Compiler-Optimized Kernels: An Efficient Alternative to Hand-Coded Inner Kernels*

José R. Herrero and Juan J. Navarro

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

Abstract. The use of highly optimized inner kernels is of paramount importance for obtaining efficient numerical algorithms. Often, such kernels are created by hand. In this paper, however, we present an alternative way to produce efficient matrix multiplication kernels based on a set of simple codes which can be parameterized at compilation time. Using the resulting kernels we have been able to produce high performance sparse and dense linear algebra codes on a variety of platforms.

1 Introduction

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, codes specially optimized for a particular target computer can be written in a high level language [1, 2]. 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. Many linear algebra codes can be implemented in terms of matrix multiplication [3, 4]. Thus, it is important to have efficient matrix multiplication routines at hand. The Fortran implementation of Basic Linear Algebra Subroutines (BLAS) [5] is inefficient, and developing efficient codes for a variety of platforms can take a great effort. Consequently, there have been attempts to produce such codes automatically. A new paradigm was created: Automated Empirical Optimization of Software (AEOS). The goal is to use empirical timings to adapt a package automatically to a new computer architecture. PHiPAC [6] was the first such project. Later, the Automatically Tuned Linear Algebra Software (ATLAS) project [7] appeared, which continues to date. Today, ATLAS-tuned libraries represent one of the most widely used libraries. ATLAS uses many well known optimization techniques developed by both linear algebra and compiler optimization experts. However, and despite its name, a great effort has been applied to produce high performance inner kernels for matrix multiplication using hand-coded routines

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

ICCSA 2006, LNCS 3984, pp. 762–771, 2006.

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

contributed by some experts. Directory `tune/blas/gemm/CASES` within the ATLAS distribution contains about 90 files which are, in most cases, written in assembler, or use some instructions written in assembler to do data prefetching. Often, one or more of these codes outperform the automatically generated codes. The best code is automatically selected as the inner kernel. The use of such hand-made inner kernels has improved significantly the overall performance of ATLAS subroutines on some platforms. Many processors have specific kernels built for them. However, there exist processors for which no such hand-made codes are available. Then, the performance obtained by ATLAS on the latter platforms is comparatively worse than that obtained on the former.

BLAS routines are usually optimized to deal with matrices of different sizes. One amongst several inner codes can be selected at runtime depending on matrix dimensions. This is very convenient for medium and large matrices of different sizes. When small matrices are provided as input, however, the overhead incurred becomes too large and the performance obtained is poor. There are applications which produce a large number of matrix operations on small matrices. For instance, programs which deal with sparse problems or multimedia codes. In those cases, the use of BLAS can be ineffective to provide high performance.

We are interested in obtaining efficient codes when working on both small and large matrices on a variety of platforms. For these purposes we have created a framework which allows us to produce ad hoc routines which can perform matrix multiplications quite efficiently operating on small matrices. Based on these routines, we have produced efficient implementations of both sparse and dense codes for several platforms. In the following sections we present our approach and comment on the results.

2 Generation of Efficient Inner Kernels for Matrix Multiplication

Our 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 sometimes it can yield even better code.

2.1 Taking Advantage of Compiler Optimizations

Compiler technology is a mature field. Many optimization techniques have been developed over the years. A very complete survey of compiler optimization techniques can be found in [8]. The knowledge of the target platform introduced in the compiler by its creators together with the use of well known optimization techniques such as software pipelining, loop unrolling, auto-vectorization, etc., can result in efficient codes which can exploit the processor's resources in an effective way. This is specially true when it is applied to highly regular codes such as a matrix multiplication kernel. Since many platforms have outstanding optimizing compilers available nowadays, we want to let the compiler do the creation of optimized object code for our inner kernels.

2.2 Smoothing the Way to the Compiler

The optimizations performed by the compiler can be favored by certain characteristics of the compiled code. For instance, some loop orders can be more beneficial than others. Some access patterns can be more effective in using memory. Knowing the number of iterations of a loop can help the compiler decide on the application of some techniques such as loop unrolling or software pipelining. We have taken this approach for creating a Small Matrix Library (SML). Basically, we:

- Provide the compiler with as much information as possible regarding matrix leading dimensions and loop trip counts;
- Try several variants of code, with different loop orders or unroll factors.

In addition, in some cases the resulting code can be more efficient if:

- Matrices are aligned;
- All matrices are accessed with stride one;
- Store operations are removed from the inner kernel.

2.3 Creation of a Small Matrix Library

For each desired operation, we have written a set of codes in Fortran. We concentrate on the matrix multiplication since it is one of the most important kernels. Figure 1a shows the performance of different routines for matrix multiplication for several matrix sizes on an Alpha-21164.[1] 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). Thus, we call the vendor BLAS routine *dgemm_nts*.

The 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 for the Alpha-21164). This is due to the overhead of passing a large number of parameters, checking for their correctness, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when the 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 32x32. The latter is the maximum value that allows for a good use of the L1 cache on the Alpha unless tiling techniques are used.

[1] Labels in this figure refer to the three dimensions of the iteration space. However, for all the other figures matrices are assumed to be square and of equal size. In all plots the dashed line at the top shows the theoretical peak performance of the processor.

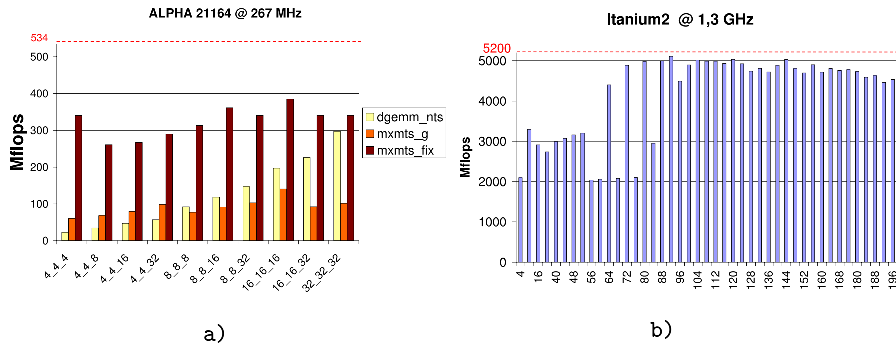

Fig. 1. a) Comparison of performance of different routines for several matrix sizes on an Alpha. **b)** Peak Mflops for SML matrix multiplication routines on an Itanium2.

For a matrix multiplication we have codes with different loop orders (*kji, ijk*, etc.) and unroll factors. We compile each of them using the native compiler and trying several optimization options. For each resulting executable, we automatically execute it and register its performance. These results are kept in a database and finally employed to produce a library using the best combination of parameters. This process is done automatically with a benchmarking tool [9]. 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 useful only for matrix operations conforming to 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. Each one has its own name in the library. In this paper, however, we refer to any of them as *mxmts_fix*. The best loop order and unroll factor obtained for some matrix dimensions on one processor is not necessarily the best for other matrix dimensions or platforms. Choosing a single code for all cases would result in an important performance loss. We also tried feedback driven compilation using the Alpha native compiler but performance either remained the same or even decreased slightly. Results on the R10000 processor are similar to those of the Alpha with the only difference that the *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. 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. Further details can be found in [9].

Figure 1b shows the performance of SML matrix multiplication routines for several matrix sizes on an Itanium2 processor. It is interesting to note that the highest performance was obtained for matrix sizes which exceed the capacity of the level 1 (L1) data cache. The reason for this is that on such processor floating-point data never resides in L1 cache but in the upper levels [10]. Table 1

shows the minimum latency for floating-point loads in each cache level for the Alpha 21264 and the Itanium2 processors. The latency of a floating-point load which hits in the Itanium2 level i cache is similar to that of the Alpha level $i-1$ cache. The Intel Fortran compiler applied the software pipelining technique automatically for tolerating such latency and produced efficient codes. Thus, for our SML matrix multiplication routines on the Itanium2 the best performance was obtained for matrices which exceed the capacity of the L1 data cache.

Table 1. Minimum floating-point load latency when load hits in cache

Cache Level	ALPHA 21264	Itanium2
L1	4	-
L2	13	6
L3	-	13

Originally, we have used this library in a sparse Cholesky factorization based on a hypermatrix data structure [11, 12]. Later we have used it on dense hypermatrix Cholesky factorization and multiplication ($C = C - A \times B^T$).

2.4 Using Hypermatrices to Exploit the Memory Hierarchy

We have used the hypermatrix data structure to adapt our codes to the underlying memory hierarchy. A hypermatrix is a hierarchical data structure with one or more levels of pointer matrices which must be followed to reach a final level of data submatrices. Our code can be parameterized with the number of pointer levels and block sizes mapped by each level.

Sparse Matrices. The application of SML routines to a sparse Cholesky factorization using a hypermatrix scheme has been presented in [13]. Their use improved the factorization efficiency about 12% on a given sparse matrix test suite.

Dense Matrices. The hypermatrix data structure can also be used for dense matrix computations. 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 matrix multiplication routine. The one providing the best performance is taken. As seen above, 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. Such values are known to reduce cache conflicts due to accesses to the other matrices [14].

We found that, for the machines studied, we needed only two levels of pointers for dense operations. On matrix multiplication there is an improvement in the performance obtained when the upper level is orthogonal [15] 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. On a MIPS R10000 processor we found that our code was outperforming both ATLAS and the vendor BLAS on dense Cholesky

factorization and matrix multiplication. 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.

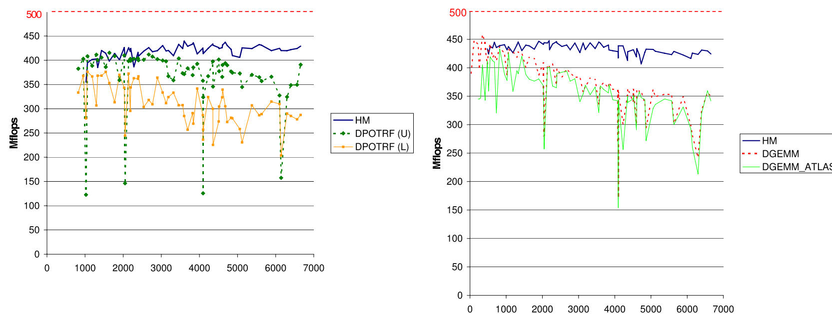

Fig. 2. Performance of dense Cholesky factorization (left) and matrix multiplication (right) on an R10000 processor

We have also compared our dense hypermatrix multiplication and Cholesky factorization with ATLAS DGEMM and DPOTRF routines on two other platforms. On an ALPHA-21264A ATLAS uses hand-made codes specially designed for this platform (Goto BLAS) and outperforms our matrix multiplication code. However, we obtain the same performance as DPOTRF for large matrices. On the Itanium2 our performance got close to ATLAS' both for DGEMM and DPOTRF. It was similar to ATLAS for large matrices. Details can be found in [15].

3 Extension: New Kernels and Matrix Storage

3.1 Generalization of the Matrix Multiplication Codes

We have generalized our matrix multiplication codes to be able to perform the matrix operations $C = \beta C + \alpha op(A) \times op(B)$ where α and β are scalars and op(A) is A or A^t. Actually, we consider $\beta = 1$ since it is more efficient to perform the multiplication of matrix C by β before calling the matrix multiplication kernel

rather than performing this multiplication within it [16]. We allow values of 1 and -1 for α. We parameterize our kernels with preprocessor symbols which are adequately defined at compilation time to determine the type of operation performed. Thus, given a particular loop order and unroll factor, we can produce up to eight kernels: those corresponding to the combinations of transposition of matrices and values of α. The case $C = C + \alpha A^t \times B$ is particularly appealing since it allows accessing all three matrices with stride one. In addition, references to C can be hoisted from the inner loop. Thus, there are no stores in the inner loop. Actually, this is the kernel used in ATLAS. The experiments presented in the rest of the paper refer to this kernel.

3.2 Alignment

Contrary to what happens on other platforms tested, the results obtained for our matrix multiplication routines on an Intel Xeon were initially very poor. This machine allows for vectorization with the SSE2 instruction set [17]. The Intel Fortran compiler can take advantage of them. However, we were getting values around 2300 Mflops, when the theoretical peak for this machine is 4800. By forcing the alignment of matrices to 16, the Intel Fortran Compiler was able to vectorize the code, resulting in a substantial performance increase. The best case is that with $A^t \times B$ which gets a peak performance of 3810 Mflops. Table 2 summarizes these results. On this platform, as on the Itanium2, floating-point loads are not cached in the L1 cache. Thus, the block size automatically chosen (104) targets the L2 cache.

Table 2. Peak Mflops of inner kernel on a Pentium Xeon Northwood

	$A \times B^t$	$A^t \times B$
No align	3334	3220
Align	3457	3810

3.3 Adapting the Codes to the Memory Hierarchy

The use of hypermatrices on dense operations is inefficient both in terms of storage (keeping pointer matrices) and computation (following pointers and recursing through the data structure). The pointer matrices can be avoided for dense operations, keeping only data submatrices. These matrices can be accessed calculating their relative position. We store matrices as a set of submatrices in block major format. We call this scheme TDL in the graphs to refer to the two dimensional layout of data submatrices. In order to use the memory hierarchy we have codes which allow us to do tiling. We have written a code generator which can be used to produce codes with permutations of loop orders. Some of them can behave better than others [18, 19]. For the time being, however, the selection of one of these codes is not automated. We just run some executions and choose the one producing the best results. Block (tile) sizes are again chosen to be multiples of the lower levels close to the value $\sqrt{C/2}$, where C is the cache size in double words [14].

3.4 Results

We present results for matrix multiplication on three platforms. Each of them shows the results of DGEMM in ATLAS, Goto or the vendor BLAS, and TDL using our SML. Goto BLAS [20] are known to obtain excellent performance. They are coded in assembler and targeted to each particular platform. The dashed line at the top of each plot shows the theoretical peak performance of the processor.

For the Intel machines (figure 3) we have included the Mflops obtained with a version of the ATLAS library where the hand-made codes were not enabled at ATLAS installation time. We refer to this code in the graphs as 'nc ATLAS'. We can observe that in both cases ATLAS performance drops heavily. TDL with SML kernels obtain performance close to that of ATLAS on the Pentium 4 Xeon, similar to ATLAS on the Itanium2, and better than ATLAS on the Power4. For the latter we show the Mflops obtained by the vendor DGEMM routine which outperform both ATLAS and TDL (figure 4).

Results for TDL assume matrices already stored in block major format. Although new matrix storage formats have been proposed [21, 22, 23, 24] the matrix will probably need to be transformed from column major order into block major order. We have measured the time necessary to create the three matrices used

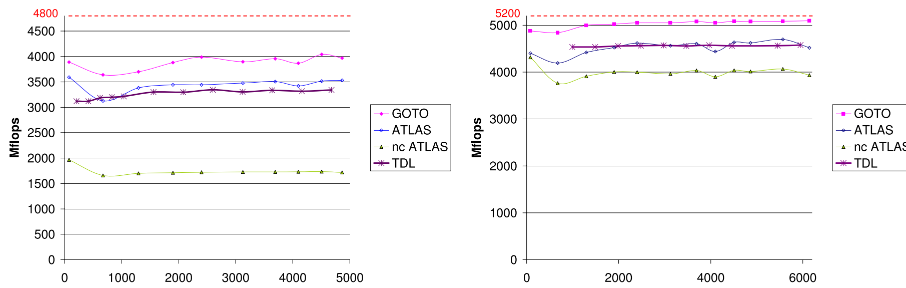

Fig. 3. Performance of dense matrix multiplication on an Intel Pentium 4 Xeon (left) and an Intel Itanium 2 processor (right)

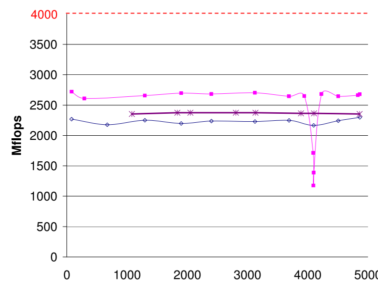

Fig. 4. Performance of dense matrix multiplication on a Power4 processor

in a matrix multiplication. Taking that into account, the performance of TDL drops by about 10% for small matrices, and as low as 1% for the largest matrices tested. The reason for this is that the cost of this transformation is $O(N^2)$ while for the multiplication the cost is $O(N^3)$.

4 Conclusions

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. 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, in certain cases, it can yield even better code. We have taken this approach for creating our Small Matrix Library (SML). It is important to have data properly aligned and accessed with stride one. Loop orders which allow for the reduction of references to matrices in the inner loop favor performance. Situations where stores can be removed from the inner loop are specially beneficial. The inner kernel can target the first or the second level cache. The latter happens on processors which do not cache floating-point data in the level 1 cache.

The use of a simple two dimensional layout of data submatrices (TDL) avoids the overhead of a hypermatrix data structure, in which pointers have to be followed recursively, resulting in better performance.

Although our approach is not fully automatic as yet, we have been able to test it on several platforms and for many matrix sizes. We have shown that the results obtained on large dense matrices are close to those of hand-written matrix multiplication routines and can outperform ATLAS on some platforms. In addition, our approach can be used to produce efficient kernels for smaller matrix kernels which has been successfully used in a sparse Cholesky factorization. We believe this could also be useful in other types of applications such as multimedia codes.

References

1. Kamath, C., Ho, R., Manley, D.: DXML: A high-performance scientific subroutine library. Digital Technical Journal **6** (1994) 44–56
2. 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
3. Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Croz, J.D., Hammarling, S., Demmel, J., Bischof, C., Sorensen, D.: LAPACK: A portable linear algebra library for high-performance computers. In: Proc. of Supercomputing '90, IEEE Press (1990) 1–10
4. Kågström, B., Ling, P., van Loan, C.: Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. ACM Transactions on Mathematical Software (TOMS) **24** (1998) 268–302

5. Dongarra, J.J., Du Croz, J., Duff, I.S., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Software **16** (1990) 1–17
6. 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
7. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing '98, IEEE Computer Society (1998) 211–217
8. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys **26** (1994) 345–420
9. 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 (2003) 701–706
10. Intel: Intel(R) Itanium(R) 2 processor reference manual for software development and optimization (2004)
11. 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
12. Noor, A., Voigt, S.: Hypermatrix scheme for the STAR–100 computer. Comp. & Struct. **5** (1975) 287–296
13. Herrero, J.R., Navarro, J.J.: Improving Performance of Hypermatrix Cholesky Factorization. In: Euro-Par 2003,LNCS2790, Springer-Verlag (2003) 461–469
14. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proceedings of ASPLOS'91. (1991) 67–74
15. Herrero, J.R., Navarro, J.J.: Adapting linear algebra codes to the memory hierarchy using a hypermatrix scheme. In: Int. Conf. on Parallel Processing and Applied Mathematics. (2005)
16. Daydé, M.J., Duff, I.S.: The use of computational kernels in full and sparse linear solvers, efficient code design on high-performance RISC processors. In: VECPAR. (1996) 108–139
17. SSE2: (Streaming SIMD Extensions 2 for the Pentium 4 processor) http://www.intel.com/software/products/college/ia32/sse2.
18. Gunnels, J.A., Henry, G., van de Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In: International Conference on Computational Science (1). (2001) 51–60
19. 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)
20. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Univ. of Texas at Austin (2002)
21. Gustavson, F.G.: New generalized data structures for matrices lead to a variety of high performance algorithms. In: PPAM. (2001) 418–436
22. 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** (2001) 214–244
23. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast parallel matrix multiplication. In: Proc. of the 11th annual ACM symposium on Parallel algorithms and architectures, ACM Press (1999) 222–231
24. 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