Exposing Inner Kernels and Block Storage for Fast Parallel Dense Linear Algebra Codes*

José R. Herrero**

Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona, Spain
josepr@ac.upc.edu

Abstract. Efficient execution on processors with multiple cores requires the exploitation of parallelism within the processor. For many dense linear algebra codes this, in turn, requires the efficient execution of codes which operate on relatively small matrices. Efficient implementations of dense Basic Linear Algebra Subroutines exist (BLAS libraries). However, calls to BLAS libraries introduce large overheads when they operate on small matrices. High performance implementations of parallel dense linear algebra codes can be achieved by replacing calls to standard BLAS libraries with calls to specialized inner kernels which work efficiently on small data submatrices.

Key words: Inner kernels, block storage, dense linear algebra, multicore.

1 Introduction

The arrival of multi-core platforms poses a challenge in their programmability. Medium grain parallelism must be exploited within the processor. For that purpose, several frameworks for dynamic scheduling of tasks in multi-core platforms have been introduced recently [1–4]. They handle tasks which are scheduled dynamically. These tasks work on data submatrices which are stored as a sequence of submatrices, typically square blocks. This approach requires efficient operation on small matrices.

1.1 New Data Structures for Parallel Dense Liner Algebra

Matrices have traditionally been stored using canonical storage, either column-major or row-major storage. Such storage, however, can suffer from the lack of locality for certain access patterns. This happens, for instance, when the Cholesky factorization works on the lower triangular matrix and the matrix

* This work was supported by the Ministerio de Educación y Ciencia of Spain (TIN2007-60625).

** Currently on sabbatical leave at Barcelona Supercomputing Center

is stored column-wise (bottom curve in Figure 1). We can also observe that the performance obtained with the traditional approach, where parallelism is exploited only within each iteration of the factorization, does not scale well.

Poor performance is also obtained when working on matrices stored using packed storage. Such format is interesting, however, since it allows for considerable reduction in the amount of space needed to store symmetric and triangular matrices.

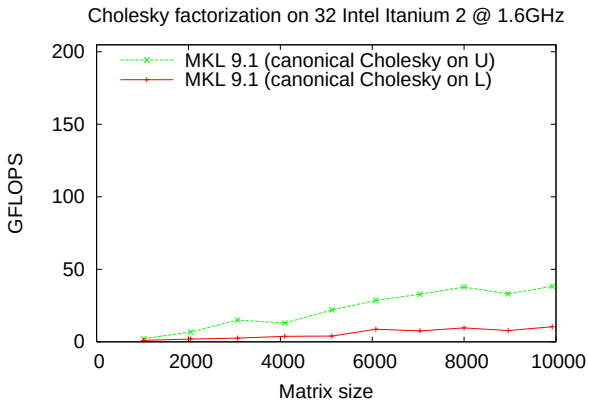

Fig. 1. Parallel Cholesky: Tiled vs Traditional

For these reasons, a lot of work has been developed in order to introduce new data formats based on the storage of matrices by blocks [5–12]. For symmetric and triangular matrices such formats allow for the reduction of the storage space required while keeping performance similar to full storage [13–16].

The use of these new formats has become commonplace since the appearance of multi-core chips and new programming models which address the new scenario [1–4]. These frameworks handle tasks which are scheduled dynamically. In this way different iterations can be overlapped and parallelism is better exploited. These tasks work on data submatrices which are stored as a sequence of submatrices, typically square blocks.

1.2 Overhead

In search for high performance portability, linear algebra codes call Basic Linear Algebra Subroutines (BLAS). Within BLAS, parameters are checked to enforce robustness and submatrices are copied in order to improve locality when the inner kernels are executed. Since they are done for every call, this implies extra overhead is payed when BLAS are called many times to work on relatively small matrices.

As an example, consider the Cholesky factorization with Square Blocked Lower Packed Format (SBPF). Figure 2 illustrates this format. Our code is

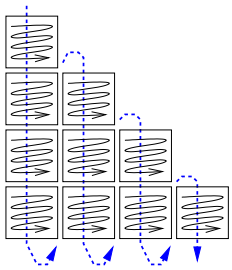

Fig. 2. Square Blocked Lower Packed Format (SBPF).

based on subroutine DPSTRF by F. G. Gustavson, as appears in Fig.10, page 44 in [10].

Figure 3 shows the performance of routines DPOTRF and DPSTRF on an Intel Itanium 2 running at 1.5 GHz and using Goto's library [17]. We can observe that as the block size used in routine DPSTRF is smaller its performance drops. This is due to the overhead introduced by calling BLAS routines multiple times on smaller matrices instead of having a few calls which work on larger submatrices.

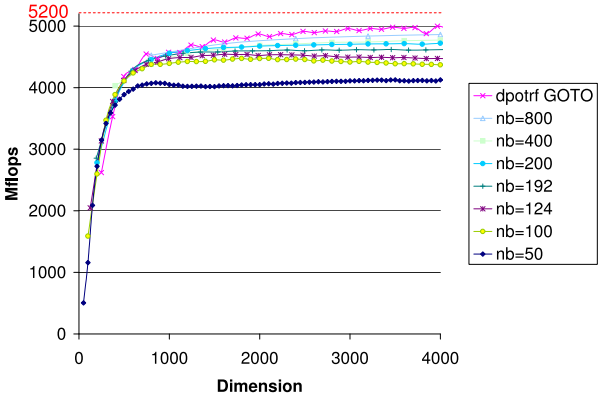

Fig. 3. Performance of Cholesky factorization for several block sizes using Goto's library.

In this paper, we show that the use of specialized inner kernels which have low overhead combined with a framework for dynamic scheduling of tasks can provide excellent performance.

2 Specialized inner kernels

The efficiency of the inner kernels is of paramount importance. For this reason such kernels are usually ad-hoc codes written in assembler for each platform. Our approach, however, is different. In previous papers [18, 19] we presented our work on the creation of a Small Matrix Library (SML): a set of routines, written in Fortran, specialized in the efficient operation on matrices which fit in the first level cache. The advantage of our method lies in the ability to generate very efficient inner kernels by means of an optimizing compiler. Working on regular codes for small matrices, most of the compilers we have used in different platforms create very efficient inner kernels for regular codes such as the matrix-matrix multiplication.

2.1 Creation of inner kernels

Our approach to the creation of high performance specialized kernels consists in:

- Profiling: Optimization efforts must be applied to those parts of code which take up most computation time. In this case, for instance, this means focusing on the optimization of the matrix-matrix multiplication routine first.
- Specialization: Simplify code to do only what is strictly necessary. Simple codes are easier to optimize, both automatically and manually.
- Bottom-up creation of data structure: We drive the creation of the structure from the bottom: the inner kernel fixes the size of the data submatrices [20]. Then the rest of the data structure is produced in conformance. We do this because the performance of the inner kernel has a dramatic influence in the overall performance of the algorithm.

2.2 Practical application

Applying this approach to the Cholesky factorization we first create the inner kernel for the matrix multiplication operation (GEMM). Once the block size is already fixed we apply the same approach to the other operations (TRSM, SYRK, and POTRF). We use the resulting routines, which we store within the SML, as the inner kernels of our general linear algebra codes.

2.3 Low Overhead

When these kernels are called directly from linear algebra codes which store matrices using non-canonical data structures the overhead is very low. This happens because there are no costs associated to copying data and checking certain parameters. Resulting codes can avoid most of the overhead and have high performance.

In [16], Herrero shows that the performance obtained from the resulting sequential Cholesky factorization approaches that of a hand-optimized implementation in which most representative parts of the code are written in assembly code (Goto BLAS). In the following sections we show results when our SML is used in a code parallelized with SMPSs.

3 Parallelization of a Cholesky factorization with SMPSs

We needed a flexible way to parallelize our code and overlap different iterations of the Cholesky factorization. Thus, we have parallelized our code using SMP Superscalar (SMPSs) [4]. SMPSs is a programming environment for multi-core chips and Symmetric Multiprocessors (SMP) in general. It is based on function level parallelism. An SMPSs program is a sequential program annotated with directives which identify functions in the code that are candidates to be run in parallel. These are called tasks and are treated as the unit of parallel computation. Function (task) annotations include information about the parameters and their directionality. In this way, the programmer indicates which functions can be run in parallel and which data they will use.

At execution time SMPSs builds a Task Dependency Graph (TDG) based on data dependencies. The framework schedules tasks dynamically according to the TDG which is a directed acyclic graph. More information about SMPSs can be retrieved from [21].

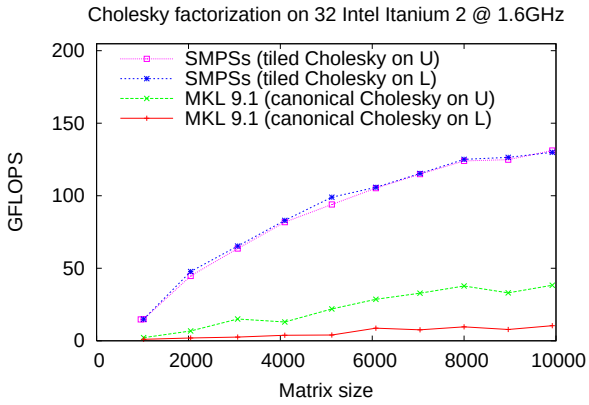

Fig. 4. Parallel Cholesky: Tiled vs Traditional

Figure 4 shows the performance obtained for different versions of the Cholesky factorization on a machine with 16 dual core Intel Itanium 2 processors with a total of 32 cores. Each core runs at 1.6 GHz and can perform four floating point operations per cycle. Thus, the theoretical peak for the machine is 204.8 GFlops.

Our Cholesky factorization uses SMPSs on matrices stored as contiguous square blocks with data stored row-wise within those blocks. We have used routine DPOTRF in version 9.1 of Intel Math Kernel Library (MKL) as a representative of the traditional way to parallelize linear algebra codes in which matrices are stored using canonical format and iterations are not overlapped.

We can observe that the parallelization with SMPSs was very effective. This is mainly due to the possibility to overlap different iterations of the Cholesky factorization, which helps in keeping the cores busy, together with the improved locality of the block storage. Note that this code does not suffer any performance drop when working neither on the lower (L) nor the upper (U) submatrix, while this is often the case in traditional codes which work on canonical storage.

4 Results

Figure 5 shows the performance obtained on 32 Itanium 2 cores running at 1.6 GHz for the dense Cholesky factorization parallelized using SMPSs where the matrix is stored by blocks. Each curve shows the performance obtained when the underlying BLAS library used was either Intel's MKL or our SML. We can observe that the latter provides larger performance due to the good performance and lower overhead of our specialized inner kernels.

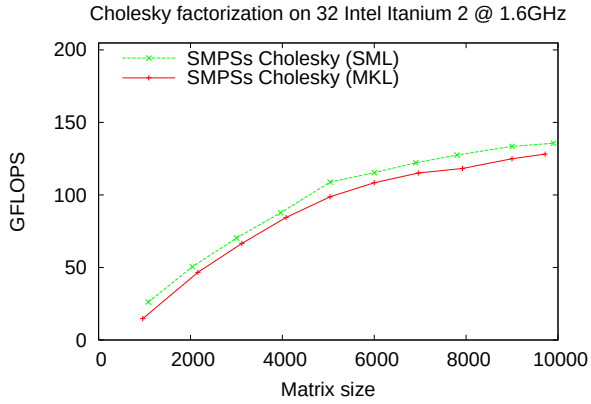

Fig. 5. Performance of Cholesky factorization using MKL and SML.

The following two figures show the influence of tile size (TS) for different matrix dimensions when MKL (Figure 6) and SML (Figure 7) are used. We can observe that the behavior is similar in both cases. Small tile sizes can benefit matrices with dimensions up to around 3000 since they expose more parallelism. For larger matrix dimensions, however, the overhead of operating with such small tiles is too large. Then, larger block sizes suffer less overhead while exposing enough parallelism.

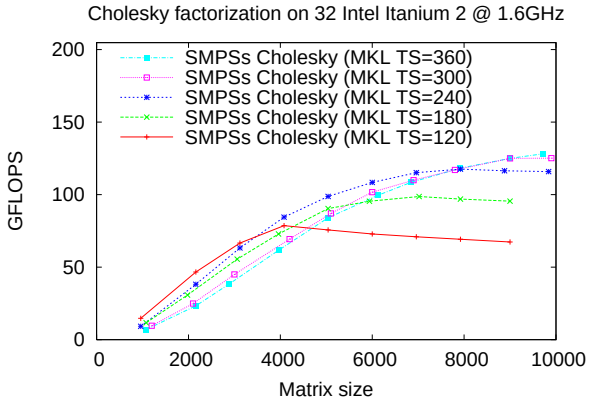

Fig. 6. Performance of Cholesky factorization per tile size using MKL.

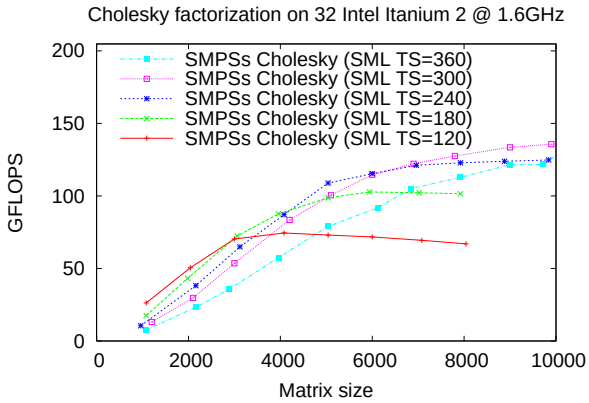

Fig. 7. Performance of Cholesky factorization per tile size using SML.

The following two figures show the scalability of the factorization of a matrix of dimension 6000 when MKL (Figure 8) and SML (Figure 9) are used. Again, results for several tile sizes are presented. We can observe that the behavior is similar. For this matrix dimension small tile sizes should be avoided for any number of cores.

There is a trade-off in the tile size. On the one hand, it has to be small enough to expose parallelism. On the other hand is must be large enough to reduce the overhead introduced by calling BLAS subroutines repetitively, together with the extra overhead introduced by the parallel runtime framework for dynamic scheduling of tasks. The optimal size depends on the matrix dimension and the number of CPUs.

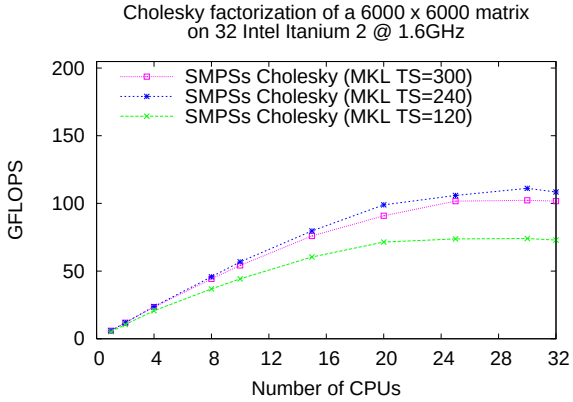

Fig. 8. Scalability of Cholesky factorization per tile size using MKL.

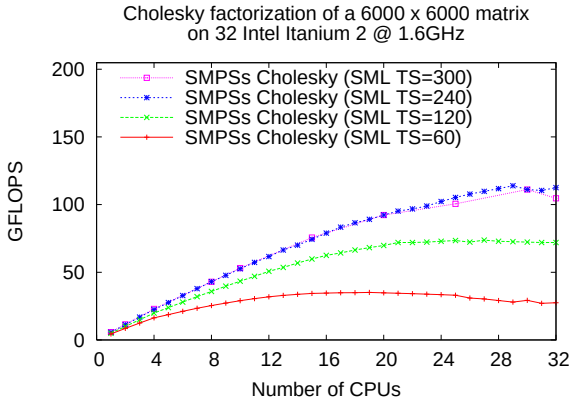

Fig. 9. Scalability of Cholesky factorization per tile size using SML.

5 Conclusions

Our experience with the parallelization of the dense Cholesky factorization with SMPSs has been greatly rewarding. We found that framework to be very intuitive and the parallelization could be done very quickly. In addition, the performance obtained was very good.

The use of non-canonical array layouts suffers the overhead of BLAS routines since these are called many times to work on relatively small data submatrix. Specialization of the inner kernels reduces the overhead and exposes simple and regular codes which a compiler can optimize.

The combination of Square Block Packed format (SBPF) together with the parallelization with SMPSs and the use of the specialized kernels from our Small Matrix Library (SML) allows for high performance with reduced storage. An interesting observation is that this combination can outperform hand-optimized codes written in assembler.

Acknowledgements

Thanks to the Ministerio de Educación y Ciencia of Spain for funding this project (grant TIN2007-60625) and Barcelona Supercomputing Center (BSC) for providing some of the resources used in this work.

References

1. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM Press (2006) 86
2. Chan, E., Zee, F.V., van de Geijn, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Satisfying your dependencies with SuperMatrix. In: IEEE Cluster 2007. (2007) 92–99
3. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, Innovative Computing Laboratory, University of Tennessee Knoxville (September 2007) LAPACK Working Note 191.
4. Perez, J.M., Badia, R.M., Labarta, J.: A flexible and portable programming model for SMP and multi-cores. Technical report, Barcelona Supercomputing Center - Centro Nacional de Supercomputación (June 2007) Technical report 03/2007.
5. McKellar, A.C., E. G. Coffman, J.: Organizing matrices and matrix operations for paged memory systems. Communications of the ACM **12**(3) (1969) 153–165
6. 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 Notices **32**(7) (1997) 206–216
7. Gustavson, F., Henriksson, A., Jonsson, I., Kågström, B.: Recursive blocked data formats and BLAS's for dense linear algebra algorithms. LNCS **1541** (1998) 195–206
8. 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

9. Gustavson, F.G.: New generalized data structures for matrices lead to a variety of high-performance algorithms. In Engquist, B., ed.: Simulation and visualization on the grid: Parallelldatorcentrum, Kungl. Tekniska Högskolan, 7th annual conference, Stockholm, Sweden, December 1999: proceedings. Volume 13 of Lecture Notes in Computational Science and Engineering., Springer-Verlag Inc. (2000) 46–61
10. Gustavson, F.G.: High-performance linear algebra algorithms using new generalized data structures for matrices. IBM J. Res. Dev. **47**(1) (January 2003) 31–55
11. Elmroth, E., Gustavson, F., Jonsson, I., Kågström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review **46**(1) (2004) 3–45
12. Bader, M., Mayer, C.: Cache oblivious matrix operations using Peano curves. In: PARA'06. Volume 4699 of Lecture Notes in Computer Science., Springer-Verlag (June 2006) 521–530
13. Andersen, B.S., Wasniewski, J., Gustavson, F.G.: A recursive formulation of Cholesky factorization of a matrix in packed storage. ACM Transactions on Mathematical Software (TOMS) **27**(2) (2001) 214–244
14. Gunnels, J.A., Gustavson, F.G.: A new array format for symmetric and triangular matrices. In Dongarra, J., Madsen, K., Wasniewski, J., eds.: PARA. Volume 3732 of Lecture Notes in Computer Science., Springer (2004) 247–255
15. Andersen, B.S., Gunnels, J.A., Gustavson, F.G., Reid, J.K., Waśniewski, J.: A fully portable high performance minimal storage hybrid format Cholesky algorithm. ACM Transactions on Mathematical Software **31**(2) (June 2005) 201–227
16. Herrero, J.R.: New data structures for matrices and specialized inner kernels: Low overhead for high performance. In: Int. Conf. on Parallel Processing and Applied Mathematics (PPAM'07). Volume 4967 of Lecture Notes in Computer Science., Springer-Verlag (September 2007) 659–667
17. Goto, K., van de Geijn, R.A.: Anatomy of a high-performance matrix multiplication. ACM Transactions on Mathematical Software **34**(3) (September 2007)
18. Herrero, J.R., Navarro, J.J.: Automatic benchmarking and optimization of codes: an experience with numerical kernels. In: Int. Conf. on Software Engineering Research and Practice, CSREA Press (June 2003) 701–706
19. Herrero, J.R., Navarro, J.J.: Compiler-optimized kernels: An efficient alternative to hand-coded inner kernels. In: Proceedings of the International Conference on Computational Science and its Applications (ICCSA). LNCS 3984. (May 2006) 762–771
20. Herrero, J.R., Navarro, J.J.: Using non-canonical array layouts in dense matrix operations. In: PARA'06. Volume 4699 of Lecture Notes in Computer Science., Springer-Verlag (June 2006) 580–588
21. Barcelona Supercomputing Center: SMP Superscalar (SMPSs) (2007) http://www.bsc.es/plantillaG.php?cat_id=385.