Level-3 Cholesky Factorization Routines as Part of Many Cholesky Algorithms

Fred G. Gustavson
IBM T.J. Watson Research Center, Emeritus and Umeå University, Adjunct
and
Jerzy Waśniewski
Department of Informatics and Mathematical Modelling, Tech. University of Denmark
and
Jack J. Dongarra
University of Tennessee, Oak Ridge National Laboratory and University of Manchester
and
José R. Herrero
Computer Architecture Dept., Universitat Politècnica de Catalunya, BarcelonaTech
and
Julien Langou
University of Colorado at Denver

June 15, 2011

DTU/IMM-Technical Report-2011-11

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 E-mail: 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

Some Linear Algebra Libraries use Level-2 routines during the factorization part of any Level-3 block factorization algorithm. We discuss four Level-3 routines called DPOTF3i, `i = a,b,c,d`, a new type of BLAS, for the factorization part of a block Cholesky factorization algorithm for use by LAPACK routine DPOTRF or for BPF (Blocked Packed Format) Cholesky factorization. The four routines DPOTF3i are Fortran routines. Our main result is that performance of routines DPOTF3i is still increasing when the performance of Level-2 routine DPOTF2 of LAPACK starts to decrease. This means that the performance of DGEMM, DSYRK, and DTRSM will increase due to their use of larger block sizes and also by making less passes over the matrix elements. We present corroborating performance results for DPOTF3i versus DPOTF2 on a variety of common platforms. The four DPOTF3i routines use simple register blocking; different platforms have different numbers of registers and so our four routines have different register blocking sizes.

Blocked **P**acked **F**ormat (BPF) is introduced and discussed. 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 forms of BPF: we call them lower and upper BPF. Upper BPF is shown to be identical to **S**quare **B**lock **P**acked **F**ormat (SBPF). SBPF is used in "LAPACK" implementations on multi-core processors. Performance results for DBPTRF using upper BPF and DPOTRF for large n show that routines DPOTF3i do increase performance for large n. Lower BPF is almost always less efficient than upper BPF. A form of inplace transposition called vector inplace transposition can very efficiently convert lower BPF to upper BPF.

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, Cache Blocking, BLAS, 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.

1. INTRODUCTION

We consider Cholesky block factorizations of a symmetric positive definite matrix A where A is stored in **B**lock **P**acked **F**ormat (BPF) [Gustavson 2000; Gustavson 2003]. In [Andersen et al. 2005], [Gustavson et al. 2007, Algorithm 685] a variant of BPF called BPHF,where H stands for Hybrid, is presented. We will mostly study a block factoring of A into $U^T U$, where U is an upper triangular matrix. Upper BPF is also **S**quare **B**lock **P**acked **F**ormat (SBPF) [Gustavson 2000]; see Section 1.2 for details. We also show in Section 1.2 that the implementation of _BPTRF using BPF is a restructured form of the LAPACK factorization routines _PPTRF or _POTRF. _BPTRF uses slightly more storage than _PPTRF. _BPTRF uses about half the storage of _POTRF; however, _BPTRF performance is better than or equal to _POTRF performance. Matrix-matrix operations that take advantage of Level-3 BLAS are used by _BPTRF and thereby its higher performance [Dongarra et al. 1990; IBM 1986] is achieved. This paper focuses on the replacement of routines LAPACK _PPTF2 or _POTF2, which are based on Level-2 BLAS operations, by routines _POTF3i [1]. _POTF3i are Level-3 Fortran routines that use register block-

[1] i stands for `a,b,c,d`. We consider four DPOTF3 routines. Henceforth, the suffix i will mean `i = a,b,c,d`.

ings; see [Gustavson et al. 2007]. The four routines _POTRFi use different register blocking sizes.

The performance numbers presented in Section 3 show that the Level-3 factorization Fortran routines _POTF3i give improved performance over the traditional Level-2 factorization _POTF2 routine used by LAPACK. The 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. It is necessary that _POTF3i not call the BLAS for register block sizes kb that are tiny. The calling overheads have a disastrous effect. We give performance results to demonstrate what will happen; see also, [Gustavson and Jonsson 2000; Gunnels et al. 2007].

The performance gains are two fold. First, for $n \approx nb$, _POTF3i outperforms both _POTRF and _POTF2. For large n, _POTRF routines gain from Level-3 BLAS routines _GEMM, _SYRK and _TRSM that are performing better due to using a larger nb value than the default nb value used by LAPACK. Some performance results for DGEMM, DTRSM and DSYRK are presented to show this fact. Also, performance results for DPOTRF and DBPTRF are reported for large n. These gains, especially for DBPTRF, suggests a change of direction for traditional LAPACK packed software. We mention that "LAPACK" implementations for Cholesky inversion use SBPF [Agullo et al. 2010; Bouwmeester and Langou 2010]. Therefore, these implementations can be done using upper BPF.

A main point of our paper is that the Level-3 Fortran routines _POTF3i allows one to increase the block size nb used by a traditional LAPACK routine such as _POTRF. Our experimental results show that performance usually starts degrading around block size 64 for _POTF2. However, performance continues to increase past block size 64 to 120 or more for our new Level-3 Fortran routines _POTF3i. Such an increase in nb will improve the overall performance of _BPTRF as the Level-3 BLAS _TRSM, _SYRK and _GEMM will perform better for two reasons. The first reason is that Level-3 BLAS perform better when the $K = nb$ dimension of _GEMM is larger. The second reason is that Level-3 BLAS are called less frequently by a ratio of increased block size of the Level-3 Fortran routines _POTF3i over the block size used by Level-2 routine _POTF2. Experimental verifications of these assertions are given by our performance results and also by the performance results in [Andersen et al. 2005]. The recent paper by [Whaley 2008] also demonstrates that our assertions are correct; he gives both experimental and qualitative results.

One variant of our BPF, lower BPF, is not new. It was used by [D'Azevedo and Dongarra 1998] as the basis for packed distributed storage used by a variant of ScaLAPACK. This storage layout consist of a collection of block columns; each of these has column size nb. Each block column is stored in standard **C**olumn **M**ajor (CM) format. In this variant one does a LL^T Cholesky factorization, where L is a lower triangular block matrix. Lower BPF is not a preferred format as it does not give rise to contiguous SB. It probably should never be used. All of our performance results only use upper BPF. Therefore, another point of our paper is that we can transpose each block column inplace to obtain upper BPF which is then also a SB format data layout. Both layouts use about the same storage as LAPACK _PPTRF routines. More importantly, BPF can use Level-3 BLAS routines so their performance is about the same as LAPACK _POTRF routines and hence they have much better performance than the LAPACK _PPTRF routines.

The field of data structures using matrix tiling with contiguous blocks dates back at least to 1997. Space does not allow a detailed listing of this large area of research. We refer readers to a survey paper which partially covers the field up to 2004 [Elmroth et al. 2004], and to five more recent papers [Herrero and Navarro 2006; Herrero 2007; Kurzak et al. 2008; Agullo et al. 2010; Bouwmeester and Langou 2010].

1.1 Use of _GEMM, _TRSM, _SYRK, _GEMV and _POTF3i in this paper

We use standard vendor or ATLAS BLAS in this paper. _POTF2 uses _GEMV to get its performance. **A P**rogramming **I**nterface, (API), for these BLAS is full format. Use of BLAS can be considered as using a "black box", since the BLAS we use do not know that we are using BPF instead of full format. Hence, our use of these BLAS may be suboptimal [Gustavson 2000; Gustavson 2003; Gustavson et al. 2007; Herrero 2007]; eg, these BLAS may re-format BPF when this re-formatting is unnecessary. It is beyond the scope of this paper to deal with this issue. The four _POTF3i routines are Fortran routines! So, strictly speaking they are not highly tuned as the BLAS are. However, they give surprisingly good performance on several current platforms. Like all traditional BLAS, their API is full format which is the standard two dimensional array format that both Fortran and C support. One could change the API for _POTF3i to be "register block" format and achieve even better performance. However, for portability reasons this has not been done.

All of our performance studies except one concern a single processor so parallelism is not an issue except for that processor. However, in Section 3.7 we consider an Intel/Nehalem X5550, 2.67 GHz, 2x Quad Core processor using a Portland compiler and vendor BLAS for Double Precision computations using LAPACK DPOTRF with DPOTF2 and DPOTF3i and DBPTRF using BPF with DPOTF3i. We note or remind the reader that the vendor BLAS have been optimized for this platform but that routines DBPTRF and DPOTF3i are *not* optimized for this platform.

1.1.1 *Use of SBPF and Customized BLAS.* We illustrate what is possible with SBPF when the architecture is known and hence DGEMM, DSYRK, DTRSM and a special DPOTF3 [2] routine are designed using this knowledge. In Fig. 1 we compare a right looking version of DPOTRF with a right looking SBPF implementation of Cholesky factorization [Gustavson 2003].

Fig. 1 gives performance in MFlops versus matrix order n. The x-axis is log scale. We let order n range from 10 to 2000. Like the four DPOTF3i routines, the DPOTF3 routine is written in Fortran.

The square block size has order $nb = 88$. SBPF Cholesky performance exhibits some choppy behavior, especially when $n \approx nb$. The matrix orders where this behavior occurs satisfy $\bmod(n, kb) \neq 0$; eg, when $n = 70$, the performance is about the same as when $n = 60$. This is because SBPF Cholesky routine is solving an order $n = 72$ problem where two rows and columns have zero or one padding; the MFlops calculation is done for $n = 70$. SBPF Cholesky performance in Fig. 1 is always faster than DPOTRF performance by as much as a factor of 4 or 400% when $n = 60$ and at least 15% for $n = 2000$.

[2] DPOTF3 uses an order $kb = 4$ register block size on an IBM POWER3; it makes use of 24 FP out of 32 FP registers.

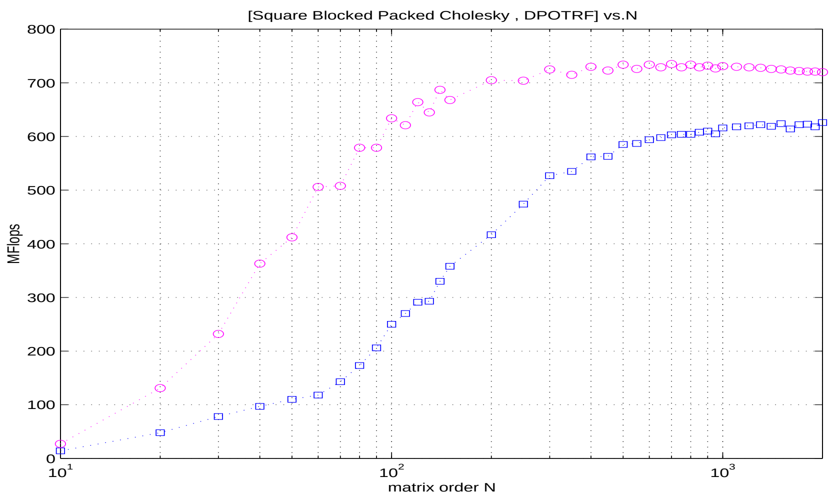

Fig. 1. Performance of Right Looking SBPF (plot symbol ○) and DPOTRF (plot symbol □) Cholesky factorization algorithms on an IBM POWER3 of peak rate 800 MFlops. DPOTRF calls DPOTF2 and ESSL BLAS. SBPF Cholesky calls DPOTF3 and BLAS kernel routines

1.2 Introduction to BPF

The purpose of packed storage for a matrix is to conserve storage when that matrix has a special property. Symmetric and triangular matrices are two examples. In designing the Level-3 BLAS, [Dongarra et al. 1990] did not specify packed storage schemes for symmetric, Hermitian or triangular matrices. The reason given at the time was "such storage schemes do not seem to lend themselves to partitioning into blocks ... Also packed storage is required much less with large memory machines available today". Our BPF algorithms, using BPF, show that "packing and Level-3 BLAS" are compatible resulting in no performance loss. As memories continue to get larger, the problems that are solved get larger: there will always be an advantage in saving storage especially if performance can be maintained.

We pack a symmetric matrix by using BPF where each block is held contiguously in memory [D'Azevedo and Dongarra 1998; Gustavson 2000]. This usually avoids the data copies, see [Gustavson et al. 2007], that are inevitable when Level-3 BLAS are applied to matrices held in standard CM or **R**ow **M**ajor (RM) format in rectangular arrays. Note, too, that many data copies may be needed for the same submatrix in the course of a Cholesky factorization [Gustavson 2000; Gustavson 2003; Gustavson et al. 2007].

We define *lower and upper BPF* via an example in Fig. 2 with varying length rectangles of width $nb = 2$ and SB of order $nb = 2$ superimposed. Fig. 2 shows where each matrix element is stored within the array that holds it. The rectangles of Fig. 2 are suitable for passing to the BLAS since the strides between elements of each rectangle is uniform. In Fig. 2a we do *not* further divide each rectangle into SB's as these SB are *not* contiguous as they are in Fig. 2b. BPF consists of a collection of $N = \lceil n/nb \rceil$ rectangular matrices concatenated together. The size of the i^{th} rectangle is $n - i \cdot nb$ by nb for $i = 0, \ldots, N-1$. Consider the i^{th}

2a. 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

2b. Upper Blocked Packed Format

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

Fig. 2. Lower Blocked Column Packed and Upper Square Blocked Packed Formats

```
do $i$ = 1, $N$                              ! $N = \lceil n/nb \rceil$
    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. 3. LAPACK _POTRF algorithms for BPF of Fig. 2. The BLAS calls take the forms _SYRK(uplo,trans,...), _POTF2(uplo,...), _GEMM(transa,transb,...), and _TRSM(side,uplo,trans,...).

rectangle. Its leading dimension, called LDA, is either $i \cdot nb$ or nb. In Figs. 2a, b the LDA's are $n - i \cdot nb, nb$. The rectangles in Fig. 2a are the transposes of the rectangles in Fig. 2b and vice versa. The rectangles of Fig. 2b have a major advantage over the rectangles of Fig. 2a: the i^{th} rectangle consists of $N - i$ square blocks. This gives two dimensional contiguous granularity to _GEMM for upper BPF which lower BPF *cannot* possess. We therefore need a way to get from a lower layout to an upper layout in-place. If the matrix order is n and the block size is nb, and $n = N \cdot nb$ then this rearrangement may be performed *very efficiently* in-place by a "vector transpose" routine [Gustavson 2008; Karlsson 2009; Gustavson et al. 2011]. Otherwise, this rearrangement, if done directly, becomes very costly. Therefore, this condition becomes a *crucial* condition. So, when the order n is *not* an integer multiple of the block size, we pad the rectangles so the i^{th} LDA becomes $(N - i) \cdot nb$ and hence a multiple of nb. In effect, we waste a little storage in order to gain some performance. We further assume that nb is 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. In Section 1.5 we briefly discuss vector transposition.

We factorize the matrix A as laid out in Figs. 2 using LAPACK's _POTRF routines trivially modified to handle the BPF of Figs. 2; see Fig. 3. This trivial modification is shown in Fig. 3 where one needs to call _SYRK and _GEMM $i - 1$ times at factor stage i because the layout of the block rectangles do not have uniform stride across the block rectangles. For all our performance studies in Section 3 we only use upper BPF. We do *not* try to take advantage of additional parallelism that is inherent in upper BPF. This allows for a fair comparison of _POTRF and _BPTRF in an SMP environment that is traditionally Level-3 BLAS based. In fact, this decision is decidedly 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 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 makes $O(N^2)$ calls.

Now we briefly discuss an additional advantage of only upper BPF: One can call _GEMM $(N-i-1)(i-1)$ times where each call is a SB _GEMM update. This approach was used by a LAPACK multicore implementation [Kurzak et al. 2008]; see also Section 1.4. We close Section 1.2 with an important observation: a BPF layout supports both traditional and multicore LAPACK implementations.

1.3 Four other Definitions of Lower and Upper BPF

One can transpose each of the variable $N = \lceil n/nb \rceil$ rectangular blocks of lower _BPF. What one gets is a set of $N(N+1)/2$ SB each stored rowwise. We call this format *lower SBPF*. Now reflect lower SBPF along its main block diagonal. What one gets is upper _BPF. Thus, lower SBPF and upper _BPF are isomorphic or identical. Now take upper _BPF. Note that its $N(N+1)/2$ SB are stored block rowwise in the order of lower packed format of size N. We now "sort" these $N(N+1)/2$ blocks so that they are stored block columnwise in upper packed format order of size N. Note that this "sort" can be done in-place using the mapping $k \to \bar{k}$ of lower blocked packed storage to upper blocked packed storage: $k = j(2N-j+1)/2 + i - j \to j(j+1)/2 + i = \bar{k}$. Each domain element k of this in-place map corresponds to a SB so that each storage move of a SB at memory locations $k : k + nb^2 - 1$ to a SB at memory locations $\bar{k} : \bar{k} + nb^2 - 1$ corresponds to nb^2 contiguous elements being moved. When done the $N(N+1)/2$ SB will be in regular upper packed format where each scalar a_{ij} is a SB. Here we use a vector transpose algorithm similar to the algorithm briefly described in Section 1.5. The vector length is nb^2. We define this data layout as another form of upper SBPF and call it *upper columnwise SBPF*. This is definition one. Next, transpose each of the $N(N+1)/2$ SB of upper columnwise SBPF. Now we have *upper columnwise SBPF order with each SB stored rowwise*. This is definition two.

Upper columnwise SBPF with each SB stored rowwise has its block column i consisting of i SB concatenated together. Hence this "block column i is single row matrix of size $nb \times i \cdot nb$ with $LDA = nb$. Now transpose each of these N variable rectangular row blocks to get N rectangular blocks stored columnwise, with $LDA = i \cdot nb$, using the vector transpose algorithm described in Section 1.5. Call the resulting format *upper rectangular _BPF*. This is definition three. Now reflect this upper rectangular _BPF along its main block diagonal. What one gets is lower rectangular _BPF with N rectangular blocks stored rowwise. These two formats are isomorphic or identical.

Finally, transpose each of the $N(N+1)/2$ SB of upper _BPF, see Fig. 2b, to get each of its $N(N+1)/2$ SB to be stored rowwise. The storage order of these $N(N+1)/2$ SB is lower packed storage order. Now reflect upper SBPF with all SB stored rowwise along its main block diagonal. One gets isomorphic or identical lower SBPF with all SB stored columnwise. Call the resulting format *lower rowwise SBPF order with each SB stored columnwise*. This is definition four.

1.4 Use of upper _BPF on Multicore Processors

Let A be an order n symmetric matrix. Because of symmetry only about half the elements of A need to be stored. Here _SBPF is upper _BPF; see Fig. 2b. Lower _BPF, see Fig. 2a, can be easily converted in-place to _SBPF; see Section 1.5. Now 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 *no* storage is wasted! In [Agullo et al. 2010; Bouwmeester and Langou 2010] the authors use SBPF. These two papers concern LAPACK implementations of Cholesky inversion _POTRI on multicore processors. _POTRI uses three LAPACK codes: _POTRF, _TRTRI and _LAUUM. All of these four codes are LAPACK codes and hence A requires storage of $LDA \times n$ where $LDA \geq n$. The authors note that they get better parallel performance when they use extra buffer storage for their tiles (SB). However, it is *not* true that they use extra storage over what _POTRI requires: They must use SBPF to obtain high performance. Hence, even with the extra storage allocated for the buffers (to gain better performance) these authors are using less storage than the storage that full format LAPACK _POTRI requires. So, based on a storage comparison alone, they probably should be comparing their performance results to parallel implementations of _PPTRI.

1.5 In-place transformation of lower BPF to upper BPF

We briefly describe how one gets from standard CM format to SB format for a rectangle with LDA a multiple of nb. Denote any rectangle i of lower BPF as a matrix B and note that B is in CM format: B consists of nb contiguous columns; B has its LDA $= (N - i) \cdot nb$. Think of B as a $N - i$ by nb matrix whose elements are column vectors of length nb. Now "vector transpose" this $N - i$ by nb matrix B of vectors of length nb inplace. After "vector transposition" B will be replaced (overwritten) by B^T which is a size nb by $N - i$ matrix of vectors of length nb. It turns out, as a little reflection will indicate, that B^T can also be viewed as consisting of $N - i$ SB matrices of order nb concatenated together; see Fig. 2a and Fig. 2b for examples. This process is very efficient as data is moved in contiguous memory chunks of size nb. For lower BPF 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. It is beyond the scope of this paper to discuss the details of inplace transposition [Gustavson and Swirszcz 2007] and "vector transposition" [Gustavson 2008; Karlsson 2009; Gustavson et al. 2011]. We only mention that inplace transposition of scalars has very poor performance and inplace transformation of contiguous vectors has excellent performance.

2. THE _POTF3I ROUTINES

_POTF3i are modified versions of LAPACK _POTRF and they can be used as subroutines of LAPACK _POTRF. They can be used as a replacement for _POTF2. However, they are very different from _POTF2. _POTF3i work very well on a contiguous SB that fits into L1 or L2 caches. They use tiny block sizes kb. We

```
DO k = 0, nb/kb - 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. 4. Corresponding _GEMM loop code for the _GEMM _TRSM fusion computation.

mostly choose $kb = 2$. Blocks of this size are called *register* blocks. A 2×2 block contains four elements of A; we load them into four scalar variables `t11, t12, t21` and `t22`. This alerts most compilers to put and hold the small register blocks in registers. For a diagonal block $a_{i:i+1,i:i+1}$ we load it into three of four registers `t11, t12` and `t22`, update it with an inline form of _SYRK, factor it, and store it back into $a_{i:i+1,i:i+1}$ as $u_{i:i+1,i:i+1}$. This combined operation is called fusion by the compiler community. 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. In the scaling operation we replace divisions by $u_{i,i}, u_{i+1,i+1}$ by reciprocal multiplies. The two reciprocals are saved in two registers during the inline form of a _SYRK and factor fusion computation. Fusion, as used here, avoids procedure call overheads for very small computations; in effect, we replace all calls to Level-3 BLAS by in-line code. See [Gustavson 1997; Gustavson and Jonsson 2000; Gunnels et al. 2007] for related remarks on this point. Note that _POTRF does *not* use fusion since it explicitly calls Level-3 BLAS. However, these calls are made at the $nb \gg kb$ block size level or larger area level; the calling overheads are therefore negligible.

The key loop in the inline form of the _GEMM and _TRSM fusion computation is the inline form of the _GEMM loop. For this loop, the code of Fig. 4 is what we used in one of the _POTF3i versions, called DPOTF3a.

In Fig. 4 we show the inline form of the _GEMM loop of the inline form of the fused _GEMM and _TRSM computation. The underlying array is $A_{i,j}$ and the 2 by 2 register block starts at location `(ii,jj)` of array $A_{i,j}$. A total of 8 local variables are involved, which most compilers will place in registers. The loop body involves 4 memory accesses and 8 floating-point operations.

In another _POTF3i version, called DPOTF3b, we accumulate into a vector block of size $1{\times}4$ in the inner inline form of the _GEMM loop. Each execution of the vector loop involves the same number of floating-point operations (8) as for the $2{\times}2$ case; it requires 5 real numbers to be loaded from cache instead of 4.

On most of our processors, faster execution was possible by having an inner inline form of the _GEMM loop that updates both $A_{i,j}$ and $A_{i,j+1}$. This version of _POTF3i is called DPOTF3c. The scalar variables `aki` and `aki1` need only be loaded once, so we now have 6 memory accesses and 16 floating-point operations. This loop uses 14 local variables, and all 14 of them should be assigned to registers. We found that DPOTF3c gave very good performance, see Section 3. The implementation of this version of _POTF3i is available in the TOMS Algorithm paper [Gustavson et al. 2007, Algorithm 685].

Routine DPOTF3d is like DPOTF3a. The difference is that 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.

2.1 _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 $0 \leq j < nb$, only address locations $L(j) = j(nb - j)$ of the upper triangle of Fig. 2b [3] 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 nb^2 cache. This means that _POTF3i can use a larger block size before its performance will start to degrade. This fact is true for all four _POTF3i computations. This is what our experiments showed: As nb increased from 64 to 120 or more the performance of _POTF3i increased. On the other hand, _POTF2 performance started degrading relative to _POTRF as nb increased beyond 64. In Section 3.2 we give performance results that experimentally verify these assertions.

Furthermore, and this is one of our main results, as nb increases so does the k dimension of _GEMM increase as $k = nb$ is used for all _GEMM calls in _POTRF and _BPTRF. 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. See the papers of [Andersen et al. 2005; Whaley 2008] where performance evidence of this assertion is given. In Sections 3.4 to 3.7 we give performance results that experimentally verify these assertions for large n. In Section 1.1.1 we gave an extremely good experimental result of both assertions of this Section. That result used a highly tuned version of _POTF3 and Level-3 BLAS kernels for right looking SBPF Cholesky. For DPOTRF using DPOTF2, the same BLAS kernels were used as building blocks for the Level-3 BLAS that DPOTRF was using.

3. PERFORMANCE

We want to experimentally verify three conjectures. In Section 2, we argued, based on theoretical considerations, that these conjectures are true. In [Gustavson 2000; Gustavson 2003; Andersen et al. 2005; Whaley 2008] similar theoretical and experimental results were given and demonstrated. Here are the conjectures:

(1) _GEMM performance on SMP processors increases as nb increases when _GEMM calling variables M, N and K equal nb, n and nb respectively and $n > nb$. The same type of statement is true for _TRSM and _SYRK.
(2) Using the four Fortran _POTF3i routines with _BPTRF gives better SMP performance than using _POTF2 routine with full format _POTRF.

[3] $nb = 2$ in Fig. 2b. In real applications $nb \approx 100$ and so the triangle holds 5050 elements out of 10000 when $nb = 100$.

(3) Using a small register block size kb as the block size for _BPTRF and then calling _BPTRF with $n = nb$ degrades performance over just calling Fortran codes _POTF3i with $n = nb$; in particular, calling DPOTF3c.

Conjecture (1) is true because the _GEMM flop count ratio per matrix element is $r_{32} = nb_3/nb_2$. Here nb_3 and nb_2 are the block sizes used by the Level-3 _POTF3i routines and the Level-2 _POTF2 routine. Roughly speaking the performance of Level-3 _GEMM is proportional to this ratio r_{32}.

In Experiment I, we are concerned with performance when $n \approx nb$. We demonstrate that for larger nb _POTF3i gives better performance than _POTF2 or _POTRF using _POTF2. This fact, using the results of Experiment II, implies _BPTRF and _POTRF have better performance for all n. Conjecture (2) is true for all n because besides _GEMM, both _SYRK and _TRSM have the same ratio r_{32} of Experiment II. Experiment II runs DGEMM, DTRSM and DSYRK for different M, N and K values as specified in Conjecture (1) above. Therefore, for large n, the Flop count of _POTRF and _BPTRF is almost equal to the combined Flop counts of these three Level-3 BLAS routines; the Flop count of _POTF3i is tiny by comparison.

Conjecture (3) is true because the number of subroutine calls in _BPTRF is r^2 where ratio $r = nb/kb$. Hence for $nb = 64$ and $kb = 2$ there are *over one thousand* subroutine calls to Level-3 BLAS with *every one* having their K dimension equal to kb. On the other hand, the four _POTF3i routines make *no* subroutine calls. The conclusion is that, at the register block level, the calling overhead is too high in _BPTRF. More importantly, the flop counts per BLAS calls to _SYRK, _GEMM and _TRSM are very small when their K dimension equals $\approx kb$ and kb has register block sizes; the results of Experiment III in Sections 3.6 and 3.7 experimentally verify Conjecture (3) above.

3.1 Calling Overhead for _POTRF, _BPTRF and SBPF Cholesky

The traditional Level-3 BLAS approach for LAPACK factorization routines like LU=PA, QR and Cholesky factorization was to cast as much of the computation as possible in terms of Level-3 BLAS. The API of Level-3 BLAS is full format. For Cholesky, BPF allows the use of Level-3 BLAS as well as using about half the storage of full format. SBPF is the same as upper BPF. Full format does have an advantage over BPF and SBPF in that for large n the number of Level-3 BLAS subroutines calls is much lower for full format. We now demonstrate this. Let $N = \lceil n/nb \rceil$ be the block order of the Cholesky problem. A simple counting analysis demonstrates that the number of calls for _POTRF is $\max(4(N-1), 1)$, for _BPTRF it is N^2 and for _SBPF it is $N(N+1)(N+2)/6$. Here is a breakdown of the number of calls to Factor, _SYRK, _TRSM, and _GEMM for routine _POTRF, _BPTRF and SBPF: For _POTRF they are N, $N-1$, $N-1$, $N-2$; for _BPTRF of Fig. 3 they are N, $N(N-1)/2$, $N-1$, $(N-1)(N-2)/2$; and for SBPF Cholesky of Section 1.1.1, using the upper BPF of Fig. 2b, they are N, $N(N-1)/2$, $N(N-1)/2$, $N(N-1)(N-2)/6$. In Table 1 we show values for these number of calls for three matrix orders n and eight block sizes nb that explicitly indicate the number of subroutine calls made by _POTRF, _BPTRF and SBPF. In experiments II and III of Sections 3.4 to 3.7 only routines _POTRF, _BPTRF are used.

nb	n=120				n=500				n=4000			
	N	PO	BP	SBP	N	PO	BP	SBP	N	PO	BP	SBP
2	60	236	3600	37820	250	996	62500	2.6E6	2000	7996	4.0E6	1.3E9
8	15	28	225	680	63	248	3969	43680	500	1996	2.5E5	2.1E7
32	4	12	16	20	16	60	256	816	125	496	15625	3.3E5
64	2	4	4	4	8	28	64	120	63	248	3969	43680
96	2	4	4	4	6	20	36	56	42	164	1764	13244
120	1	1	1	1	5	16	25	35	34	132	1156	7140
128	1	1	1	1	4	12	16	20	32	124	1024	5984
200	1	1	1	1	3	8	9	10	20	76	400	1540

Table 1. Number of Subroutine Calls for Full Format DPOTRF, BPF DBPTRF and SBPF Cholesky.

3.2 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.

We do our calculations in DOUBLE PRECISION. The DOUBLE PRECISION names of the subroutines used in this section are DPOTRF and DPOTF2 from the LAPACK library and four simple Fortran Level-3 DPOTF3i routines described below. These four routines are subroutines used by DBPTRF for matrix orders above size 120. LAPACK DPOTF2 is a Fortran routine that calls Level-2 BLAS routine DGEMV and it is called by DPOTRF. DPOTRF and DBPTRF also call Level-3 BLAS routines DTRSM, DSYRK, and DGEMM. DPOTRF also calls LAPACK subroutine ILAENV which sets the block size used by DPOTRF. As described above the four Fortran routines DPOTF3i are a new type of Level-3 BLAS called FACTOR BLAS.

Table 2 contains 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.

This table has thirteen columns. The first column shows the matrix order. The second column contains results for the vendor optimized Cholesky routine DPOTRF and the third column has results for the Recursive Algorithm [Andersen et al. 2001].

Column four contain 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 routines are called by DBPTRF with block size $nb = 64$. We now denote these four routines by suffixes a,b,c,d.

The resolution of our timer used to obtain the results in Table 2 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; eg, for $n = 40$ the results for columns 4 and 5 should have column 4 less than column 5.

3.3 Interpretation of Performance Results for Experiment I

There are five Fortran routines used in this study besides DPOTRF and DBPTRF:

Mat	Ven	Recur	dpotf2		2x2 w. fma		1x4		2x4		2x2	
ord	dor	sive			8 flops		8 flops		16 flops		8 flops	
	lap	lap	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

Table 2. Performance in Mflop/s of the Kernel Cholesky Algorithm. Comparison between different computers and different versions of subroutines.

(1) The LAPACK routine DPOTF2: The fourth and fifth columns have results of using routine DPOTRF to call DPOTF2 and routine DPOTF2 directly: these results are tabulated in the fourth and fifth columns respectively.

(2) The 2×2 blocking routine DPOTF3a is specialized for the operation FMA $(a \times b + c)$ using seven floating point registers (FPRs). This 2×2 blocking DPOTF3a routine replaces routine DPOTF2: these results are tabulated in the sixth and seventh columns respectively.

(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. This 1×4 blocking routine DPOTF3b replaces routine DPOTF2: these results are tabulated in the eighth and ninth columns respectively.

(4) The 2×4 blocking routine DPOTF3c uses fourteen FPRs. This 2×4 blocking routine DPOTF3c replaces routine DPOTF2: these results are tabulated in the

tenth and eleventh columns respectively.

(5) The 2×2 blocking routine DPOTF3d; see Fig. 4. It is not specialized for the FMA operation and uses six FPRs. This 2×2 blocking routine DPOTF3d replaces DPOTF2: these performance results are tabulated in the twelfth and thirteenth columns respectively.

Before continuing, we note that Level-3 BLAS will only be called in columns 4, 6, 8, 10, 12 for block sizes 72 and 100. This is because ILAENV has set the block size to be 64 in our study. Hence, Level-3 BLAS only have effect on our performance study in these five columns.

The DPOTF3c code with submatrix blocks of size 2×4, see column eleven, is remarkably successful for the Sun (Newton), SGI (Freke), IBM (Huge) and Quad Core Xeon (Zoot) computers. For all these four platforms, 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 IBM vendor's optimized codes, except for $n = 40$, are superior to it on this IBM platform. The 2×2 DPOTF3d code in column thirteen, not prepared for the FMA operation, is superior on the Intel Xeon (Battle) computer. The 1×4 DPOTF3b in column nine is superior on the Dual Core AMD (Nala) platform. All the superior results are colored in red.

These performance numbers reveal an innovation about the use of Level-3 Fortran DPOTF3(a,b,c,d) codes over use of Level-2 LAPACK DPOTF2 code. We demonstrate why in the next two paragraphs.

The results of columns 10 and 11 are about the same for $n = 40$ and $n = 64$. For column 10 some additional work is done. DPOTRF calls ILAENV which sets $nb = 64$. It then calls DPOTF3c and returns after DPOTF3c completes. For column 11 only DPOTF3c is called. Hence column 10 takes slightly more time than column 11. However, in column 10, for $n = 72$ and $n = 100$ DPOTRF, via calling ILAENV, still sets $nb = 64$ and then DPOTRF does a Level-3 blocked computation. For example, take $nb = 100$. With $nb = 64$ DPOTRF does a sub blocking of nb sizes equal to 64 and 36. Thus, DPOTRF calls Factor(64), DTRSM(64,36), DSYRK(36,64), and Factor(36) before it returns. The two Factor calls are to the DPOTF3c routine. However, in column 11, DPOTF3c is called only once with $nb = 100$. In columns ten and eleven performance is always increasing over doing the Level-3 blocked computation of DPOTRF. This means the DPOTF3c routine is out performing DTRSM and DSYRK.

Now, take columns four and five. For $n = 40$ and $n = 64$ the results are again about equal for the reasons cited above. For $n = 72$ and $n = 100$ the results favor DPOTRF with Level-3 blocking except for the Zoot platform and the Battle platform for $n = 72$. The DPOTF2 performance is decreasing relative to the blocked computation as n increases from 64 to 100. The opposite 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 platform is IBM Huge for columns (6,7), (8,9), (12,13). This platform has 32 FPRs. Column (10,11) is using only 14 FPRs and DPOTF3c exhibits the favorable pattern. The three exceptional columns for DPOTF3(a,b,d) use 7, 8 and 6 FPRs respectively.

An essential conclusion is that the faster four Level-3 DPOTF3i Fortran routines really help to increase performance for all n if used by DPOTRF instead of using DPOTF2. Here is why. Take any n for DPOTRF. DPOTRF can choose a larger block size nb and it will do a blocked computation with this block size for $n \geq nb$. All three BLAS subroutines, DGEMM, DSYRK and DTRSM, of DPOTRF will perform better by calling DPOTRF with this larger block size. See the last paragraph of Section 3 for a reason.

The paper [Andersen et al. 2005] gives large n performance results for BPHF where nb was set larger than 64. The results for $nb = 100$ were much better. The above explanation in Section 3 explains why this was so. It also confirms the results of [Whaley 2008]; Finally see Section 1.1.1 and the remaining Sections of 3 where we give 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 but many users do not do this finer tuning. 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. 2007, Algorithm 685]. The code for _POTRF and its subroutines is available from the LAPACK package [Anderson et al. 1999].

3.4 Performance Preliminaries for Experiments II and III

Due to space limitations we only consider two processors: a Sun-Fire-V440, 1062 MHz, 4 CPU processor and an Intel/Nehalem X5550, 2.67 GHz, 2 x Quad Core, 4 instruction/cycle processor. The results of Experiments II and III are given in Sections 3.5 to 3.7.

For Experiment II, see Table 3, DGEMM is run to compute $C = C - A^T B$ for $M = K = nb$ and $N = n$ where usually $N \gg nb$. The case used here of $A^T B$ is a good case for DGEMM as the rows of A and columns of B are both stride one. For this case of $A^T B$ each $c_{ij} \in C$ is loaded, K FMA operations are performed and then c_{ij} is stored. One expects that as K increases DGEMM performance increases when K is sufficiently small.

Table 3 also gives performance of DTRSM for $M, N = nb, n$ and DSYRK for $N, K = nb, nb$. The values chosen were n = 100, 200, 500, 1000, 2000, 4000 and nb = 2, 40, 64, 72, 80, 100, 120, 200. The matrix form parameters for DGEMM are `'Transpose', 'Normal'`, for DTRSM are `'Left', 'Upper', 'Transpose', 'No Unit'`, and for DSYRK are `'Upper', 'Transpose'`.

For experiment III in Table 4, we mostly consider performance of DPOTRF and DBPTRF using upper BPF for matrix orders n = 250, 500, 720, 1000, 2000, 4000. For each matrix order n we use three values of block size nb = 2, 64, 120. Table 4 has twelve columns. Columns 1 and 2 give n and nb. Columns 3 to 12 give performance in MFlops of various Cholesky Factorization routines run for these n, nb values using either full format, upper BPF or Recursive Full Packed (RFP) format. The Cholesky routines are DPOTRF, DBPTRF and RFP Cholesky. Column three gives vendor LAPACK (vLA) DPOTRF performance. Column four gives recursive performance of RFP Cholesky; see [Andersen et al. 2001]. Column five gives LAPACK DPOTRF using DPOTF2 and column six gives performance of calling only

Sun-Fire-V440, 1062 MHz, 8GB memory, Sys. Clock 177 MHz, using 1 out of 4 CPU's. SunOS sunfire 5.10, Sunperf BLAS													
nb	MM	TS	MM	TS	MM	TS	MM	TS	MM	TS	MM	TS	SYRK
	n = 100		n = 200		n = 500		n = 1000		n = 2000		n = 4000		n = nb
1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	57	31	70	38	74	44	84	51	90	58	96	65	.059
40	1190	760	1216	841	1225	784	1231	759	1231	744	1219	777	579
64	1528	1046	1313	1102	1572	1044	1565	1035	1474	1010	1407	956	741
72	1688	1182	1725	1209	1654	1148	1566	1139	1465	1082	1475	1018	872
80	1721	1219	1753	1238	1674	1192	1515	1196	1515	1143	1519	1073	994
100	1733	1226	1771	1254	1733	1213	1593	1235	1593	1195	1586	1161	968
120	1778	1270	1798	1345	1738	1293	1641	1297	1641	1248	1657	1231	1129
200	1695	1307	1759	1358	1748	1379	1756	1375	1756	1360	1777	1357	1096
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Table 3. Performance in Mflop/s for large n and various nb of DGEMM, DTRSM and DSYRK. The headings MM and TS are abbreviations for GEMM and TRSM.

DPOTF2. The factor kernels DPOTF3a,c are used in columns 7 to 9 and 10 to 12. The three headings of each triple (FLA,BPF,fac) mean Full format LAPACK using DPOTRF; Cholesky DBPTRF factorization using upper BPF, see Figs. 2b and 3; and using only DPOTF3i, `i = a,c`. Column one of each triple uses full format DPOTRF with DPOTF3a,c instead of using DPOTF2. Column two of each triple uses upper BPF DBPTRF with DPOTF3a,c. Column three of each triple uses only full format DPOTF3a,c.

3.5 Interpretation of Performance Results for Experiment II and partly Experiment III

We first consider Experiment II; see Table 3. As nb increases performance of DGEMM and DTRSM increases except at $nb = 200$ where it is leveling off. This increase is very steep for small nb values. The experiment also verifies that using a tiny register block size $kb = 2$ for the K dimension of the Level-3 BLAS DGEMM, DTRSM and DSYRK gives very poor performance. There are two explanations: First, the flop count is too small to cover the calling overhead cost and second, a tiny K dimension implies Level-2 like performance. In any case, the assertions of Conjecture 1 and partly of 3 have been verified experimentally on this processor for DGEMM, DTRSM and DSYRK. Entries for $n = 100$, see columns 2, 3 and 14 of Table 3, and row entries $nb = 64,\ 120$ of Table 3 show performance gains of 16%, 21%, 52% respectively for DGEMM, DTRSM, DSYRK.

We were only given one CPU for Experiment II so parallelism was not exploited. Nonetheless, look at columns 2 and 12 of Table 3 corresponding to DGEMM performance at $n = 100$ and 4000. On a multi-core processor, one could call DGEMM forty times in parallel using upper BPF and get about a forty-fold speed-up as upper BPF stores the B and C matrices of DGEMM as 40 disjoint concatenated contiguous matrices. For full format the B and C matrices do not have this property; DGEMM would require data copy and its parallel performance would probably degrade sharply.

At the i^{th} block step of DBPTRF, see Fig. 3, DSYRK must be called $i - 1$ times in a loop. This is why we did not include performance runs for DSYRK for $(i - 1)nb \times nb$ size A. Nonetheless, DPOTRF calls DSYRK only once during its

Sun-Fire-V440, 1062 MHz, 8GB memory, Sys. Clock 177 MHz, using 1 out of 4 CPU's, SunOS sunfire 5.10, Sunperf BLAS											
n	nb	vLA	rec	dpotf2		2x2 w. fma			2x4		
				FLA	fac	FLA	BPF	fac	FLA	BPF	fac
1	2	3	4	5	6	7	8	9	10	11	12
250	2	1006	1017	653	1042	641	179	1229	655	179	1367
	64	1015	1026	1067	1022	1102	1074	1258	1117	1097	1436
	120	988	1027	1014	1032	1059	1091	1256	1105	1102	1431
500	2	1109	1097	745	1130	743	204	1379	747	204	1527
	64	1162	1127	1224	1130	1256	1251	1378	1194	1252	1493
	120	1208	1089	1192	1126	1233	1233	1393	1243	1277	1552
720	2	1184	1126	711	622	695	178	937	705	176	1149
	64	1180	1113	1220	613	1270	1241	959	1239	1296	1009
	120	1236	1155	1279	688	1242	1322	910	1303	1329	1024
1000	2	1158	1067	504	270	598	142	630	558	134	607
	64	1149	1080	1162	270	1157	1252	554	1175	1194	775
	120	1278	1099	1231	274	1254	1327	623	1242	1302	644
2000	2	1211	1117	473	226	462	101	489	460	101	480
	64	1169	1114	1241	214	1223	1193	477	1265	1193	481
	120	1139	1086	1280	230	1318	1365	569	1296	1365	460
4000	2	1119	1102	385	207	448	99	432	445	98	530
	64	1213	1109	1226	239	1238	1216	499	1270	1179	545
	120	1210	1127	1423	219	1416	1495	501	1417	1489	516
1	2	3	4	5	6	7	8	9	10	11	12

Table 4. Performance in Mflop/s on a single CPU processor, for large n and various nb, of DPOTRF and DBPTRF using DPOTF2 and DPOTF3a,c on a Sun four CPU processor .

i^{th} block step; this is an example where full format has less calling overhead than BPF; see Table 1.

3.6 Interpretation of Performance Results for Experiment III using _POTRF and _BPTRF on the Sunfire Processor

As mentioned in Section 3.5 we were only given one processor for this processor. Table 4 concerns performance results for DPOTRF using DPOTF2 and DBPTRF using the two best performing routines DPOTF3a,c. Note that columns 3, 4, 6, 9 and 12 should have the same MFlops value for three rows of the same n value as all of these column values do not depend on nb; the different values seen show the resolution of our timer. For $n > 500$ we see that $nb = 120$ gives better performance than the default block size $nb = 64$ for both full format and BPF computations. For $n \leq 500$ and $nb = 64$, 120 the performance results for DPOTRF and DBPTRF are about equal. For DPOTRF, performance at $nb = 64$ is slightly better than at $nb = 120$.

In [Whaley 2008], it is observed that as n increases performance of DPOTRF will increase if nb is increased. See also the last paragraph of Section 3. We also see this in Table 4. For DBPTRF, performance results at $nb = 120$ are always better than at $nb = 64$ for all values of $n > 500$. This result was also experimentally verified in Section 1.1.1. This suggests that setting a single value of nb for all n for BPF is probably a good strategy. For columns 5, 7 and 10 we see that DPOTRF performance is about the same using DPOTF2 and DPOTF3a,c. This is expected as these three routines contribute only a very tiny amount of flop count to the overall flop count of _POTRF when n is large.

For $n = 250$, 500, DBPTRF performance is maximized using DPOTF3a,c alone; see columns 9 and 12. This is not true for DPOTRF; see also Experiment II. In Section 2.1, we saw that maximum cache usage of DPOTF3i was $nb^2/4$. This fact helps explain the results of columns 9 and 12 for $n = 250$, 500.

Finally, we discuss the negative performance results when using $nb = 2$ which is a register block size. The main reason for poor performance is the amount of subroutine calls for both DPOTRF and DBPTRF; see Table 1. Each call has a tiny flop count and consequently the calling overhead results in severely degrading their MFlops; see columns 5, 7, 10 and 8, 11. The number of calls of upper BPF is N^2 and for full format is $\max(4(N-1), 1)$. The quadratic nature of the calls is readily apparent in columns 8 and 11.

We briefly mention columns 3 and 4. The performance of vendor code (vLA) is slightly better than LAPACK code. The BPF codes are generally the best performing codes. The recursive codes of column 4 perform quite well.

3.7 Interpretation of Performance Results for _POTRF and _BPTRF for the Intel/Nehalem Processor

For this processor it is important to realize that vendor BLAS for this platform have been optimized for parallelism. Thus we will see an example where DPOTF2 is outperforming DPOTF3a,b. The reason for this is that DPOTF2 calls Level-2 DGEMV which has been parallelized by the vendor. In Table 5, we mostly consider performance of DPOTRF and DBPTRF for matrix orders $n = 250, 500, 1000, 2000, 4000$. For each matrix order n we use six values of block size $nb = 2, 8, 32, 64, 96, 120$. Table 5 has twelve columns arranged exactly like Table 4. Therefore, we only describe these table differences; see Section 3.4 for a description of Table 4. The factor kernels are DPOTF3a,b instead of DPOTF3a,c and these results are given in columns 7 to 9 and 10 to 12. Column three of each triple uses only full format DPOTF3a,b. The reader is alerted to re-read the paragraph on the bottom of page six and the top of page seven as a preview to understanding how DPOTF2 can outperform DPOTF3i in a parallel environment that uses optimized Level-2 BLAS. Again columns 3, 4, 6, 9 and 12 should have identical values for a given n row value as none of these column values depend on nb. The variability of these performance numbers indicates what the variability is in our timer CPU-SEC.

Note that DPOTF2 is now giving better performance than DPOTF3a,b for $n \leq 2000$. For $n = 4000$ DPOTF3a outperforms DPOTF2. The reason for this is that DGEMV has been parallelized; DPOTF3a,b were not. DPOTF2 and DPOTF3a,b do not use cache blocking. As n increases performance degrades as more matrix data resides in higher level caches or memory than when cache blocking is used; thus many more cache misses occur and each miss penalty is huge.

We used small nb values to demonstrate the effect of calling overhead of DPOTRF and DBPTRF. For block size $nb \geq 8$ performance is quite good even for BPF; BPF calling overhead is $O(N^2)$ whereas DPOTRF calling overhead is $O(N)$; see Table 1 for details. We did *not* run an experiment for SB format where the calling overhead is $O(N^3)$. Note however that, for all value of n, multi-core SB (or upper BPF) is best as it exposes more usable parallelism; see remark made in paragraph 2 of

Intel/Nehalem X5550, 2.67 GHz, 2x Quad Core,											
Portland compiler and BLAS, Double Precision.											
n	nb	vLA	rec	dpotf2		2x2 w. fma			1x4		
				FLA	fac	FLA	BPF	fac	FLA	BPF	fac
1	2	3	4	5	6	7	8	9	10	11	12
250	2	6222	6154	2841	4168	2933	479	3465	2908	478	3812
	8	6223	6158	5046	4124	5298	3567	3467	5241	3559	3811
	32	6235	6147	6326	4138	6599	6313	3472	6604	6306	3802
	64	6224	6155	6307	4135	6349	6316	3470	6489	6466	3801
	96	6218	6154	6184	4176	6056	6082	2888	6387	6410	3776
	120	6239	6213	5763	4167	5482	5496	3471	5925	5977	3789
500	2	7943	7776	3213	4280	3225	533	3544	3215	533	3828
	8	7961	7791	6129	4280	6227	4312	3543	6199	4302	3828
	32	7958	7782	7872	4282	7993	7464	3546	7995	7412	3828
	64	7960	7789	7994	4280	7977	7885	3542	8050	7908	3826
	96	7955	7792	7893	4279	7820	7854	3544	7994	7972	3826
	120	7984	7832	7730	4275	7624	7668	3543	7826	7744	3827
1000	2	9078	8985	3476	4204	3487	567	3520	3482	567	3791
	8	9081	8985	6891	4204	6869	4833	3521	6909	4831	3791
	32	9089	8985	8945	4194	8944	8175	3522	8983	8189	3788
	64	9080	8983	9192	4204	9127	8884	3521	9186	8917	3791
	96	9080	8985	9212	4204	9160	9107	3521	9256	9157	3789
	120	9102	9007	9152	4205	9103	9084	3522	9218	9044	3792
2000	2	9954	9862	3129	3260	3100	580	3191	3131	580	2973
	8	9955	9859	6949	3228	7048	5024	3185	6972	5024	2963
	32	9953	9860	9360	3228	9392	8569	3178	9368	8496	2965
	64	9959	9845	9773	3232	9773	9459	3194	9754	9415	2992
	96	9947	9860	9862	3206	9908	9792	3184	9912	9760	2956
	120	9964	9870	9875	3214	9921	9796	3187	9930	9783	3001
4000	2	10581	10558	2619	2212	2612	580	2798	2620	580	2094
	8	10569	10551	6557	2207	6534	4957	2803	6533	4940	2095
	32	10576	10544	9430	2211	9440	8619	2809	9432	8576	2096
	64	10581	10540	10104	2206	10099	9654	2807	10093	9681	2093
	96	10575	10548	10300	2214	10356	10096	2804	10351	10113	2091
	120	10583	10551	10381	2208	10431	10198	2807	10431	10286	2098
1	2	3	4	5	6	7	8	9	10	11	12

Table 5. Performance in Mflop/s for large n and various nb of DPOTRF and DBPTRF using DPOTF2 and DPOTF3a,b on an Intel/Nehalem Processor .

Section 3.5. However, for $nb = 2$ one can see the effect of $O(N^2)$ calling overhead for BPF DBPTRF over the only $O(N)$ calling overhead for full format DPOTRF for all value of n; see columns 5, 7 and 10 for DPOTRF and 8 and 11 for DBPTRF.

We now further discuss *full format* DPOTRF. So, any thing we say here has nothing to do with DBPTRF which uses BPF. We mainly discuss DPOTRF performance using DPOTF2 (see column 5), DPOTRF performance using DPOTF3a instead of DPOTF2 (see column 7) and DPOTRF performance using DPOTF3b instead of DPOTF2 (see column 10). For $n = 250$, $nb = 32$ is the best block size for DPOTRF. Also, even though DPOTF2 shows better MFlops (see column 6) than DPOTF3b (see column 12), DPOTRF using DPOTF3b outperforms DPOTRF using DPOTF2 by over 250 MFlops. We think there are two reasons:

(1) Full Format DPOTRF A lays out the submatrices A_{ii} holding its diagonal

blocks in at least $n \times nb$ storage. So, accessing any A_{ii} will cause more cache misses than if A_{ii} were held in nb^2 contiguous storage.

(2) This is a parallelization issue: Level-3 BLAS are competing with Level-2 DGEMV over how a quad core will be utilized for parallelism.

Let us briefly discuss item (2) above before going on. DGEMV has been parallelised by the vendor. For large n all of the available parallelism of the platform should probably be used by Level-3 BLAS DGEMM, DTRSM and DSYRK. Using some of the available parallelism in DGEMV is probably a suboptimal performance choice when running DPOTRF.

For $n = 500$, $32 \leq nb \leq 64$ is the best block size when using DPOTF3a,b; see columns 7 and 10. Using DPOTF2, DPOTRF has slightly better performance when $nb = 64$. DPOTRF has about equal performance with DPOTF3a,b and DPOTF2; see columns 7, 10 and 5. However, the values in columns 10 are better than the values in column 5. For DPOTRF performance at $n = 1000$, the best block size using either DPOTF2 or DPOTF3a,b is $nb = 96$. The best performance number occurs in column 10 although all three values are about equal. For DPOTRF performance at $n = 2000$, the best block size using either DPOTF2 or DPOTF3a,b is $96 \leq nb \leq 120$. Again performance numbers favor column 10 slightly; however, all three values are about equal. Finally, for DPOTRF performance at $n = 4000$, the best block size with either DPOTF2 or DPOTF3a,b is slightly favoring $nb = 120$ but any block size between $64 \leq nb \leq 120$ is about equally good. Again performance numbers favor column 10 slightly; however, all three values are about equal.

This same type of result of DPOTRF performance increasing for n increasing as nb increased was also observed in [Whaley 2008].

Now we discuss BPF and DBPTRF performance. One can see that $nb = 96$ is a near optimal performance choice for $250 \leq n \leq 4000$ for DBPTRF using DPOTF3a,b. The performance of DBPTRF with DPOTF3a is about equal to the performance of DBPTRF with DPOTF3b with DPOTF3b numbers being slightly better. Finally, for $250 \leq n \leq 4000$, performance of DPOTRF using DPOTF2 is slightly better overall than DBPTRF performance. We briefly suggest why. There are more subroutine calls to Level-3 BLAS using BPF. Roughly speaking, there is less opportunity to parallelize DGEMM because the size of the submatrices per call is smaller with BPF than with full format. Nonetheless, BPF and DBPTRF also admit good multi-core implementations whereas such implementations for DPOTRF using only standard full format will not perform well.

4. SUMMARY AND CONCLUSIONS

We have shown that four simple Fortran codes DPOTF3i produce Level-3 Cholesky factorization routines that perform better than the Level-2 LAPACK DPOTF2 routine. We have also shown that their use enables LAPACK routine DPOTRF to increase its block size nb. Since nb is the k dimension of the _GEMM, _SYRK and _TRSM Level-3 BLAS, their SMP performance will improve and hence the overall performance of SMP _POTRF will improve. We provided “three performance conjectures” with explanations on why they were “true”. Also, three performance studies were conducted which “verified” these conjectures. Our results corroborate results that were observed by [Andersen et al. 2005; Whaley 2008]. It was seen that

DBPTRF performance was less sensitive to the choice of one nb for an entire range of n values. For DPOTRF using DPOTF2 one needed to increase nb as n increased for optimal performance whereas for DBPTRF using DPOTF3i usually a single nb value gave uniformly good performance.

We used BPF format in this paper. It is a generalization of standard packed format. We discussed lower BPF format which consisted of $N = n/nb$ rectangular blocks whose LDA's were $n = j \cdot nb$ for $0 \leq j < N$. We showed that upper packed format had the additional property that its rectangular blocks were really a multiple number of $i = N - j$ square blocks for rectangle j. In all there are $N(N+1)/2$ SB. We gave LAPACK _POTRF and _PPTRF algorithms using BPF and showed that these codes were trivial modifications of current _POTRF algorithms. In the multicore era it appears that SB format will be the data layout of choice. Thus, we think that for upper BPF format the current Cell implementations of [Kurzak et al. 2008] will carry over with trivial modifications. The very recent papers [Agullo et al. 2010; Bouwmeester and Langou 2010] actually demonstrate that this remark is true.

We also indicated how a rectangular block could be transformed inplace to a multiple of square blocks by a vector inplace transpose algorithm. Another purpose of our paper is to promote the new **Block Packed Data Format** storage or variants thereof; see Section 1.3. BPF algorithms are variants of the BPHF algorithm and they use slightly more computer memory than $n\times(n+1)/2$ matrix elements. They usually perform better or equal to the full format storage algorithms. The full format algorithms require additional storage of $(n-1)\times n/2$ matrix elements in the computer memory but never reference these elements. Finally, full format algorithms and their related Level-3 BLAS are no longer being used on multi-core processors. For symmetric and triangular matrices the format of choice is SBPF which is the same as upper BPF.

5. ACKNOWLEDGMENTS

The performance results in this paper were obtained on nine computers, an IBM, SGI, two SUNs, two AMD, and three Intel. The IBM, AMD/Opteron 2356, Intel/Nehalem X5550, and SGI machines belong to the Center for Scientific Computing at Aarhus University (Denmark), the two SUN machines to the Danish Technical University (Denmark), and the AMD/Opteron 265 and two Intel-Xeon machines belong to the Innovative Computing Laboratory of the University of Tennessee at Knoxville (USA). We would like to thank Bernd Dammann and Poul Erik Thamdrup for consulting help on the SUN systems; Niels Carl W. Hansen for consulting help on the IBM, SGI, one AMD, and one Intel systems; and Paul Peltz and Barry Britt for consulting help on the AMD and two 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 (Feb. 22), U of Tenn at Knoxville and U of Colorado at Denver.

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 Mathemat-*

ical 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.

Bouwmeester, H. and Langou, J. 2010. A Critical Path Approach to Analyzing Parallelism of Algorithmic Variants. Application to Cholesky Inversion. arXiv: 1010.2000v1 (Oct. 11), University of Colorado at Denver.

D'Azevedo, E. and Dongarra, J. J. 1998. Packed storage extension of ScaLAPACK. ORNL Report 6190 (May), 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. Soft. 16*, 1 (March), 18–28.

Elmroth, E., Gustavson, F. G., Kågström, B., and Jonsson, I. 2004. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review 46*, 1 (March), 3–45.

Gunnels, J. A., Gustavson, F. G., Pingali, K., and Yotov, K. 2007. Is cache-oblivious dgemm viable. In *Applied Parallel Computing, State of the Art in Scientific Computing, PARA 2006*, Volume LNCS 4699 (Springer-Verlag, Berlin Heidelberg, 2007), pp. 919–928. Springer.

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. 2000. New Generalized Data Structures for Matrices Lead to a Variety of High-Performance Algorithms. In P. T. P. T. Ronald F. Boisvert Ed., *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, Number TC2/WG2.5 in IFIP (Ottawa, Canada, Oct. 2000), pp. 211–234. Kluwer Academic Publishers.

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. 2008. The relevance of New Data Structure Approaches for Dense Linear Algebra in the new Multicore / Manycore Environments. IBM RC Report 24599 (July), IBM Research, Yorktown.

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. and Jonsson, I. 2000. Minimal storage high performance cholesky via blocking and recursion. *IBM Journal of Research and Development 44*, 6 (Nov), 823–849.

Gustavson, F. G., Karlsson, L., and Kågström, B. 2011. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. *ACM Transactions on Mathematical Software 37*, xx–xx+33.

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.

Gustavson, F. G. and Swirszcz, T. 2007. In-place transposition of rectangular matrices. In *Applied Parallel Computing, State of the Art in Scientific Computing, PARA 2006*, Volume LNCS 4699 (Springer-Verlag, Berlin Heidelberg, 2007), pp. 560–569. Springer.

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.

IBM. 1986. *Engineering and Scientific Subroutine Library, Guide and Reference*. First Edition (Program Number 5668-863).

KARLSSON, L. 2009. Blocked in-place transposition with application to storage format conversion. Report # uminf - 09.01. issn 0348-0542. dept. of computer science (Jan), 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 (CANA08)* (2008).