Level-3 Cholesky Kernel Subroutine of a Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm

Fred G. Gustavson
IBM T.J. Watson Research Center
and
Jerzy Waśniewski
Department of Informatics and Mathematical Modelling
Technical University of Denmark
and
Jack J. Dongarra
University of Tennessee, Oak Ridge National Laboratory and University of Manchester

Updated: September 18, 2008 by Jerzy

The TOMS paper "A Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm" by Andersen, Gunnels, Gustavson, Reid, and Waśniewski, used a level 3 Cholesky kernel subroutine instead of level 2 LAPACK routine POTF2. We discuss the merits of this approach and show that its performance over POTRF is considerably improved on a variety of common platforms when POTRF is solely restricted to calling POTF2.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra – Linear Systems (symmetric and Hermitian); G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms, BLAS, Performance

Additional Key Words and Phrases: real symmetric matrices, complex Hermitian matrices, positive definite matrices, Cholesky factorization and solution, recursive algorithms, novel packed matrix data structures.

1. INTRODUCTION

We consider the Cholesky factorization of a symmetric positive definite matrix where the data has been stored using Block Packed Hybrid Format (BPHF) [Andersen et al. 2005; Gustavson et al. 2007]. We will examine the case where the matrix A is factored into LL^T, where L is a lower triangular matrix. See also

Authors' addresses: F.G. Gustavson, IBM T.J. Watson Research Center, Yorktown Heights, NY-10598, USA, email: fg2@us.ibm.com; J. Waśniewski, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, Building 321, DK-2800 Kongens Lyngby, Denmark, email: jw@imm.dtu.dk; J.J. Dongarra, Electrical Engineering and Computer Science Department, University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN 37996-3450, USA, email: dongarra@cs.utk.edu.

1a. Lower Packed Format

0									
1	10								
2	11	19							
3	12	20	27						
4	13	21	28	34					
5	14	22	29	35	40				
6	15	23	30	36	41	45			
7	16	24	31	37	42	46	49		
8	17	25	32	38	43	47	50	52	
9	18	26	33	39	44	48	51	53	54

1b. Lower Blocked Hybrid Format

0									
1	2								
3	4	5							
6	7	8	27						
9	10	11	28	29					
12	13	14	30	31	32				
15	16	17	33	34	35	45			
18	19	20	36	37	38	46	47		
21	22	23	39	40	41	48	49	50	
24	25	26	42	43	44	51	52	53	54

Fig. 1. Lower Packed and Blocked Hybrid Formats

do $j = 1, l$! $l = \lceil n/nb \rceil$
    do $k = 1, j-1$
        $A_{jj} = A_{jj} - L_{jk}L_{jk}^T$! Call of Level-3 BLAS _SYRK
        do $i = j+1, l$
            $A_{ij} = A_{ij} - L_{ik}L_{jk}^T$! Call of Level-3 BLAS _GEMM
        end do
    end do
    $L_{jj}L_{jj}^T = A_{jj}$! Call of LAPACK subroutine _POTRF
    do $i = j+1, l$
        $L_{ij}L_{jj}^T = A_{ij}$! Call of Level-3 BLAS _TRSM
    end do
end do

Fig. 2. LL^T Implementation for Lower Blocked Hybrid Format. The BLAS calls take the forms `_SYRK('U','T',...)`, `_GEMM('T','N',...)`, `_POTRF('U',...)`, and `_TRSM('L','U','T',...)`.

papers [Herrero and Navarro 2006; Herrero 2007]. We will show that the implementation can be structured to use matrix-matrix operations and take advantage of the Level 3 BLAS and thereby achieving high performance. This implementation has a parallel in the LAPACK routine, which is not based on Level 3 BLAS operations [Gustavson 2003]. A form of register blocking is used for the Level-3 kernel routines of this paper.

The performance numbers presented in Section 3 bear out that the Level-3 based factorization kernels for Cholesky improves performance over the traditional Level-2 routines used by LAPACK. Put another way the use of square block (SB) format allows one to utilize Level-3 BLAS kernels. Hence, one can rewrite the LAPACK implementation which uses a standard row column format with Level-3 BLAS to using SB format with Level-3 BLAS kernels. This paper suggests a change direction for LAPACK software in the multi-core era of computing. This is the main point of our paper.

1.1 Introduction to BPHF

In designing the Level-3 BLAS, [Dongarra et al. 1990] the authors did not specify packed storage schemes for symmetric, Hermitian or triangular matrices. The reasoning given at the time was 'such storage schemes do not seem to lend themselves

```
do i = 1, l                                   ! l = ⌈n/nb⌉
    A_ii = A_ii − Σ_{k=1}^{i−1} (U_ki^T U_ki)         ! Call of Level-3 BLAS _SYRK
    U_ii^T U_ii = A_ii                                 ! Call of LAPACK subroutine _POTF2
    A_ij = A_ij − Σ_{k=1}^{i−1} (U_ki^T U_kj), ∀j > i  ! Single call of Level-3 BLAS _GEMM
    U_ii^T U_ij = A_ij, ∀j > i                         ! Single call of Level-3 BLAS _TRSM
end do
```

Fig. 3. LAPACK Cholesky Implementation for Upper Full Format. The BLAS calls take the forms _SYRK('U','T',...), _POTF2('U',...), _GEMM('T','N',...), and _TRSM('L','U','T',...).

to partitioning into blocks ... Also packed storage is required much less with large memory machines available today'. The BPHF algorithm demonstrates that packing is possible without loss of performance. While memories continue to get larger, the problems that are solved get larger too and there will always be an advantage in saving storage.

We pack the matrix by using a blocked hybrid format in which each block is held contiguously in memory. This usually avoids the data copies, see [Gustavson et al. 2007], that are inevitable when Level-3 BLAS are applied to matrices held conventionally in rectangular arrays. Note, too, that many data copies may be needed for the same submatrix in the course of a Cholesky factorization [Gustavson 1997; Gustavson 2003; Gustavson et al. 2007].

We show an example of standard lower packed format in Fig. 1a, with blocks of size 3 superimposed. Fig. 1 shows where each matrix element is stored within the array that holds it. It is apparent that the blocks of Fig. 1a are not suitable for passing to the BLAS since the stride between elements of a row is not uniform. We therefore rearrange each trapezoidal block column so that it is stored by blocks with each block in row-major order, as illustrated in Fig. 1b. If the matrix order is n and the block size is nb, this rearrangement may be performed efficiently in place with the aid of a buffer of size $n \times nb$. Unless the order is an integer multiple of the block size, the final block will be shorter than the rest. We further assume that the block size is chosen so that a block fits comfortably in level-1 cache.

We factorize the matrix A as defined in Fig. 1b using the algorithm defined in Fig. 2. This is standard blocked based algorithm similar to the LAPACK algorithm and it is also described more fully in [Andersen et al. 2005; Gustavson 2003].

2. THE KERNEL ROUTINE

Each of the computation lines in the Fig. 2 can be implemented by a single call of a Level-3 BLAS [Dongarra et al. 1990] or LAPACK [Anderson et al. 1999] subroutine POTRF. However, we found it better to make a direct call to an equivalent 'kernel' routine that is fast because it has been specially written for matrices that are held in contiguous memory and are of a form and size that permits efficient use of the level-1 cache. Please compare Fig. 3 and 4; see also, [Andersen et al. 2005; Gustavson 2003]

Another possibility is to use a block algorithm with a very small block size kb, designed to fit in registers. To avoid procedure call overheads for a very small computations, we replace all calls to BLAS by in-line code. This means that it is not advantageous to perform a whole block row of _GEMM updates at once and a

do $i = 1, l$! $l = \lceil n/kb \rceil$
  $A_{ii} = A_{ii} - \sum_{k=1}^{i-1}(U_{ki}^T U_{ki})$! Like Level-3 BLAS _SYRK
  $U_{ii}^T U_{ii} = A_{ii}$! Cholesky factorization of block
  do $j = i + 1, n$
    $A_{ij} = A_{ij} - \sum_{k=1}^{i-1}(U_{ki}^T U_{kj})$! Like Level-3 BLAS _GEMM
    $U_{ii}^T U_{ij} = A_{ij}$! Like Level-3 BLAS _TRSM
  end do
end do

Fig. 4. Cholesky Kernel Implementation for Upper Full Format.

```
DO k = 1, ii - 1
   aki = a(k,ii)
   akj = a(k,jj)
   t11 = t11 - aki*akj
   aki1 = a(k,ii+1)
   t21 = t21 - aki1*akj
   akj1 = a(k,jj+1)
   t12 = t12 - aki*akj1
   t22 = t22 - aki1*akj1
END DO
```

Fig. 5. Code corresponding to _GEMM.

whole block row of _TRSM updates at once (see last two lines of the loop in Fig. 3). This leads to the algorithm summarized in Fig. 4.

We have found the tiny block size $kp = 2$ to be suitable. The key loop is the one that corresponds to _GEMM. For this, the code of Fig. 5 is suitable. The block $A_{i,j}$ is held in the four variables, `t11`, `t12`, `t21`, and `t22`. We reference the underlying array directly, with $A_{i,j}$ held from `a(ii,jj)`. It may be seen that a total of 8 local variables are involved, which hopefully the compiler will arrange to be held in registers. The loop involves 4 memory accesses and 8 floating-point operations.

We also tried accumulating a block of size 1×4 in the inner _GEMM loop of the unblocked code ($kp = 1$). Each execution of the loop involves the same number of floating-point operations (8) as for the 2×2 case, but requires 5 reals to be loaded from cache instead of 4. We were not surprised to find that it ran slower on our platforms. However, on Intel, ATLAS [Whaley et al. 2000] uses a 1×4 kernel with extreme unrolling with good effect. Thus we were somewhat surprised that 1×4 unrolling also did poorly on our Intel platform.

On some processors, faster execution is possible by having an inner _GEMM loop that updates $A_{i,j}$ and $A_{i,j+1}$. The variables `aki` and `aki1` need only be loaded once, so we now have 6 memory accesses and 16 floating-point operations and need 14 local variables, hopefully in registers.

We found that this algorithm gave very good performance (see next section). Our implementation of this kernel is available in the TOMS Algorithm paper [Gustavson et al. 2007], but alternatives should be considered. Further, every computer hardware vendor is interested in having good and well-tuned software libraries.

We recommend that all the alternatives of the BPHF paper [Andersen et al. 2005] be compared. Our kernel routine is available if the user is not able to perform such a comparison procedure or has no time for it. Finally, note that LAPACK [Anderson et al. 1999], AtlasBLAS [Whaley et al. 2000], GotoBLAS [Goto and van de Geijn 2008a; Goto and van de Geijn 2008b], and the development of computer vendor software are ongoing activities. The implementation that is the slowest today might be the fastest tomorrow.

3. PERFORMANCE

Mat ord	Vendor lap	Recursive lap	dpotf2 lap	dpotf2 ker	2x2 w. fma 7 flops lap	2x2 w. fma 7 flops ker	1x4 8 flops lap	1x4 8 flops ker	2x4 16 flops lap	2x4 16 flops ker	2x2 6 flops lap	2x2 6 flops ker
1	2	3	4	5	6	7	8	9	10	11	12	13
Newton: SUN UltraSPARC IV+, 1800 MHz, dual-core, Sunperf BLAS												
40	759	547	490	437	1239	1257	1004	1012	1515	**1518**	1299	1317
64	1101	1086	738	739	1563	1562	1291	1295	1940	**1952**	1646	1650
72	1183	978	959	826	1509	1626	1330	1364	1764	**2047**	1582	1733
100	1264	1317	1228	1094	1610	1838	1505	1541	1729	**2291**	1641	1954
Freke: SGI - Intel Itanium2, 1.5 GHz/6, SGI BLAS												
40	396	652	399	408	1493	1612	1613	1769	2045	**2298**	1511	1629
64	623	1206	624	631	2044	2097	1974	2027	2723	**2824**	2065	2116
72	800	1367	797	684	2258	2303	2595	2877	2945	**3424**	2266	2323
100	1341	1906	1317	840	2790	2648	2985	3491	3238	**4051**	2796	2668
Huge: IBM Power6, 4.7 GHz, DualCore, ESSL BLAS												
40	5716	1796	1240	1189	3620	3577	2914	4002	4377	**5903**	3508	4743
64	8021	3482	1265	1293	5905	6019	5426	5493	7515	**7700**	6011	5907
72	8289	3866	1622	1578	5545	5178	5205	4601	6416	**6503**	5577	4841
100	9371	5423	3006	2207	7018	5938	6699	6639	7632	**8760**	7050	6487
Battle: 2×Intel Xeon, CPU @ 1.6 GHz, Atlas BLAS												
40	333	355	455	461	818	840	781	799	806	815	824	**846**
64	489	483	614	620	1015	1022	996	1005	1003	1002	1071	**1077**
72	616	627	648	700	914	1100	898	1105	903	1090	936	**1163**
100	883	904	883	801	1093	1191	1080	1248	1081	1210	1110	**1284**
Nala: 2×AMD Dual Core Opteron 265 @ 1.8 GHz, Atlas BLAS												
40	350	370	409	397	731	696	812	**784**	773	741	783	736
64	552	539	552	544	925	909	1075	**1064**	968	959	944	987
72	568	570	601	568	871	909	966	**1065**	901	964	926	992
100	710	686	759	651	942	1037	972	**1231**	949	1093	950	1114
Zook: 4×Intel Xeon Quad Core E7340 @ 2.4 GHz, Atlas BLAS												
40	497	515	842	844	1380	1451	1279	1294	1487	**1502**	1416	1412
64	713	710	1143	1146	1675	1674	1565	1565	1837	**1841**	1674	1674
72	863	874	1203	1402	1522	1996	1492	1877	1633	**2195**	1527	1996
100	1232	1234	1327	1696	1533	2294	1503	2160	1563	**2625**	1530	2285
1	2	3	4	5	6	7	8	9	10	11	12	13

Table 1. Performance in Mflop/s of the Kernel Cholesky Algorithm. Comparison between different computers and different versions of subroutines.

We consider orders 40, 64, 72, and 100 since these will typically allow the computation to fit comfortably in level-1 cache.

Table 1 contain comparison numbers in Mflop/s. There are results for six computers inside the table: SUN UltraSPARC IV+, SGI - Intel Itanium2, IBM Power6, Intel Xeon, AMD Dual Core Opteron, and Intel Xeon Quad Core.

The table has 13 columns. The first column shows the matrix order. The second column contains results of the vendor Cholesky routine of DPOTRF, the third one has results of the Recursive Algorithm [Andersen et al. 2001]. The columns from 4th to 13th contain results of Cholesky routine using one of the kernel routine, and results when the kernel is called directly instate DPOTRF. There are five kernel routines:

(1) The LAPACK kernel routine DPOTF2: The column 4th has results of DPOTRF, and column 5th of DPOTF2 (compiled routines).

(2) The 2×2 blocking kernel routine specialized for the operation FMA ($a \times b + c$) using 7 floating point (fp) registers (this 2×2 blocking kernel routine replaces the DPOTF2): The performance results are stored in columns 6th and 7th respectively.

(3) The 1×4 blocking kernel routine only optimized for the case $mod(n, 4)$ (n is the matrix order) using 8 fp registers (this 1×4 blocking kernel routine replaces the DPOTF2): the results are stored in 8th and 9th columns respectively.

(4) The 2×4 blocking kernel routine using 16 fp registers (this 2×4 blocking kernel routine replaces the DPOTF2): the results are stored in 10th and 11th columns respectively.

(5) The 2×2 (see Fig. 5) blocking kernel routine not specialized for the operation FMA using 6 floating point (fp) registers (this 2×2 blocking kernel routine replaces the DPOTF2): The performance results are stored in columns 12th and 13th respectively.

It may be seen that the blocked code with blocks of sizes 2×4 (column number 11) is remarkably successful for Sun (Newton), SGI (Freke), IBM (Huge) and quad core Xeon (Zook) computers. In all these four cases, it significantly outperforms the compiled LAPACK code and the recursive algorithm. It outperforms the vendor's optimized codes except on the IBM (Huge) platform. The kernel 2×2 (not prepared for the FMA operation; column number 13) is superior for the Battle computer. The kernel 1×4 (column number 9) is superior for the duel core AMD (Nala) computer. All the superior results are colored in red.

For further details please see the sections 6 and 7.1 of [Andersen et al. 2005]. The code of all kernel subroutines, except _POTF2, is available in [Gustavson et al. 2007]. The code of _POTF2 is from the LAPACK package [Anderson et al. 1999].

4. SUMMARY AND CONCLUSIONS

(1) The purpose of our paper is to promote the new **Block Packed Data Format** storage or variants thereof. These variants of BPHF algorithm use slightly more than $n \times (n+1)/2$ matrix elements of computer memory and always work not slower than the full format data storage algorithms. The full format algorithms store $(n-1) \times n/2$ matrix elements in the computer memory but never reference them.

(2) This paper isn't an original paper. It contains the results and part of the text from the TOMS paper [Andersen et al. 2005]. However, its results are currently so important that we think we should again repeat them.

The performance results in Table 1 are replaced with numbers obtained on more novel computers.

5. ACKNOWLEDGMENTS

The results in this paper were obtained on six computers, an IBM, SGI, SUN, Core2 and two Intel-Xeon. The IBM and SGI machines belong to the Center for Scientific Computing at Aarhus, the SUN machines to the Danish Technical University, and the Core2 and Intel-Xeon machines belong to the Innovative Computing Laboratory of the University of Tennessee at Knoxville, USA. We would like to thank Bernd Dammann for consulting on the SUN system; Niels Carl W. Hansen for consulting on the IBM and SGI systems; and Paul Peltz and Barry Britt for consulting on the Core2 and Intel-Xeon systems.

REFERENCES

Andersen, B. S., Gustavson, F. G., Reid, J. K., and Waśniewski, J. 2005. A Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm. *ACM Transactions on Mathematical Software 31*, 201–227.

Andersen, B. S., Gustavson, F. G., and Waśniewski, J. 2001. A Recursive Formulation of Cholesky Facorization of a Matrix in Packed Storage. *ACM Transactions on Mathematical Software 27*, 2 (Jun), 214–244.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide* (Third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 16*, 1 (March), 18–28.

Goto, K. and van de Geijn, R. 2008a. High performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software 35*, 1, 12.

Goto, K. and van de Geijn, R. A. 2008b. GotoBLAS Library. http://doi.acm.org/10.1145/1356052.1356053. The University of Texas at Austin, Austin, TX, USA.

Gustavson, F. G. 1997. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development 41*, 6 (November), 737–755.

Gustavson, F. G. 2003. High Performance Linear Algebra Algorithms using New Generalized Data Structures for Matrices. *IBM Journal of Research and Development 47*, 1 (January), 823–849.

Gustavson, F. G., Gunnels, J., and Sexton, J. 2007. Minimal Data Copy for Dense Linear Algebra Factorization. In *Applied Parallel Computing, State of the Art in Scientific Computing, PARA 2006*, Volume LNCS 4699 (Springer-Verlag, Berlin Heidelberg, 2007), pp. 540–549. Springer.

Gustavson, F. G., Reid, J. K., and Waśniewski, J. 2007. Algorithm 865: Fortran 95 Subroutines for Cholesky Factorization in Blocked Hybrid Format. *ACM Transactions on Mathematical Software 33*, 1 (March), 5.

Herrero, J. R. 2007. 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* (Sept. 2007), pp. 659–667. springer.

Herrero, J. R. and Navarro, J. J. 2006. 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), pp. 762–771.

Whaley, R. C., Petitet, A., and Dongarra, J. J. 2000. ATLAS: Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/. University of Tennessee at Knoxville, Tennessee, USA.