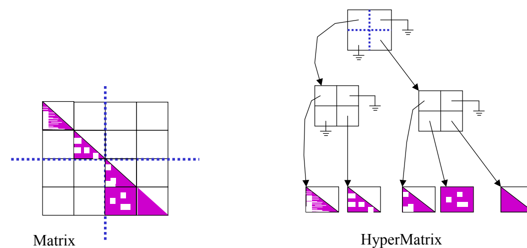

Fig. 1. A sparse matrix and its corresponding hypermatrix.

Choosing a block size for data submatrices is rather difficult. Large block sizes favour greater potential performance when operating on dense matrices. On the other hand, the larger the block is, the more likely it is to contain zeros. Since computation with zeros is useless, effective performance can therefore be low. Thus, a trade-off between performance on dense matrices and operation on non-zeros must be reached. The use of *windows* of non-zero elements within blocks allows for a larger default block size. When blocks are quite full operations performed on them can be rather efficient. However, in those cases where only a few non-zero elements are present in a block, only a subset of the total block is computed. Figure 2 shows a window of non-zero elements within a larger block. The window of non-zero elements is defined by it's top-left and bottom right corners. Zeros stored outside those limits are not used in the computations. Null elements within the window are still computed. However, the overhead can be greatly reduced.

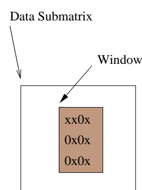

Fig. 2. A data submatrix and a window within it.

A commercial package known as PERMAS uses the hypermatrix structure [3]. It can solve very large systems out-of-core and can work in parallel. However, the disadvantages of the hypermatrix structure mentioned above introduce a large overhead. Recently a variable size blocking was introduced to save storage and to speed the parallel execution [4]. In this way the HM was adapted to the sparse matrix being factored.

The work presented in this paper is focused on the optimization of a hypermatrix Cholesky factorization based on the data structure presented above.

We have developed some rather efficient routines which work on small matrices. The purpose of these routines is to get high performance while keeping low the overhead due to unnecessary computations on null elements.

1.2 Goals

This work focuses on obtaining high performance from a hypermatrix Cholesky factorization. We want to reduce the overhead introduced by operations on zeros when large blocks are used. This can be done by reducing the block size. However, in order to keep high performance, we need specialized routines which operate very efficiently on small matrices.

1.3 Motivation

Figure 3 shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164 processor.

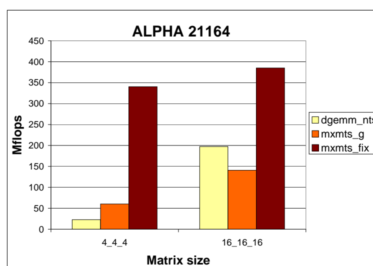

Fig. 3. Comparison of performance of different routines for matrix sizes 4x4 and 16x16.

The vendor's BLAS routine, labeled as *dgemm_nts*, fails to produce good performance for a very small matrix product (4x4) getting better results as matrix dimensions grow towards a size that fills the L1 cache (16x16). A simple matrix multiplication routine *mxmts_g* which avoids any parameter checking and scaling of matrices (*alpha* and *beta* parameters in *dgemm*) can outperform the BLAS for very small matrix sizes. The matrix multiplication code *mxmts_fix* in our library gets excellent performance for small matrices of sizes 4x4 and 16x16. There is actually one routine for each matrix size. In this paper we call all of them *mxmts_fix* for convenience. Each *mxmts_fix* routine is obtained by fixing leading dimensions and loop limits at compilation time and trying a set of codes with different loop orders and unroll factors. The one producing the best result is then selected.

1.4 Related work

The Cholesky factorization of a sparse matrix has been an active area of research for more than 30 years [1, 5–9].

Iterative compilation [10] consists in a repetitive compilation of code using different parameters. Program transformations like loop tiling and loop unrolling are very effective techniques to exploit locality and expose instruction level parallelism. The authors claim that finding the optimal combination of tile size and unroll factor is difficult and machine dependent. Thus, they propose an optimization approach based on the creation of several versions of a program and

decide upon the best by actually executing them and measuring their execution time. Our approach for obtaining high performance codes is similar with the difference that, while they apply a set of transformations, we use simple codes and let the compiler do its best.

Several projects were targeted at producing efficient BLAS routines through automatic tuning [11–13]. The difference with our work is that they are not focused on operations on small matrices.

A software for the parallel solution of sparse linear systems called BlockSolve95 [14] uses macros to put code inline for Level 2 BLAS routines GEMV and TRMV. They claim an improvement ratio between 1.2 and 2 for single processor codes working on small systems. Our approach however, is based on the improvement of Level 3 BLAS routines working on small matrices.

The reminder of the paper is organized as follows: first we present our optimization of routines operating on small matrices, namely the matrix multiplication operation. Then we show the impact of its application to the HM Cholesky factorization.

2 The Small Matrix Library (SML)

2.1 Generation of efficient code

Creation of efficient code has traditionally been done manually using assembly language and based on a great knowledge of the target architecture. Such an approach, however cannot be easily undertaken for many target architectures and algorithms.

Alternatively, creation of efficient codes specific for a target computer can be written in a high level language [15, 16]. This approach avoids the use of the assembly language but keeps the difficulty of manually tuning the code. It still requires a deep knowledge of the target architecture and produces a code that, although portable, will rarely be efficient on a different platform.

A cheaper approach relies on the quality of code produced by current compilers. The resulting code is usually less efficient than that written manually by an expert. However, its performance can still be extremely good and some times it can even yield better code. We have taken this approach for creating a Small Matrix Library (SML).

For each desired operation, we have written a set of codes in Fortran. For instance, for a matrix multiplication we have codes with different loop orders (*kji, ijk*, etc.) and unroll factors. Using a *Benchmarking Tool* [17], we compile each of them using the native compiler trying several optimization options. For each resulting executable, we automatically execute it and register its highest performance. These results are kept in a database and finally employed to produce a library using the best combination of parameters.

By fixing the leading dimensions of matrices and the loop trip counts we have managed to obtain very efficient codes for matrix multiplication on small matrices. Since several parameters are fixed at compilation time the resulting object code is only useful for matrix operations using these fixed values. Actual

parameters of these routines are limited to the initial addresses of the matrices involved in the operation performed. Thus, there is one routine for each matrix size. For convenience, we call all of them *mxmts_fix* in this paper, but each one has its own name in the library.

We also tried feedback driven compilation using the Alpha native compiler but performance either remained the same or even decreased slightly. We conclude that, as long as a good compiler is available, fixing leading dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels.

2.2 Matrix multiplication performance

Figure 4a shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164. The matrix multiplication performed in all routines benchmarked uses the first matrix without transposition (n), the second matrix transposed (t), and subtracts the result from the destination matrix (s). This is the reason why we call the BLAS routine *dgemm_nts*.

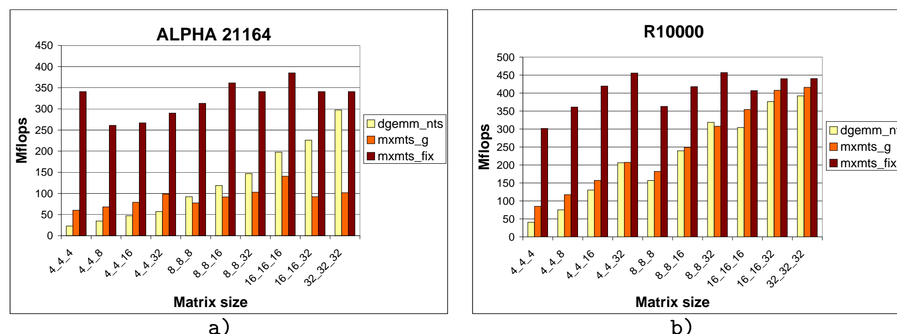

Fig. 4. Comparison of the performance of different routines for several matrix sizes: **a)** on an Alpha **b)** on an R10000.

The vendor BLAS routine *dgemm_nts* yields very poor performance for very small matrices getting better results as matrix dimensions grow towards a size that fills the L1 cache (8 Kbytes). This is due to the overhead of passing a large number of parameters, checking for their feasibility, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when operation is performed on large matrices. However, it is notable when small matrices are multiplied. Also, since its code is prepared to deal with large matrices, further overhead can appear in the inner code by the use of techniques like strip mining.

A simple matrix multiplication routine *mxmts_g* which avoids any parameter checking and scaling of matrices can outperform the BLAS for very small matrix sizes.

Finally, our matrix multiplication code *mxmts_fix* with leading dimensions and loop limits fixed at compilation time gets excellent performance for all block sizes ranging from 4x4 to 16x16. The latter is the maximum value that allows for a good use of the L1 cache on the Alpha unless tiling techniques are used.

Figure 4b shows the performance of different routines for several matrix sizes on the R10000 processor. Results are similar to those of the Alpha with the only differences that the L1 cache is larger (32 Kbytes) and *mxmts_g* performs very well. This is due to the ability of the MIPSpro F77 compiler to produce software pipelined code, while the Alpha compiler hardly ever manages to do so.

Though our SML codes were adapted to the underlying architecture, they were written in a high level programming language (FORTRAN). The resulting codes were very good for both the Alpha-21164 and the R10000 processors. We believe these results can be generalized for other current superscalar architectures.

3 Using SML routines as computational kernels for HM Cholesky

We have used SML routines to improve our sparse matrix application based on hypermatrices. Matrices were ordered with METIS [18] and renumbered by an elimination tree postorder. Performance varies substantially from matrix to matrix due to the different sparsity patterns and density. However, we see that using our matrix multiplication in SML improves performance substantially for all the benchmarks and block sizes.

Figure 5 shows results of the HM Cholesky factorization on an R10000[1] for matrix QAP15 from the Netlib set of sparse matrices corresponding to linear programming problems [19]; and problem pds40 from a Patient Distribution System (40 days) [20]. Ten submatrix sizes are shown: 4x4, 4x8, 4x16, . . . 32x32. Effective Mflops are presented. They refer to the number of useful floating point operations performed per second. This metrics excludes useless operations on zeros performed by the HM Cholesky algorithm when data submatrices contain zeros.

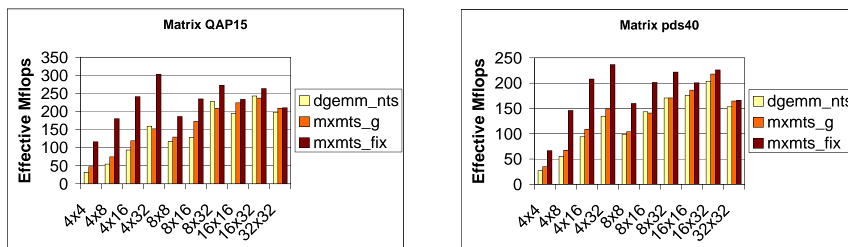

Fig. 5. Factorization of matrices QAP15 and pds40: Mflops obtained by different MxM codes in HM Cholesky on an R10000.

When *dgemm_nts* is used, the best performance is usually obtained with data submatrices of size 16×16 or 16×32. Since the amount of zeros used can be large, the effective performance is quite low. Using *mxmts_fix* however, smaller submatrix sizes usually produce better results than larger submatrix sizes. Particularly effective in this application is the use of rectangular matrices due to the fill-in produced by the Cholesky factorization (skyline). For instance, using 4×16

[1] Results on the Alpha are similar.

or 4×32 submatrix sizes the routine used yields very good performance. Since the number of operations on zeros is considerably lower, the effective Mflops obtained are much higher than those of any other combination of size and routine.

The use of a fixed dimension matrix multiplication routine speeded up our Cholesky factorization between 20% and 100% depending on the input matrix.

Matrix	*Dimension*	*Factor NZs*	*Density*	*Mflops*
TRIPART1	4.2	1.1	0.127	223.0
TRIPART2	19.7	5.9	0.030	226.9
TRIPART3	38.8	17.8	0.023	237.4
TRIPART4	56.8	76.8	0.047	278.2
pds40	76.7	27.6	0.009	236.6
pds50	95.9	36.3	0.007	249.9
pds80	149.5	64.1	0.005	254.4
pds90	164.9	70.1	0.005	263.3
QAP15	6.3	8.7	0.436	303.1

Table 1. Characteristics and performance of HM Cholesky on several LP problems

Table 1 shows the characteristics of several matrices obtained from linear programming problems [19, 20] and the performance obtained by our modified hypermatrix code. The factorization was performed on an R10000 processor with a theoretical peak performance of 500 Mflops. Dimensions are in thousands and Factor non-zeros are in millions. Using SML routines our HM Cholesky often gets over half of the processor's peak performance for medium size matrices factored in-core.

4 Conclusions

We have shown that fixing dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels. Since no single algorithm was the best for all matrix sizes in any platform, we conclude that an exhaustive search is necessary to get the best one for each matrix size. For this reason we have implemented a *Benchmarking Tool* which automates this process.

We have generated a small matrix library (SML) on a couple of systems obtaining very efficient codes specialized on operations on small matrices. These routines outperform the vendor's BLAS routine for small matrix sizes.

Fast computations on small matrices are of great utility in sparse matrix computations. A direct application of our small matrix library can be found in the hypermatrix Cholesky factorization. High performance routines operating on small matrices allow for the election of small block sizes. This choice avoids operation of non-zeros while retaining good performance. In practice, we found that blocks of size 4x16 and 4x32 often produced the best results for the hypermatrix Cholesky factorization.

We obtained important speed-ups in our Cholesky factorization application by using the SML routines. Therefore, we believe it is worthwhile to develop this sort of library.

References

1. Duff, I.S.: Full matrix techniques in sparse Gaussian elimination. In: Numerical analysis (Dundee, 1981). Volume 912 of Lecture Notes in Math. Springer, Berlin (1982) 71–84
2. Fuchs, G., Roy, J., Schrem, E.: Hypermatrix solution of large sets of symmetric positive-definite linear equations. Comp. Meth. Appl. Mech. Eng. **1** (1972) 197–216
3. Ast, M., Fischer, R., Manz, H., Schulz, U.: PERMAS: User's reference manual, INTES publication no. 450, rev.d (1997)
4. Ast, M., Barrado, C., Cela, J., Fischer, R., Laborda, O., Manz, H., Schulz, U.: Sparse matrix structure for dynamic parallelisation efficiency. In: Euro-Par 2000,LNCS1900. (2000) 519–526
5. George, A., Liu, J.W.H.: Computer Solution of Large Sparse Positive-Definite Systems. Prentice-Hall, Englewood Cliffs, NJ (1981)
6. George, A., Gilbert, J.R., Liu, J.W., eds.: Graph Theory and Sparse Matrix Computation. Volume 56 of The IMA volumes in mathematics and its applications. Springer-Verlag, New York (1993)
7. Ng, E.G., Peyton, B.W.: Block sparse Cholesky algorithms on advanced uniprocessor computers. SIAM J. Sci. Comput. **14** (1993) 1034–1056
8. Ashcraft, C., Grimes, R.G.: The influence of relaxed supernode partitions on the multifrontal method. ACM Trans. Math. Software **15** (1989) 291–309
9. Rothberg, E., Gupta, A.: An efficient block-oriented approach to parallel sparse Cholesky factorization. SIAM J. Sci. Comput. **15** (1994) 1413–1439
10. Kisuki, T., Knijnenburg, P., O'Boyle, M.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Parallel Architectures and Compilation Techniques. (2000) 237–246
11. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In: 11th ACM Int. Conf. on Supercomputing, ACM Press (1997) 340–347
12. Cuenca, J., Gimenez, D., Gonzalez, J.: Towards the design of an automatically tuned linear algebra library. In: Proceedings. 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing. (2002) 201–208
13. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing '98, IEEE Computer Society (1998) 211–217
14. Jones, M.T., Plassmann, P.E.: BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical report, Argonne National Laboratory (1995)
15. Kamath, C., Ho, R., Manley, D.: DXML: A high-performance scientific subroutine library. Digital Technical Journal **6** (1994) 44–56
16. Navarro, J.J., García, E., Herrero, J.R.: Data prefetching and multilevel blocking for linear algebra operations. In: Proceedings of the 10th international conference on Supercomputing, ACM Press (1996) 109–116
17. Herrero, J.R., Navarro, J.J.: Automatic benchmarking and optimization of codes: an experience with numerical kernels. In: Proceedings of the 2003 International Conference on Software Engineering Research and Practice, CSREA Press (2003)
18. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR95-035, Department of Computer Science, University of Minnesota (1995)
19. NetLib: (Linear programming problems) http://www.netlib.org/lp/.
20. Frangioni, A.: (Multicommodity Min Cost Flow problems) http://www.di.unipi.it/di/groups/optimize/Data/.