Level-3 Cholesky Factorization Routines Improve Performance of Many Cholesky Algorithms

FRED G. GUSTAVSON and JERZY WAŚNIEWSKI, IBM T.J. Watson Research Center, Emeritus, and Umeå University, Technical University of Denmark
JACK J. DONGARRA, University of Tennessee, Oak Ridge National Laboratory and University of Manchester
JOSÉ R. HERRERO, Universitat Politècnica de Catalunya, BarcelonaTech
JULIEN LANGOU, University of Colorado at Denver

Four routines called DPOTF3i, `i = a,b,c,d`, are presented. DPOTF3i are a novel type of level-3 BLAS for use by BPF (*B*locked *P*acked *F*ormat) Cholesky factorization and LAPACK routine DPOTRF. Performance of routines DPOTF3i are still increasing when the performance of Level-2 routine DPOTF2 of LAPACK starts decreasing. This is our main result and it implies, due to the use of larger block size nb, that DGEMM, DSYRK, and DTRSM performance also increases! The four DPOTF3i routines use simple register blocking. Different platforms have different numbers of registers. Thus, our four routines have different register blocking sizes.

BPF is introduced. LAPACK routines for _POTRF and _PPTRF using BPF instead of full and packed format are shown to be trivial modifications of LAPACK _POTRF source codes. We call these codes _BPTRF. There are two variants of BPF: lower and upper. Upper BPF is "identical" to *S*quare *B*lock *P*acked *F*ormat (SBPF). "LAPACK" implementations on multicore processors use SBPF. Lower BPF is less efficient than upper BPF. Vector inplace transposition converts lower BPF to upper BPF very efficiently. Corroborating performance results for DPOTF3i versus DPOTF2 on a variety of common platforms are given for $n \approx nb$ as well as results for large n comparing DBPTRF versus DPOTRF.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear Systems*; G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms, Performance

Additional Key Words and Phrases: LAPACK, real symmetric matrices, complex Hermitian matrices, positive definite matrices, Cholesky factorization and solution, novel blocked packed matrix data structures, inplace transposition, Cache Blocking, BLAS

The authors thank the Ministerio de Educación y Ciencia of Spain for funding project grant TIN2007-60625.
Authors' addresses: F. G. Gustavson, 3 Welsh Court, East Brunswick, NJ-08816, USA; email: fg2935@hotmail.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; J. R. Herrero, Computer Architecture Department, Universitat Politècnica de Catalunya, BarcelonaTech, C/ Jordi Girona 1-3, C6-206, 08034 Barcelona, Spain; email: josepr@ac.upc.edu; J. Langou, Department of Mathematical and Statistical Sciences, University of Denver, Colorado, 1250 14th Street-Rm 646, Denver, CO 80217-3364, USA; email: julien.langou@ucdenver.edu.

DOI:http://dx.doi.org/10.1145/2427023.2427026

ACM Reference Format:
Gustavson, F. G., Waśniewski, J., Dongarra, J. J., Herrero, J. R., and Langou, J. 2013. Level-3 Cholesky factorization routines improve performance of many Cholesky algorithms. *ACM Trans. Math. Softw.* 39, 2, Article 9 (February 2013), 10 pages.
DOI:http://dx.doi.org/10.1145/2427023.2427026

1. INTRODUCTION

Cholesky block factorizations of symmetric positive definite matrices started to appear when cache blocking was first introduced [Gallivan et al. 1987; IBM 1986]. We consider A where A is stored in **B**lock **P**acked **F**ormat (BPF) [Gustavson 2001, 2003]. In Andersen et al. [2005] and Gustavson et al. [2007b, Algorithm 865] a variant of BPF called BPHF, where H stands for Hybrid, was presented. BPF has two variants called lower and upper BPF. Here we mostly study upper BPF, which is a block factoring of A into U^TU, where U is an upper triangular matrix. Upper BPF is also *S*quare *B*lock *P*acked *F*ormat (SBPF) [Gustavson 2001] for packed format and SBPF is the format used by multicore implementations. In Section 2 algorithm _BPTRF, which uses BPF, is given. _BTRF is a restructured form of the LAPACK factorization routines _PPTRF or _POTRF. _BPTRF uses about the same storage as _PPTRF does. However, _BPTRF performance is better than or equal to _POTRF performance as BPF can also take advantage of Level-3 BLAS operations [Dongarra et al. 1990; IBM 1986]. Finally, _BPTRF using BPF is very competitive with multicore implementations of Cholesky factorization, whereas traditional _POTRF implementation are `not`; see Kurzak et al. [2008] for _POTRF and Agullo et al. [2010; Bouwmeester and Langou 2010] for _POTRI. Section 3 details another main difference between the _BPTRF and _POTRF algorithms. _BPTRF uses routines _POTF3i[1]. _POTF3i are Level-3 Fortran routines that use register blockings [Gustavson 2004; Gustavson et al. 2007a]. The four routines _POTRFi use different register blocking sizes. LAPACK _POTRF uses _POTF2, which is based on Level-2 BLAS operations.

Section 4 gives performance results showing the Level-3 Fortran routines _POTF3i can increase the block size nb used by a traditional LAPACK routine such as _POTRF where performance usually starts to degrade at $nb = 64$ for _POTF2. However, performance increases past block size 64 to 120 or more for our Level-3 Fortran routines _POTF3i. These performance gains come from the use of *S*quare *B*lock (SB) format, the use of Level-3 register blocking and the elimination of all subroutine calls within _POTF3i. Section 3.1 gives further reasons why _POTF3i can use a larger nb. The increase in nb improves the overall performance of _BPTRF: the main computational parts of _BPTRF consist of calls to Level-3 BLAS _TRSM, _SYRK and _GEMM. For example, all calls to level-3 BLAS _GEMM performs better when its k dimension is larger and for _BPTRF $k = nb$. It therefore follows that, for all n, overall performance of `_POTRF` and `_BPTRF` increases: `_GEMM` performance is the key performance component of `_POTRF` and `_BPTRF`. In Gustavson et al. [2011b], an enlarged version of this article, performance results for large n verifying these remarks are given; see also Andersen et al. [2005] and Whaley [2008] where additional performance evidence of these assertions are given.

Lower BPF is not new. It was used by D'Azevedo and Dongarra [1998] as the basis for a Cholesky packed distributed storage version of ScaLAPACK. This storage layout consists of a collection of block columns where each block column has size nb. Lower BPF is *not* a preferred format over upper BPF, as it does not give rise to contiguous SB. Therefore, Section 2.1 indicates how to very efficiently transform each lower block column in place to obtain upper BPF.

---

[1] i stands for one of the four letters `a,b,c,d` as we consider four DPOTF3 routines.

1a. Lower Blocked Packed Format

0							
1	9						
2	10	16					
3	11	17	23				
4	12	18	24	28			
5	13	19	25	29	33		
6	14	20	26	30	34	36	
7	15	21	27	31	35	37	39

1b. Upper Blocked Packed Format

0	2	4	6	8	10	12	14
	3	5	7	9	11	13	15
		16	18	20	22	24	26
			19	21	23	25	27
				28	30	32	34
					31	33	35
						36	38
							39

Fig. 1. Lower Blocked Column Packed and Upper Square Blocked Packed Formats.

Matrix data structures that use matrix tiling of contiguous blocks date back to 1997. We do not have space to fully reference this large area of research; readers are referred to a survey paper that partially covers this field up to 2004 [Elmroth et al. 2004], and to five more recent papers [Agullo et al. 2010; Bouwmeester and Langou 2010; Herrero 2007; Herrero and Navarro 2006; Kurzak et al. 2008].

2. INTRODUCTION TO BPF

Packed storage of a matrix is used to conserve storage when that matrix has special properties. Two examples are symmetric and triangular matrices. By using BPF we may partition a symmetric matrix where each submatrix block is held contiguously in memory [D'Azevedo and Dongarra 1998; Gustavson 2001]. This gives another way to pack a symmetric matrix and it avoids the data copies (see [Gustavson et al. 2007a]), that are inevitable when Level-3 BLAS are applied to matrices held in standard *C*olumn *M*ajor (CM) or *R*ow *M*ajor (RM) format as well as in standard packed format.

We define *lower* and *upper* BPF via an example in Figure 1 with varying length rectangles of width $nb = 2$ and SB of order $nb = 2$ superimposed. Figure 1 gives the memory addresses of the array that holds the matrix elements of BPF. The rectangles making up the array of Figure 1 are in standard Fortran format and hence BPF supports calls to level-3 BLAS. The rectangles in Figure 1(a) are *not* further divided into SB as these SB are *not* contiguous. Figure 1 is a collection of $N = \lceil n/nb \rceil$ rectangular matrices concatenated together. Rectangle i has size $n - i \cdot nb$ by nb for $i = 0, \ldots, N-1$. The ith rectangle has its leading dimension, called `LDA`, equal to $i \cdot nb$ or nb. In Figures 1(a), 1(b) the `LDA`'s are $n - i \cdot nb$ and nb. The rectangles in Figure 1(b) are the transposes of the rectangles in Figure 1(a) and vice versa. Figure 1(b) rectangles have a *major* advantage over the rectangles of Figure 1(a): the ith rectangle consists of $N - i$ order nb SB. This gives two dimensional contiguous granularity for _GEMM calls using upper BPF, which lower BPF *cannot* possess. Using full format requires that `LDA` $\geq n$. Clearly, this wastes about half the storage allocated by Fortran or C to A. On the other hand, for each SB, `LDA` $= nb$. This means *minimal* storage is wasted for large n! nb should be chosen so that a block fits comfortably into a Level-1 or Level-2 cache. The LAPACK ILAENV routine may be called to set nb.

We want to Cholesky factor a matrix A laid out in BPF. We use LAPACK's _POTRF routines modified to use the BPF of Figures 1(a) and 1(b). The code modifications are shown in Figure 2: one needs to call _SYRK and _GEMM $i - 1$ times at factor stage i. Here is the reason: the layout of the block rectangles do *not* have uniform strides across the block rectangles. Another advantage of using upper BPF is one may at factor stage i call _GEMM $(N - i - 1)(i - 1)$ times where each call is a parallel SB _GEMM update. This approach was used by a LAPACK multicore Cholesky implementation [Kurzak

```
do i = 1, N                            ! N = ⌈n/nb⌉
     symmetric rank K update A_ii      ! Call of Level-3 BLAS _SYRK i − 1 times
     Cholesky Factor A_ii              ! Call of LAPACK subroutine _POTF2
     Schur Complement update A_ij      ! Call of Level-3 BLAS _GEMM i − 1 times
     Scale A_ij                        ! Call of Level-3 BLAS _TRSM
end do
```

Fig. 2. LAPACK _POTRF algorithms for BPF of Figure 1. The BLAS calls take the forms `_SYRK(uplo,trans,...)`, `_POTF2(uplo,...)`, `_GEMM(transa,transb,...)`, and `_TRSM(side,uplo,trans,...)`.

et al. 2008]. This implies that a BPF layout supports both traditional and multicore LAPACK implementations.

2.1. In-Place Transformation of Lower BPF to Upper BPF

We want to transpose a rectangle of size LDA $= j \cdot nb$ by nb where $j > 1$. Let this rectangle $j = N-i$ be rectangle i of lower BPF and suppose it holds matrix B. B is in CM format and it consists of nb contiguous columns. Now think of B as being a $N-i$ by nb matrix whose elements are column vectors of length nb and inplace "vector transpose" B to become B^T. B^T consists of $(N-i) \cdot nb$ vectors concatenated together. Also, B^T can be viewed as consisting of $N-i$ order nb SB matrices concatenated together; see Figure 1(a) and Figure 1(b) for examples. This transformation process, for any B, is very efficient as data can be moved in contiguous memory chunks, called lines, of size nb. Since there are N B matrices this efficient operation is also embarrassing parallel! One can do $\lceil N/2 \rceil$ parallel operations for each of the N different rectangles that make up the lower BPF. After completion of these $\lceil N/2 \rceil$ parallel steps one has transformed lower BPF as N variable rectangles inplace to be upper BPF as $N(N+1)/2$ SB matrices. Of course, upper BPF and upper packed SB format are identical representations of the same matrix. Space constraints do not allow us to discuss any details; see Gustavson and Swirszcz [2007] for inplace transposition and Gustavson [2008], Karlsson [2009], and Gustavson et al. [2011a] for inplace "vector transposition".

3. THE _POTF3i ROUTINES

_POTF3i routines are replacement routines for _POTF2. However, they are very different from _POTF2. _POTF3i work very well on BPF and not so well on full format. We only consider upper BPF here. They use tiny block sizes kb. We mostly choose $kb = 2$. These blocks are called *register* blocks. A 2×2 block holds four elements of A; we load them into four scalar variables `t11, t12, t21` and `t22` to alert most compilers to put and hold these scalars in registers. For a diagonal block $a_{i:i+1,i:i+1}$ we load the upper triangle into `t11, t12` and `t22`, update it with an inline form of _SYRK, factor it, and store it over $a_{i:i+1,i:i+1}$ as $u_{i:i+1,i:i+1}$. This combined operation is called fusion by the compiler community. Note we are using colon notation [Golub and Van Loan 1996]. For an off diagonal block $a_{i:i+1,j:j+1}$ we load it, update it with an inline form of _GEMM, scale it with an inline form of _TRSM, and store it. This again is an example of fusion. For scaling by $u_{i,i}$ and $u_{i+1,i+1}$ we use reciprocal multiplies. The two reciprocals are saved in two registers during the factor fusion computation. As used here, fusion also avoids procedure call overheads for many very small computations that _POTF3i performs; in effect, we replace all calls to Level-3 BLAS by in-line code. Gustavson [1997], Gustavson and Jonsson [2000], and Yotov et al. [2007] for related remarks on this point.

The key loop in the inline form of our _GEMM and _TRSM fusion computation is the inline form of the _GEMM loop. For this loop, the code of Figure 3(a) is what we used in one of the _POTF3i versions, called DPOTF3a. In Figure 3(a) we show the inline

_GEMM LOOP Code	Routine Name	Number of Registers Used	Register Block Sizes C size — A size — B size
DO k = 1, ii - 1 aki = a(k,ii)			
akj = a(k,jj) t11 = t11 - aki*akj	DPOTF3a	7	2 by 2 — 1 by 2 — 1 by 2
aki1 = a(k,ii+1) t21 = t21 - aki1*akj	DPOTF3b	8	1 by 4 — 1 by 1 — 1 by 4
akj1 = a(k,jj+1) t12 = t12 - aki*akj1	DPOTF3c	14	2 by 4 — 1 by 2 — 1 by 4
t22 = t22 - aki1*akj1 END	DPOTF3d	6	2 by 2 — 1 by 2 — 1 by 2

Fig. 3. (a) _GEMM loop code for $C = C - A^T B$. & (b) Table for DPOTF3i.

form of the _GEMM loop. The underlying array is $A_{i,j}$ and the 2 by 2 register block starts at location `(ii,jj)` of array $A_{i,j}$; see Figure 3(b) where information is given for the three register blocks of _GEMM operands A, B, C. DPOTF3a uses 8 local variables that compilers will place in registers. The loop body does 4 memory accesses and 8 floating-point operations. In DPOTF3b, we accumulate into a vector block of size 1×4. Each iteration of the vector loop involves 8 floating-point operations as for the 2×2 case; however, 5 real numbers are loaded from cache instead of 4.

We usually got faster execution by having an inner inline form of the _GEMM loop that updated both 2 by 2 blocks $A_{i,j}$ and $A_{i,j+1}$. This version of _POTF3i is called DPOTF3c. For it the scalar variables `aki` and `aki1` need only be loaded once, so we now have 6 memory accesses and 16 floating-point operations. If possible, all 14 local variables of this loop should be assigned to registers. Code for _POTF3c is available in the TOMS paper [Gustavson et al. 2007b, Algorithm 865]. Routine DPOTF3d is similar to DPOTF3a. However, DPOTF3d does *not* use the FMA instruction. Instead, it uses multiplies followed by adds. We close this section by making a very important remark: Level-1 BLAS _AXPY is slower than Level-1 BLAS _DOT. The *opposite* statement is true when the matrix data resides in floating point registers.

3.1. _POTF3i Routines Can Use a Larger Block Size *nb*

The element domain of A for Cholesky factorization using _POTF3i is an upper triangle of a SB. Furthermore, in the outer loop of _POTF3i at stage j, where $0 \le j < nb$, only address locations $L(j) = j(nb - j)$ of the upper triangle of Figure 1(b)[2] are accessed. The maximum value of $nb^2/4$ of address function L occurs at $j = nb/2$. Hence, during execution of _POTF3i, only half of the cache block of size nb^2 is used and the maximum usage of cache at any time instance is just one quarter of the size of a SB. Thus, _POTF3i can use a larger block size before its performance will start to degrade. This fact is true for all four _POTF3i computations.

4. PERFORMANCE

In Gustavson et al. [2011b] we presented several experiments that corroborate our conjectures. In this article, however, we will only provide details on Experiment I.

Our calculations are done in DOUBLE PRECISION. Thus, the names of the subroutines are DPOTRF and DPOTF2 from the LAPACK library and four simple Fortran Level-3 DPOTF3i routines described in the following and also in Section 3. These four routines are subroutines used entirely by DBPTRF for matrix orders below size about

[2] $nb = 2$ in Figure 1(b). In real applications $nb \approx 100$ and so the triangle holds 5050 elements out of 10000 when $nb = 100$. Also, $nb^2/4 = 2500$.

120. LAPACK DPOTRF calls LAPACK DPOTF2, which calls Level-2 BLAS routine DGEMV. DPOTRF and DBPTRF both call Level-3 BLAS routines DTRSM, DSYRK, and DGEMM. DPOTRF also calls LAPACK subroutine ILAENV, which sets the block size nb used by DPOTRF. The four Fortran routines DPOTF3i are a new type of Level-3 BLAS called FACTOR BLAS.

We only use upper BPF in our performance studies. We do *not* try to take advantage of additional parallelism that is inherent in upper BPF. This allows for a fairer comparison of _POTRF and _BPTRF in an SMP environment that is traditionally Level-3 BLAS based. In fact, this decision is unfair to _BPTRF because _POTRF makes $O(N)$ calls to Level-3 BLAS whereas _BPTRF makes $O(N^2)$ to Level-3 BLAS; see Table 1 of Section 3.1 in Gustavson et al. [2011b] where the calling overhead of _POTRF and _BPTRF is given a detailed treatment. The reason we say unfair has to do with Level-3 BLAS having more surface area per call in which to optimize. The greater surface area comes about because _POTRF makes $O(N)$ calls whereas _BPTRF has to make $O(N^2)$ calls. In addition, a highly optimized BLAS library may have BLAS-2 routines, such as GEMV, that use thread-level parallelism that will speed up _POTF2.

4.1. Performance Preliminaries for Experiment I

We consider matrix orders of 40, 64, 72, 100 since these orders will typically allow the computation to fit comfortably in Level-1 or Level-2 caches.

Comparison numbers in Table I are given in Mflop/s. Results are given for six platforms: SUN UltraSPARC IV+, SGI - Intel Itanium2, IBM Power6, Intel Xeon, AMD Dual Core Opteron, and Intel Xeon Quad Core. Table I has 13 columns. The matrix order is in column one. Results of the vendor optimized Cholesky routine DPOTRF and the Recursive Algorithm [Andersen et al. 2001] are given in columns two and three. Column 4 contains results when DPOTF2 is used within DPOTRF with block size $nb = 64$. On most of our computers this block size was best. Column 5 contains results when DPOTF2 is called by itself. In columns 7, 9, 11, 13 the four DPOTF3i routines are called by themselves. In columns 6, 8, 10, 12 the four DPOTF3i, `i=a,b,c,d`, routines are called by DPOTRF with block size $nb = 64$.

The resolution of our timer used in Table I was too coarse. Thus, for small matrices our time is the average of several executions run in a loop. On some platforms we had to run in batch mode; eg, IBM Huge. Thus, there were some anomalous timings; for instance, for $n = 40$ column 5 time should be less than column 4 time.

4.2. Interpretation of Performance Results for Experiment I

We use five Fortran routines in this study besides DPOTRF; see Section 3 and Figure 3(b) for details. They are the following.

(1) LAPACK routine DPOTF2. Columns 4 and 5 show results of calling DPOTRF and of only calling routine DPOTF2.
(2) The 2×2 blocking routine DPOTF3a is specialized for the operation FMA $(a\times b + c)$ using seven floating point registers (FPRs). DPOTRF calls DPOTF3a in column 6 and DPOTF3a is called alone in column 7.
(3) The 1×4 blocking routine DPOTF3b is optimized for the case $\mathrm{mod}(n, 4) = 0$ where n is the matrix order. It uses eight FPRs. DPOTRF calls DPOTF3b in column 8 and DPOTF3b is called alone in column 9.
(4) The 2×4 blocking routine DPOTF3c uses fourteen FPRs. DPOTRF calls DPOTF3c in column 10 and DPOTF3c is called alone in column 11.
(5) The 2×2 blocking routine DPOTF3d. It is not specialized for the FMA operation and uses six FPRs. DPOTRF calls DPOTF3d in column 12 and DPOTF3d is called alone in column 13.

Table I. Performance in Mflop/s of the Kernel Cholesky Algorithm. Comparison between Different Computers and Different Versions of Subroutines

Mat ord	Ven dor lap	Recur sive lap	dpotf2		2x2 w. fma 8 flops		1x4 8 flops		2x4 16 flops		2x2 8 flops	
			lap	fac	lap	fac	lap	fac	lap	fac	lap	fac
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, Dual Core, 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
Zoot: 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

It is important to note that Level-3 BLAS are called only in columns 4, 6, 8, 10, 12 for block sizes 72 and 100, as ILAENV has set the block size to be 64 in our study. In odd columns 5 to 13 DPOTF2 and DPOTF3i are called.

In column 11 the DPOTF3c code is very successful on the Sun (Newton), SGI (Freke), IBM (Huge) and Quad Core Xeon (Zoot) computers. For these four platforms, it greatly outperforms the compiled LAPACK code and the recursive algorithm. Except on the IBM (Huge) platform for $n \geq 40$ it outperforms all the other vendor optimized codes. The DPOTF3d code in column 13 is best on the Intel Xeon (Battle) computer. The DPOTF3b code in column 9 is superior on the Dual Core AMD (Nala) platform. All the best results are colored in red.

Table I reveals an innovation about using Level-3 Fortran DPOTF3(a,b,c,d) codes over use of Level-2 LAPACK DPOTF2 code, which we now explain. The results of columns 10 and 11 are about equal $n = 40$ and $n = 64$. Column 10 does extra work in which DPOTRF calls ILAENV, which sets $nb = 64$. It then calls DPOTF3c and returns after DPOTF3c completes. In column 11 only DPOTF3c is called. Thus column 10 time is slightly more than column 11 time. Now take $n = 72$ and $n = 100$. In DPOTRF, ILAENV sets $nb = 64$, and then does a Level-3 blocked computation. Let $n = 100$. With nb set to 64 DPOTRF does a sub blocking of block sizes equal to 64 and 36 and DPOTRF calls Factor(64), DTRSM(64,36), DSYRK(36,64), and Factor(36) before returning. The two Factor calls are to the DPOTF3c routine. However, in column 11, DPOTF3c is called only once with $n = 100$. In column 11 performance is always increasing over doing the Level-3 blocked computation of DPOTRF. This means that DPOTF3c is outperforming DTRSM and DSYRK as n increases from 64 to 100. Now, look at columns 4 and 5. For $n = 40$ and $n = 64$ the results are again about equal. For $n = 72$ and $n = 100$ the results favor DPOTRF with Level-3 blocking except for the Zoot platform and the Battle platform for $n = 72$. Zoot and Battle are 4 way and 2 way Intel platforms. We suspect DGEMV has been made parallel; see the last paragraph of Section 4. Thus, one sees DPOTF2 performance is decreasing relative to a blocked computation as n increases from 64 to 100. An increasing result is true for most of the columns six to thirteen; namely DPOTF3(a,b,c,d) performance is increasing relative to the blocked computation as n increases from 64 to 100. The exception is the IBM Huge platform for columns (6,7), (8,9), (12,13). This platform has 32 FPRs. Column (10,11) is using 14 FPRs and DPOTF3c exhibits the increasing result. In the three exceptional columns DPOTF3(a,b,d) uses 7, 8 and 6 FPRs.

We have just seen that routines DPOTF3i outperform DPOTF2 for $n \approx nb$. Also, both DBPTRF and DPOTRF perform better for large n when DPOTF3i routines are substituted for DPOTF2. We explain. Take any n for DPOTRF. DPOTRF will do a blocked computation with this larger block size for $n \geq nb$. All three BLAS subroutines, DGEMM, DSYRK and DTRSM, of DPOTRF will now perform better when called by DPOTRF with this larger block size!

Andersen et al. [2005] give large n performance results for BPHF where nb was set larger than 64. The results for $nb = 100$ were much better. The explanations in Sections 3 and 4 explain why. They also confirm the results of Whaley [2008]. Finally, see Section 1.1.1 and the remaining Sections of 3 in Gustavson et al. [2011b] where we give further confirming experimental results for large n.

These results emphasize that LAPACK users should use ILAENV to set nb based on the speeds of Factorization, DTRSM, DSYRK and DGEMM. This information is part of the LAPACK User's Guide. The results of [Whaley 2008] provide a means of setting a variable nb for DPOTRF where nb increases as n increases.

The code for the 1×4 DPOTF3b subroutine is available from the companion paper [Gustavson et al. 2007b, Algorithm 865]. The code for _POTRF and its subroutines is available from the LAPACK package [Anderson et al. 1999].

5. SUMMARY AND CONCLUSIONS

We demonstrated that four simple Fortran codes DPOTF3i produce Level-3 Cholesky factorization routines that perform better than the Level-2 LAPACK DPOTF2 routine. DPOTF3i allowed DPOTRF to increase its block size nb. Since nb is the k dimension of the Level-3 BLAS _GEMM, _SYRK and _TRSM routines their SMP performance increases. Hence the performance of SMP _POTRF increases. In Gustavson et al. [2011b] we provided "three performance conjectures" with explanations on why they were "true". Also, three performance studies were conducted that "verified" these conjectures. These three performance results were corroborated by the results of Andersen et al. [2005] and Whaley [2008]. Also, in Gustavson et al. [2011b], DBPTRF performance was usually optimal for one nb for an entire range of n values. For DPOTRF, using DPOTF2, one needs to increase nb as n increased to obtain optimal performance. Because of space limitations this article included only performance results of experiment I from Gustavson et al. [2011b].

We described BPF format, which has two cases lower and upper BPF. Lower BPF format consists of $N = \lceil n/nb \rceil$ rectangular blocks whose LDA's are $n - i \cdot nb$ for $0 \le i < N$. Upper BPF had the additional property that each of its rectangular blocks were also a multiple number of square blocks so there are $N(N+1)/2$ SB in all. We presented algorithm DBPTRF and showed that its code were trivial modifications of the LAPACK _POTRF and _PPTRF algorithms. Upper BPF is multicore data layout. The current Cell implementations of Kurzak et al. [2008], for full format, should carry over to _BPTRF with trivial modifications. Agullo et al. [2010] and Bouwmeester and Langou [2010] indicate this is true.

We described in Section 2.1 how a vertical rectangular block could be very efficiently transformed inplace to be a multiple of square blocks by a parallel vector inplace transpose algorithm. A purpose of our article is to promote the new *Block Packed Data Format* storage or its variants. Traditional LAPACK full format algorithms and their related Level-3 BLAS are no longer being used on multicore processors. For full format symmetric and triangular matrices the format used by multicore is SBPF; for packed format SBPF is equal to upper BPF.

ACKNOWLEDGMENTS

We thank the referee of our article for suggesting an overall outine that we followed. Thanks for consulting help go to Bernd Dammann on the SUN system; to Niels Carl W. Hansen on the IBM and SGI systems; and Paul Peltz and Barry Britt on the Core2 and Intel-Xeon systems.

REFERENCES

Agullo, E., Bouwmeester, H., Dongarra, J., Kurzak, J., Langou, J., and Rosenberg, L. 2010. Towards an efficient tile matrix inversion of symmetric positive definite matrices on multicore architectures. arXiv: 1002.4057v1, University of Tenn at Knoxville and University of Colorado at Denver.

Andersen, B. S., Gustavson, F. G., and Waśniewski, J. 2001. A recursive formulation of Cholesky facorization of a matrix in packed storage. *ACM Trans. Math. Softw 27*, 2, 214–244.

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 Trans. Math. Softw. 31*, 201–227.

Anderson, E., Bai, Z., et al. 1999. *LAPACK Users' Guide* 3rd Ed. SIAM, Philadelphia, PA.

Bouwmeester, H. and Langou, J. 2010. A critical path approach to analyzing parallelism of algorithmic variants. Application to Cholesky inversion. arXiv: 1010.2000v1, University of Colorado at Denver.

D'Azevedo, E. and Dongarra, J. J. 1998. Packed storage extension of ScaLAPACK. ORNL rep. 6190, Oak Ridge National Laboratory.

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. Softw. 16*, 1, 18–28.

Elmroth, E., Gustavson, F. G., Jonsson, I., and Kågström, B. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev. 46*, 1, 3–45.

Gallivan, K., Jalby, W., and Meier, U. 1987. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Stat. Comp. 8*, 1079–1084.

Golub, G. and Van Loan, C. F. 1996. *Matrix Computations* 3rd Ed. Johns Hopkins University Press, Baltimore, MD.

Gustavson, F. G. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev. 41*, 6, 737–755.

Gustavson, F. G. 2001. New generalized data structures for matrices lead to a variety of high-performance algorithms. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics* (Revised Papers).

Gustavson, F. G. 2003. High performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev. 47*, 1, 823–849.

Gustavson, F. G. 2004. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In *Proceedings of the International Workshop on Applied Parallel*

Computing. J. W. J. J. Dongarra, K. Madsen Eds., Lecture Notes in Computer Science, vol. 3732, Springer, 11–20.

Gustavson, F. G. 2008. The relevance of new data structure approaches for dense linear algebra in the new multicore/manycore environments. IBM RC rep. 24599, IBM Research, Yorktown.

Gustavson, F. G. and Jonsson, I. 2000. Minimal storage high performance cholesky via blocking and recursion. *IBM J. Res. Dev. 44*, 6, 823–849.

Gustavson, F. G. and Swirszcz, T. 2007. In-place transposition of rectangular matrices. In *Proceedings of the International Workshop on Applied Parallel Computing*. Lecture Notes in Computer Science, vol. 4699, Springer, 560–569.

Gustavson, F. G., Gunnels, J., and Sexton, J. 2007a. Minimal data copy for dense linear algebra factorization. In *Proceedings of the International Workshop on Applied Parallel Computing*. Lecture Notes in Computer Science, vol. 4699, Springer, 540–549.

Gustavson, F. G., Reid, J. K., and Waśniewski, J. 2007b. Algorithm 865: Fortran 95 subroutines for Cholesky factorization in blocked hybrid format. *ACM Trans. Math. Softw. 33*, 1, 5.

Gustavson, F. G., Karlsson, L., and Kågström, B. 2011a. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Trans. Math. Softw 37*, xx–xx+33.

Gustavson, F. G., Waśniewski, J., Dongarra, J. J., Herrero, J. R., and Langou, J. 2011b. Level-3 Cholesky factorization routines as part of many Cholesky algorithms. Tech. rep. 249, LAPACK Working Note.

Herrero, J. R. 2007. New data structures for matrices and specialized inner kernels: Low overhead for high performance. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics (PPAM'07)*. Lecture Notes in Computer Science, vol. 4967, Springer, 659–667.

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*. Lecture Notes in Computer Science, vol. 3984, 762–771.

IBM. 1986. *Engineering and Scientific Subroutine Library, Guide and Reference* 1st Ed. (ProgramNumber 5668-863).

Karlsson, L. 2009. Blocked in-place transposition with application to storage format conversion. Report# uminf - 09.01. Department of Computer Science, Umeå University, Umeå, Sweden.

Kurzak, J., Buttari, A., and Dongarra, J. 2008. Solving systems of linear equations on the cell processor using Cholesky factorization. *IEEE Trans. Parallel Distrib. Syst. 19*, 9, 1175–1186.

Whaley, C. 2008. Empirically tuning LAPACK's blocking factor for increased performance. In *Proceedings of the Conference on Computer Aspects of Numerical Algorithms (CANA'08)*.

Yotov, K., Roeder, T., Pingali, K., Gunnels, J. A., and Gustavson, F. G. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. 93–104.

Received January 2009; revised January 2010, June 2011; accepted February 2012