Adapting Linear Algebra Codes to the Memory Hierarchy Using a Hypermatrix Scheme*

José R. Herrero, Juan J. Navarro

Computer Architecture Dept., Univ. Politècnica de Catalunya
Barcelona, (Spain)
{josepr,juanjo}@ac.upc.edu,

Abstract. We present the way in which we adapt data and computations to the underlying memory hierarchy by means of a hierarchical data structure known as hypermatrix. The application of orthogonal block forms produced the best performance for the platforms used.

1 Introduction

In order to obtain efficient codes, computer resources have to be used effectively. The code must have an inner kernel able to make use of the functional units within the processor in an efficient way. On the other hand, data must be accessible in a very short time. This can be accomplished with an adequate usage of the memory hierarchy. In this paper we present the way in which we have obtained fast implementations of two important linear algebra operations: dense Cholesky factorization and matrix multiplication.

We are interested in the development of efficient linear algebra codes on a variety of platforms. For this purpose, we use a hierarchical data structure known as *Hypermatrix* to adapt our code to the target machine.

1.1 Hypermatrix data structure

Our application uses a data structure based on a hypermatrix (HM) scheme [1, 2], in which a matrix is partitioned recursively into blocks of different sizes. A commercial package known as PERMAS uses the hypermatrix structure for solving very large systems of equations [3]. It can solve very large systems out-of-core and can work in parallel. This approach is also related to a variety of recursive/nonlinear data layouts which have been explored elsewhere for both regular [4–7] and irregular [8] applications.

The HM structure consists of N levels of submatrices. In order to have a simple HM data structure which is easy to traverse we have chosen to have blocks at each level which are multiples of the lower levels. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Null

* This work was supported by the Ministerio de Ciencia y Tecnología of Spain (TIN2004-07739-C02-01)

pointers in pointer matrices indicate that the corresponding submatrix does not have any non-zero elements and is therefore unnecessary. This is useful when matrices are sparse. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated on as such. Hypermatrices can be seen as a generalization of quadtrees. The latter partition each matrix precisely into four submatrices [9].

1.2 Motivation

In the past, we have been working on the sparse Cholesky factorization based on the hypermatrix data structure. In the sparse codes it was important to avoid unnecessary operation on zeros. At the same time, using small data submatrices produced many calls to matrix multiplication subroutines resulting in a large overhead. For this reason we created efficient routines which operate on small matrices. By small we mean matrices which fit in cache. We grouped such routines in a library called the Small Matrix Library (SML). Information about the creation of the SML can be found in [10]. Further details on the application of SML to sparse hypermatrix Cholesky can be found in [11].

The hypermatrix data structure, however, can also be used for dense matrix computations. Recently, we have applied a similar approach to work on dense matrices. Let us comment on the case of the MIPS R10000 processor. Figure 1 shows the peak performance of the $C = C - A \times B^t$ matrix multiplication routines in our SML for several matrix dimensions on a MIPS R10000 processor. On small matrices, our code (*mxmts_fix*) outperforms the DGEMM matrix multiplication routine in the vendor BLAS library. We have labeled the latter as *dgemm_nts* to note that B is used transposed while A is not. In addition, it denotes that the result of the multiplication is subtracted from the previous value in matrix C.

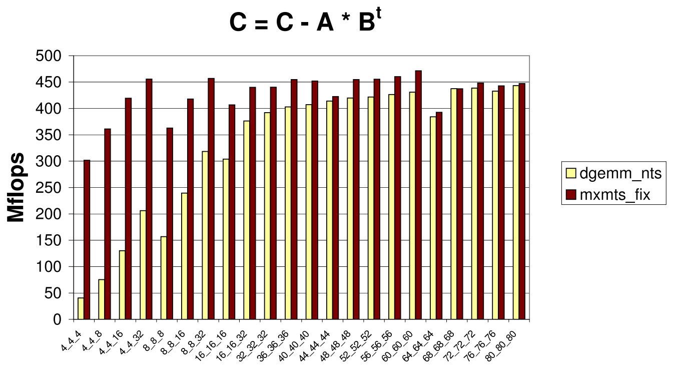

Fig. 1. Peak performance of SML dense matrix multiplication routines on an R10000.

We chose the code corresponding to matrices of size 60×60 as our inner kernel since this was the one providing best peak performance. Then, we used this kernel in two codes based on a hypermatrix data structure: a Cholesky factorization and a matrix multiplication. A data submatrix block size of 60×60 allows for a matrix to be almost permanently in L1 cache. Some conflicts might arise but they should be scarce. A second higher level of 8×8 pointers maps blocks of 480×480 data. This is adequate both for the TLB and second level cache. For the matrix sizes tested those two levels were sufficient to achieve good memory usage. For larger matrices, which required to go out-of-core, or in machines with more levels of caches more pointer levels could be used. The graph on the left part of figure 2 shows the performance obtained on this platform for a dense Cholesky factorization. The graph compares the results obtained by our code (labeled as HM) with those obtained by routine DPOTRF in the vendor library. Both when upper (U) or lower (L) matrices were input to this routine its performance was worse than that of our code. We also tried the matrix multiplication operation $C = C - A * B^T$ since this is the one which takes about 90% of Cholesky factorization. The results can be seen in the right part of figure 2. Our code outperformed the DGEMM matrix multiplication routine in both the vendor and ATLAS libraries. We must note however, that ATLAS was not able to finish its installation process on this platform. Thus, we used a precompiled version of this library which corresponds to an old release of ATLAS. These preliminary results encouraged us to work on dense algorithms based on the hypermatrix data structure.

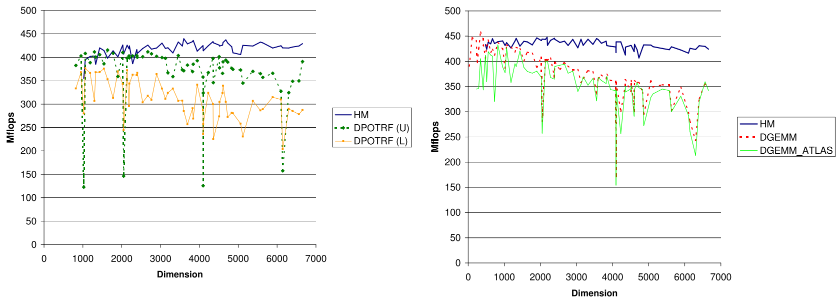

Fig. 2. Performance of dense matrix multiplication and Cholesky factorization on a MIPS R10000 processor.

In this paper we present the extension of our work to dense matrix operations. We will explain how our code is adapted to the underlying memory architecture and is able to obtain good performance.

2 Producing efficient inner kernels

In [10] we introduced a Small Matrix Library (SML). The routines in this library operate efficiently on very small matrices (which fit in level 1 cache). Basically, we try several variants of code with different loop orders and unroll factors. By fixing parameters such as matrix leading dimensions and loop trip counts at compilation time we are able to produce very efficient routines on many platforms. We used such routines to improve our sparse Cholesky factorization code based on the *Hypermatrix* data structure. We have extended our SML with routines which work with sizes larger than the ones used for the sparse codes. We choose as inner kernel the one providing best performance. On MIPS, ALPHA and Itanium2 platforms we could obtain efficient kernels for the matrix multiplication.

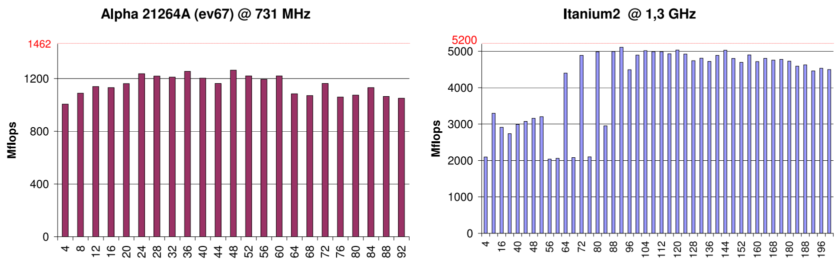

Fig. 3. Peak performance of SML dense matrix multiplication routines.

Figure 3 shows the peak performance of the $C = C - A \times B^t$ matrix multiplication routines in our SML for several matrix dimensions on two different processors. On an Alpha 21264 we chose the routine which works on matrices of size 48×48. On the Intel Itanium2 we chose size 92×92. It is interesting to note that on the Itanium2 the highest performance was obtained for matrix sizes which exceed the capacity of the level 1 (L1) data cache. The following tables show information about caches for each of the three platforms used. Table 1 shows the number of caches and their sizes. Table 2 shows the minimum latency for a floating-point load when it hits in each cache level.

Table 1. Cache sizes

Cache Level	R10000	ALPHA 21264	Itanium2
L1	32 KB	64 KB	16 KB
L2	4 MB	4 MB	256 KB
L3	-	-	3 MB

Table 2. Floating-point load latency (minimum) when load hits in cache.

Cache Level	R10000	ALPHA 21264	Itanium2
L1	3	4	-
L2	8-10	13	**6**
L3	-	-	13

The Itanium2 has three levels of cache. In the first level it has separate instruction and data caches with 16 Kbytes each. Then, it also has a 256 Kbytes L2 cache and an off-chip L3 cache with possible sizes ranging from 1.5 up to 9 MB. The configuration used had a 3 MB L3 cache. The most interesting point to note is the fact that this machine does not cache floating-point data in L1 cache and has low latency when a load hits in its level 2 (L2) cache [12]. The Intel Fortran compiler applied the software pipelining technique automatically for tolerating such latency and produced efficient codes. This is the reason why on this machine the best peak performance for our SML matrix multiplication routines was found for matrices which exceed the L1 data cache size.

3 Exploiting the memory hierarchy

3.1 Number of levels and dimension of each level

We use the hypermatrix data structure to adapt our codes to the underlying memory hierarchy. Our code can be parameterized with the number of pointer levels and block sizes mapped by each level. For the dense codes we follow the next approach: we choose the data submatrix block size according to the results obtained while creating our SML's matrix multiplication routine. The one providing the best performance is taken. As seen in section 2 we do this even when the matrix size is too large to fit in the L1 cache. Then, for the upper levels we choose multiples of the lower levels close to the value $\sqrt{C/2}$, where C is the cache size in double words [13]. We found that, for the machines studied, we only needed two levels of pointers for dense in-core operations.

Figure 4 shows the performance of our hypermatrix multiplication routine on an Itanium2 processor for several matrix dimensions. We have tried several dimensions for the second level of pointers. Values 368 and 460, close to the value $\sqrt{C/2}$, were the best. We tried using a third level with size 736 when the second one has size 368. It didn't produce any benefit since the SML routine was already using efficiently the L2 cache, and the second level of pointers was enough to use the L3 cache adequately. On the Alpha 21264 the size used for data submatrices was 48×48. Then, a second level of pointers in the hypermatrix maps blocks of dimension 480.

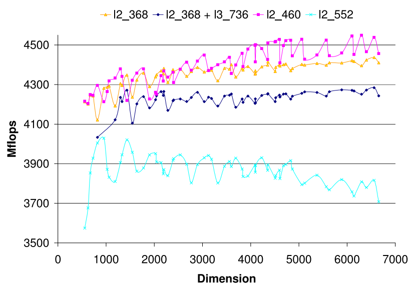

Fig. 4. Performance of dense matrix multiplication and Cholesky factorization on an Intel Itanium 2 processor.

3.2 Orthogonal blocks

In [14] a class of Multilevel Orthogonal Block forms was presented. In that class each level is orthogonal to the previous: they are constructed so that the directions of the blocks of adjacent levels are different. These algorithms exploit the data locality in linear algebra operations when executed in machines with several levels in the memory hierarchy. Figure 5 shows graphically the directions followed by two possible Multilevel Orthogonal Block (MOB) forms.

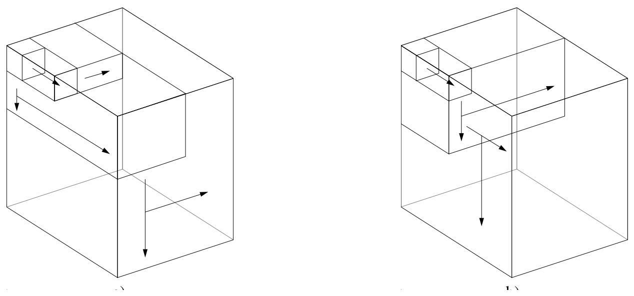

Fig. 5. Two examples of Multilevel Orthogonal Block forms

We have implemented Multilevel Orthogonal Blocks for the different levels in the hypermatrix structure. Figure 6 shows the performance obtained on a matrix multiplication performed on matrices of size 4507 on Itanium2 (left) and Alpha 21264 (right) processors for all combinations of loop orders for two level of pointers in a hypermatrix. All bars to the right of the dashed line correspond to orthogonal forms. Although we eventually use only 2 pointer levels, for hypermatrix multiplication there is an improvement in the performance obtained when the upper level is orthogonal to the lower. In this way the upper level cache is properly used. The performance improvement is modest, but results were always better than those corresponding to non-orthogonal block forms.

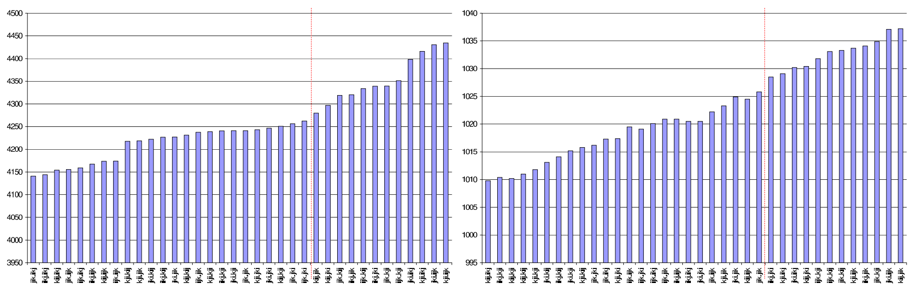

Fig. 6. Performance of HM dense matrix multiplication for several loop orders on Intel Itanium 2 (left) and Alpha 21264 processors (right).

4 Results

We have compared our dense hypermatrix multiplication and Cholesky factorization with ATLAS [15] DGEMM and DPOTRF routines. On the R10000 our code outperformed both the vendor and ATLAS DGEMM and DPOTRF routines. On an ALPHA-21264A ATLAS' installation phase lets the user choose whether to install a hand made code specially designed for this platform (GOTO). In both cases, on this system, ATLAS outperforms our matrix multiplication code. One reason for this can be observed in figure 3. The peak performance of the matrix multiplication routine in our SML was far from the theoretical peak performance on this machine. However, we obtain the same performance as DPOTRF for large matrices (figure 7). On the Itanium2 our performance got close to ATLAS' both for DGEMM and DPOTRF. It was similar to ATLAS for large matrices (figure 8). We must note that, in spite of its name, the ATLAS project is often based on matrix multiplication kernels written in assembly code by hand.

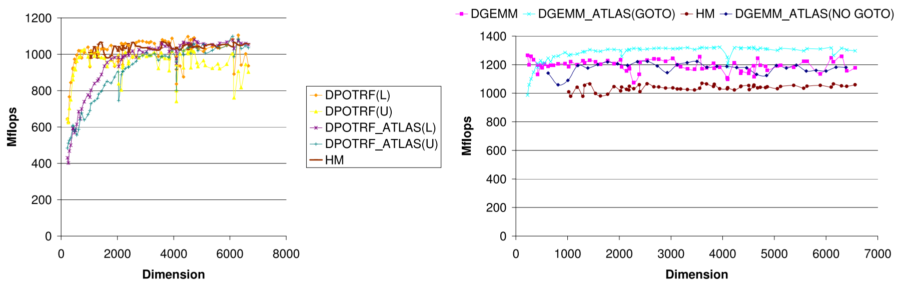

Fig. 7. Performance of dense matrix multiplication and Cholesky factorization on an Alpha 21264 processor.

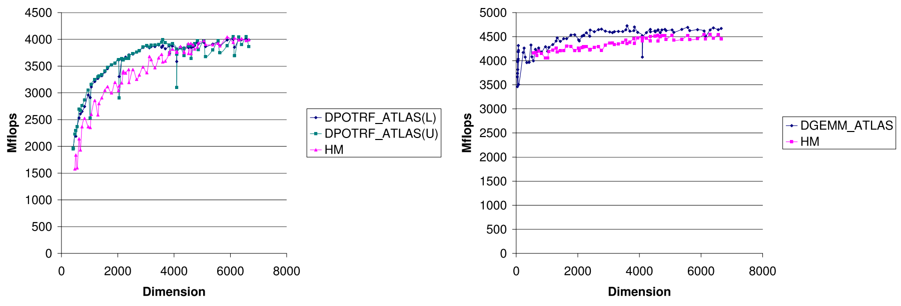

Fig. 8. Performance of dense matrix multiplication and Cholesky factorization on an Intel Itanium 2 processor.

5 Conclusions

It is possible to obtain high performance codes using a high level language and a good optimizing compiler. The inner kernel can target the first level cache. In some cases, it can also target the second level cache. This happens on processors with small level 1 caches and low latency for floating-point loads from the second level cache.

A hypermatrix data structure can be used to adapt the code to the underlying memory hierarchy. For the machines studied and working on dense matrices in-core, two levels of pointers were enough. Going out-of-core or working on machines with more levels of cache memory could benefit from the extension of this scheme to a larger number of levels in the hypermatrix. The use of Multilevel Orthogonal Block forms was always beneficial.

References

1. 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
2. Noor, A., Voigt, S.: Hypermatrix scheme for the STAR–100 computer. Comp. & Struct. **5** (1975) 287–296
3. Ast, M., Fischer, R., Manz, H., Schulz, U.: PERMAS: User's reference manual, INTES publication no. 450, rev.d (1997)
4. Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S., Thottethodi, M.: Nonlinear array layouts for hierarchical memory systems. In: Proceedings of the 13th international conference on Supercomputing, ACM Press (1999) 444–453
5. Frens, J.D., Wise, D.S.: Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not. **32** (1997) 206–216
6. Valsalam, V., Skjellum, A.: A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency and Computation: Practice and Experience **14** (2002) 805–839
7. Wise, D.S.: Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In: Euro-Par 2000,LNCS1900. (2000) 774–783
8. Mellor-Crummey, J., Whalley, D., Kennedy, K.: Improving memory hierarchy performance for irregular applications. In: Proceedings of the 13th international conference on Supercomputing, ACM Press (1999) 425–433
9. Wise, D.S.: Representing matrices as quadtrees for parallel processors. Information Processing Letters **20** (1985) 195–199
10. 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) 701–706
11. Herrero, J.R., Navarro, J.J.: Improving Performance of Hypermatrix Cholesky Factorization. In: Euro-Par 2003,LNCS2790, Springer-Verlag (2003) 461–469
12. Intel: Intel(R) Itanium(R) 2 processor reference manual for software development and optimization (2004)
13. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proceedings of ASPLOS'91. (1991) 67–74
14. Navarro, J.J., Juan, A., Lang, T.: MOB forms: A class of Multilevel Block Algorithms for dense linear algebra operations. In: Proceedings of the 8th International Conference on Supercomputing, ACM Press (1994)
15. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing '98, IEEE Computer Society (1998) 211–217