New Generalized Data Structures for Matrices Lead to a Variety of High Performance Algorithms

Fred G. Gustavson
IBM T.J. Watson Research Center
Yorktown Heights NY, 10598
1-914-945-1980
email: gustav@watson.ibm.com

October 19, 2001

Abstract

We describe new data structures for full storage of general matrices that generalize the current storage layouts of the Fortran and C programming languages. We also describe new data structures for full and packed storage of dense symmetric/triangular arrays that generalize both full and packed storage. Using the new data structures, one is led to several new algorithms that save "half" the storage of full format for symmetric matrices and outperform the current block based level 3 LAPACK, [16], algorithms done on full format symmetric matrices. We concentrate on the simplest forms of the new algorithms and show that they are a direct generalization of LINPACK, [17], algorithms. This means that level 3 BLAS are *not* required to obtain level 3 performance. The replacement for level 3 BLAS are the so-called "kernel routines," see [1], and on IBM platforms they can be generated from simple, textbook-type codes (i.e. "vanilla" codes), by the XLF Fortran compiler. On the IBM Power3 processor, with a peak performance of 800 MFLOPS, our implementation of Cholesky factorization reaches over 720 MFLOPS when $N \geq 200$ and then performance climbs to 735 MFLOPS at $N = 400$. By way of comparison, conventional full format LAPACK `DPOTRF`, employing ESSL BLAS, does not achieve 600 MFLOPS before $N = 600$ and reaches a peak of only 620 MFLOPS. To obtain this result we used simple, square blocked, full matrix data formats where the blocks themselves are stored in column-major (Fortran) order or row-major (C) order. The simple algorithms of LU factorization with partial pivoting for this new data format are a direct generalization of the LINPACK algorithm, `DGEFA`. Again, no conventional level 3 BLAS are required; the replacements are, again, kernel routines. Programming for square blocked, full matrix format can be accomplished in standard Fortran, through the use of three- and four-dimensional arrays. Thus, no new compiler support is necessary. For symmetric indefinite factorization we introduce a novel data format, called Lower Packed Blocked Overlapped Format. Using it, we describe new algorithms for solving symmetric indefinite systems of equations. These new algorithms are now part of ESSL, [15]. Also, we mention that other, more complicated, algorithms are possible; e.g., recursive algorithms. The recursive algorithms are also easily programmed via the use of tables that address where the blocks are stored in a two-dimensional recursive block array. Finally, we describe block hybrid formats. These formats allow one to use *no* additional storage over conventional (full and packed) matrix storage. This means the new algorithms are *completely portable*.

1 Introduction

In [9, 12], Recursive Blocked Data formats were introduced as a replacement for standard Fortran/C array storage. One of the key insights was to see that storing a matrix as a collection of submatrices (e.g., square blocks of size `NB`) led to very high performance on today's, RISC type, processors. Let a rectangular matrix, A, have `M` rows and `N` columns. For clarity, assume `M` $= m *$ `NB` and `N` $= n *$ `NB`. A consists of mn submatrices of size `NB` by `NB`. In [9, 12] we demonstrated that recursion (i.e., divide-and-conquer) should be used to order these mn blocks in storage. This storage arrangement leads to L2, L3, and memory blocking automatically. However, the ordering of the blocks is non-linear and tables are needed to properly address these mn blocks.

A simpler way to order the mn blocks is standard Fortran/C order; i.e., store the mn blocks either in column-major or row-major order. If one applies a recursive algorithm to this simpler block data layout one approximates an automatic blocking for L2, L3, memory; i.e., performance for this layout will approximate recursive blocked data layout, [9, 12]. The main benefit of the simpler data layout is that addressing of an arbitrary (i, j) element of `A(0:M-1,0:N-1)`, namely, $a(i, j)$ can be easily handled by a compiler and/or a programmer. Let $0 \leq i < \mathtt{M}$ and $0 \leq j < \mathtt{N}$ and write $i = I * \mathtt{NB} + i1$ and $j = J * \mathtt{NB} + j1$. If A is stored in column-block-major order and all blocks are in column-major order then $a(i, j)$ is in block $I + m * J$ at location $i1 + \mathtt{NB} * j1$ of that block. Also, IBM Power3 processors have 3 fixed point units which are mostly idle during floating point computations. Hence the additional index computations above can probably be overlapped resulting in little or no performance loss.

For level 3 algorithms, the basis of the ESSL (Engineering and Scientific Subroutine Library) are kernel routines that achieve peak performance when the underlying arrays fit into L1 cache, [1, 15]. If one were to adopt these new, simpler data formats then BLAS and LAPACK type algorithms will become almost trivial to write. We will discuss an example in the next paragraph, and in Sections 3, 4 and 5 we will show three detailed examples. However, before closing this paragraph, I want to argue that the combination of using the new data formats with kernel routines is general and that, for matrix factorization, it overcomes the current performance problems introduced by having a non-uniform, deep memory hierarchy. We use the Algorithms and Architecture approach, see [1], to present the argument.

1. Floating point arithmetic cannot be done unless the operands involved reside in L1 cache.

2. Two-dimensional Fortran and C arrays do *not* map nicely into L1 cache.

   (a) The best case happens when the array is contiguous and properly aligned.

   (b) At least a three-way set associative cache is required when matrix multiply is being done.

3. For peak performance, all matrix operands must be used multiple times when they enter L1 cache.

   (a) This assures that the cost of bringing a operand into cache is amortized by its multiple re-use.

   (b) Multiple re-use of all operands only occurs when all matrix operands map well into L1 cache.

4. Each scalar $a(i, j)$ factorization algorithm has a square submatrix counterpart `A(I:I+NB-1,J:J+NB-1)` algorithm.

   (a) Golub and Van Loan's "Matrix Computations" book, [8].

   (b) The LAPACK library.

5. Some submatrices are both contiguous and fit into L1 cache.

6. Dense matrix factorization is a level 3 computation.

   (a) Dense matrix factorization, in this context, is a series of submatrix computations.

   (b) Every submatrix computation (executing any kernel routine) is a level 3 computation.

   (c) A level 3 computation is one in which each matrix operand get used multiple times.

7. Map the input Fortran/C array (matrix A) to a set of contiguous submatrices that each fit into L1 cache.

8. Apply the appropriate submatrix algorithm.

Points 1 to 3 are architecture facts about many of today's processors. Points 4 to 6 are dense linear algebra algorithms facts. The book, [8], point 4a, gives a detailed listing of the scalar algorithms and describes (with references to the research literature) their block submatrix counterparts. The LAPACK library, point 4b, gives code for the submatrix counterpart algorithms. However, the block submatrix codes of point 4b use Fortran and C to input their matrices, so point 5 does *not* hold. See page 739 of [9] for more details. Point 5 does hold for the new data structures described here. Assuming both points 5 and 6 hold, we see that point 3 holds for every execution of the kernel routines that make up the factorization algorithm. This implies that near peak performance will be achieved. Points 7 and 8 suggest an obvious algorithmic change that is justified by points 1 to 6. Point 7 is pure overhead for the new algorithms. Using the new data formats reduces this cost to zero. By only doing point 8 we see that we get near peak performance as every subcomputation of point 8 is a point 6b computation. Note that each kernel call of the submatrix algorithm is a level 3 call and so, on average, every scalar of each submatrix gets used multiple times. There is one type of kernel routine that deserves special mention. It is the factor kernel. Neither LAPACK nor the research literature treat factor kernels in sufficient depth. For example, LAPACK factor routines are level 2 routines; they are named with the suffix TF2, and they call level 2 BLAS repetitively. On the other hand ESSL, [1], and more recently, [5, 9, 12], where recursion is used, have produced level 3 factor kernels. In the above argument we did not mention L2, L3 cache, main memory, or out-of-core. We argue that by storing the set of contiguous matrices recursively and that by using a related recursive algorithm that one automatically blocks for all these higher levels of the memory hierarchy.

Take any vanilla code, say Gaussian elimination with partial pivoting or the QR factorization of an M by N matrix, A. This code has a block equivalent where the stride one distance is NB^2 . The row stride for the vanilla code is LDA. For the block equivalent it is m*NB^2. It is quite easy to derive the block equivalent code from the vanilla code. In the vanilla code a floating point operation is usually a Fused Multiply Add (FMA), ($c = c + ab$), whose block equivalent is a call to a DGEMM kernel. Similar analogies exist; e.g., for $b = b/a$ or $b = b * a$, we have either a DTRSM or a DTRMM kernel. In the simple block equivalent codes we are led to one of the variants of IJK order, [4]. For these types of algorithms the BLAS are simply calls to kernel routines. In reality, the BLAS disappear entirely. However, more complicated algorithms can be employed; e.g., recursive algorithms. In that case, a BLAS call accesses several submatrix blocks, [12, 9]. So, like a traditional BLAS call, a blocking routine is required to call the kernel routines multiple times on different submatrix operands, as the blocking is now variable. However, unlike a traditional BLAS routine, no data copying is performed. This results in improved performance.

In the above paragraph, I indicated that new algorithms can be obtained from simple dense linear algebra vanilla codes if one first introduces the new data formats. One way to do this would be to ask the user to input his data in standard Fortran or C order with some additional storage appended below his standard array. Then the standard Fortran or C array could be transformed, inplace, to the block square format. Next, the block equivalent of the vanilla code (with calls to standard ESSL type kernels) could be performed on the transformed square block column format. The performance should be superb. For example, on a 200 MHz IBM Power3 with a peak performance of 800 MFLOPS, the performance of Cholesky factorization at order $N \geq 200$ is over 720 MFLOPS and then reaches 735 MFLOPS at $N = 500$. We did not include the cost of transforming the data to square blocked packed format. Using conventional full format LAPACK DPOTRF with ESSL BLAS one first gets to 600 MFLOPS at $N \geq 600$ and only reaches a peak of 620 MFLOPS.

There is great resistance to changing data formats that have existed a long time. This is especially so for major programming languages such as Fortran and C; here we are considering the storage layout for a two-dimensional array. Also, libraries for dense linear algebra; e.g., LAPACK, LINPACK, and ESSL, all support packed format symmetric/triangular arrays. A compelling reason for this resistance is portability: if one changes the input layout used by a library subroutine/function then existing software that calls that library subroutine/function will *not* be operational. So, we introduce modifications to our new matrix data formats which we call Block Hybrid Format (BHF). We claim that an isosceles trapezoid is an appropriately general shape to describe the uni-processor algorithms of dense linear algebra. This trapezoid consists of an isosceles right triangle and a rectangle. Hence, we store the triangle part of the trapezoid in packed format and the rectangle part as a general matrix (full format). Using BHF solves the portability problem in most cases.

Besides full storage, there is packed storage, which is used for symmetric/triangular arrays. The disadvantage of full storage when used for symmetric/triangular arrays is that it uses twice the memory, but its advantage is that it allows the usage of level 3 and level 2 BLAS calls. This dramatically speeds up the computation. With packed storage we can only use level 1 and packed level 2 BLAS and, consequently, performance suffers drastically for large problems on RISC systems. Thus, many users will opt for higher performance and pay the penalty of using twice the storage.

Nonetheless, there are problems where a full format array will *not* fit in memory and then packed format is the appropriate choice. We describe new data structures for full and packed storage of dense symmetric/triangular arrays that generalize both formats. Using the new data structures, one is led to several new algorithms that save "half" the storage of full format for symmetric matrices and outperform the current block based level 3 LAPACK algorithms done on full format symmetric matrices. Examples are give for Cholesky and symmetric indefinite factorization.

Now I turn briefly to SMP algorithms for dense linear algebra codes. In [5], superb SMP performance for QR factorization was obtained using recursive blocking on matrices stored in standard Fortran order. The new square blocked full order storage could be used instead of standard Fortran order. In that case, performance should improve even further. Similar statements can be made for other dense linear algebra algorithms, e.g., Cholesky and general LU factorization.

In Section 2 we describe some basic algorithmic and architectural results as a rationale for the work we are presenting. The idea is to elucidate, somewhat, the Algorithms and Architecture approach, [1]. An architecture with an FMA instruction has a real advantage and we argue this assertion in Section 2. We describe the linear transformation approach to producing dense linear algebra algorithms and use it as a foundation for producing their high performance implementations. Also, we re-introduce operation counts as an accurate measure of algorithm performance. This metric becomes meaningful largely because of the manner in which our new data structures interact with the memory hierarchy. In Section 3 we describe both simple square blocked full formats and simple square blocked full hybrid formats for dense matrix arrays. We describe an algorithm for Gaussian elimination with partial pivoting for the first type of storage. We then give performance results on an IBM Power 3 processor for the hybrid format and compare them with the LAPACK `DGETRF` algorithm. Our algorithm is *fully portable*. Recently, other new block data formats, [11], have been discovered while working on this project, both at Uni-C and at IBM Research. These have to do with triangular/symmetric arrays. Again, the major advantage to these new array formats is that they more closely mirror standard storage formats than do the recursive arrays, which store the blocks in a non-linear fashion. To be precise, these symmetric/triangular arrays store the blocks either as block columns or in standard packed format order. In Section 4 we describe square blocked packed formats for symmetric/triangular arrays and show that they generalize both the standard packed and full arrays used by dense linear algebra algorithms. We also a describe a hybrid version of the new data formats that allows our new algorithms to become *fully portable*. For each type of new storage format we describe an associated Cholesky factorization algorithm. Also, performance results on an IBM Power 3 processor for both algorithms versus LAPACK's `DPOTRF`/`DPPTRF` are presented. An example for the first algorithm, called `DPSTRF`, shows experimentally that operation counts for kernel routines and for the entire Cholesky algorithm are a very accurate measure of performance. In Section 5 we describe a new storage layout for Symmetric Indefinite Factorization. Also, we describe new algorithms that use it. They combine the advantages of both LAPACK algorithms for this problem: their factorization performance is better than that of the LAPACK full storage layout algorithms, and their memory requirement is only slightly larger than LAPACK's packed storage algorithms. Our new algorithms, called `DBSSV`, `DBSTRF`, and `DBSTRS` are now part of ESSL, [15]. On three IBM RS/6000 platforms, Power3, Power2, and PowerPC 604e, `DBSTRF` outperforms LAPACK's `DSYTRF` by about 20%, and `DBSTRS`, with 100 RHS (Right-Hand Sides), outperforms LAPACK's `DSYTRS` by more than 100%. These performance results are decidedly unfair to our new algorithms: we compare against level 3 algorithms as opposed to level 2 packed algorithms. In Sections 3, 4 and 5 we also give brief notes on how to program for these new formats. Equally brief are Sections 6 to 8 (on Vectors, Recursion, and BLAS), which describe how the new formats effect/are affected by these three subjects. Space limitations do not allow us to cover these topics in any detail. In Section 9 we briefly describe what kernel routines are for IBM platforms. In Section 10 we give a brief summary and present our conclusions.

2 Rationale and Underlying Foundations of our Approach

For a set of linear equations $Ax = b$ there are two points of view. The more popular view is to select an algorithm, say Gaussian elimination with partial pivoting, and use it to compute x. The other view, which we adopt here, is to perform a series of linear transformations on both A and b so that the problem, in the new coordinate system, becomes simpler to solve. Both points of view have their merits. We use the second as it demonstrates some reasons why the Algorithms and Architecture approach, [1], is so effective. Briefly, the Algorithms and Architecture approach states that the key to performance is to understand the algorithm and architecture interaction. Furthermore a significant improvement in performance can be obtained by matching the algorithm to the architecture and vice-versa. In any case, it is a very cost-effective way of providing a given level of performance.

The fundamental reason or idea behind the coordinate transformation approach is the concept of Equivalence and Elementary Matrices. In [2], pages 170-173, matrices and row-equivalence are discussed. In particular, elementary row operations of three types are cited on page 172. The most important is the addition of any multiple of one row to any other row of a matrix. See also Chapter 6, Elementary Operations and the concept of Equivalence, in [18], for another treatment.

Closely related to elementary operations (there are both row and column types) are elementary matrices E which are a rank one modification of the identity matrix $I : E = I + \sigma uv^T$, where u and v are vectors and σ is a scalar. We have the following:

THEOREM Let $Ax = b$ represent an m by n linear system of equations. Let T represent an elementary operation or an elementary matrix. Let $A_1x = b_1$ represent the m by n linear system of equations after applying T to both sides of $Ax = b$, i.e., $A_1 = TA$ and $b_1 = Tb$. Then the solution properties of both systems are the same.

Note that a more general form of an elementary operation can be considered a linear transformation.

COROLLARY Let $T_i, 1 \leq i \leq k$ represent k linear transformations where each T_i is elementary. Let $T = T_k ... T_1$. Then $Ax = b$ and $Cx = d$ where $C = TA$ and $d = Tb$ have the same solution properties.

As an example, we relate this second approach to the first approach of Gaussian elimination with partial pivoting, i.e., $LU = PA$. We get $C = U$ and the $k = n$ linear transformations $T_i = L_i$ or $k = \lceil n/NB \rceil$ when a blocked method is used. A second example would be $A = QR$ factorization where $C = R$ and the T_i would be elementary Householder matrices or the compact WY representation, [5, 6]. We remark that Section 2, pages 939-942 of [3] shows that product of n L_i to produce L requires *no* additional work, i.e., L is obtained via concatenation of the n L_i. This is *not* the case with elementary Householder matrices; see Section 2, pages 606-615 of [5].

Now we examine a single elementary column operation. Let the two columns be represented by vectors x and y and the scalar multiple by α. Then this operation is the level 1 BLAS `DAXPY` operation $y = y + \alpha x$. Note that `DAXPY` is a series of multiply-add operations. In fact the dot-product $x^T y$ operation, another pervasive operation, is also a series of multiply-add operations. For dense linear algebra we can conclude that multiplies and additions occur equally often and almost always in multiply-add pairs. Hence, from the architecture point of view, the use of the FMA instruction, `T=B+A*C`, is a natural choice for dense linear algebra. We mention that the FMA instruction costs slightly more than the multiply instruction and that doing a multiply and an add costs about 1.7 times more than a FMA, [19, 7, 20].

Next we claim that matrix multiplication is pervasive in the algorithms of dense linear algebra, e.g. see the book, [8], and the `LAPACK` library. Let R and S be linear transformations with a common set of basis vectors. Let $T = S(R)$ be the composition of the two linear transformations where we want T to be linear. This restriction, i.e., that T be linear, on the basis for T, in terms of the common set of basis vectors, *defines* matrix multiplication. Our dual points of view lets us describe dense linear algebra algorithms as a series of linear transformations with a common set of basis vectors. And for each such composition of transformations to be linear we *must* perform matrix multiplication. See Chapter 8, pages 209-214 of [2] for more details. We also note that matrix multiply is by *definition* a series of parallel dot-product and hence just a series of FMAs, which can be done independently.

We have just seen why matrix multiply repeatedly shows up in say, `LAPACK` algorithms. In fact, the level 3 BLAS, `DGEMM`, is called the most important level 3 BLAS. Our next point relates to the current block based (submatrix) matrix multiplication used by most `DGEMM` implementations. These implementations are optimal in the following sense:

THEOREM (Toledo, [21]) Any algorithm that computes $a_{i,k}b_{k,j}$ for all $1 \leq i, j, k \leq n$ must transfer between memory and an M word cache

$\Omega(n^3/\sqrt{M})$ words if $M < n^2/5$.

The current block based algorithms transfer $O(n^3/\sqrt{M})$ words. The point here is that we need not search for better ways to perform matrix multiplication via DGEMM implementation as our current algorithms are achieving the lower bound complexity measure. Also, practical experimental evidence, e.g. ESSL's DGEMM achieves better than 90 % of peak performance, tells us the same thing in a very concrete way.

We end this section with brief remarks about blocking. The general idea of blocking is to get information to a high speed storage and use it multiple times to amortize the cost of moving the data. In doing so, we satisfy points 1 and 3 of the Introduction. We only touch upon TLB (Translation Look-aside Buffer), cache, and register blocking. The TLB contains a finite set of pages. These pages are known as the *current working set* of the computation. If the computation addresses only memory in the TLB then there is no penalty. Otherwise, a TLB miss occurs resulting in a large performance penalty; see [1]. Cache blocking reduces traffic between the memory and cache. Analogously, register blocking reduces traffic between cache and the registers of the CPU. Cache and register blocking are further discussed in [1].

2.1 Operation Counts are a Good Measure of Kernel Routine Performance

From the 1960s to the 1980s it was common practice to use operation counts as a measure of the performance of dense linear algebra algorithms, or, more generally, floating point computations. This was true because the memory hierarchy was uniform. In fact, it was during this time that the FLOP and the MFLOP became measures of floating point computations. The MFLOP rate, still widely used today, is now mainly employed as a measure of peak performance, as the memory hierarchies are no longer uniform.

Our introduction of new data structures, along with their use by kernel routines, brings operation counts back into prominence because L1 cache can be considered a uniform memory hierarchy under the assumptions of points 1 to 6 in the Introduction. Another reason for their introduction here is to further emphasize our point that the FMA is indeed the key floating point instruction of dense linear algebra. To illustrate, we consider the three level 3 BLAS, DGEMM, DTRSM, DSYRK, as well as the level 3 Cholesky factor kernel, DPOFU. Given m by n matrix C, k by m matrix A, and k by n matrix B, DGEMM ($C = C - A^T B$) clearly requires $2mnk$ FLOPs, consisting entirely of mnk FMAs. Given m by n matrix B and m by m triangular matrix A, the FLOP count of DTRSM ($B = A^{-1}B$) is $m^2 n$. This FLOP count consists of mn reciprocal multiplies and $m(m-1)n/2$ FMAs. Given m by m triangular matrix C and n by m matrix A the FLOP count of DSYRK ($C = C - A^T A$) is $m(m+1)n$. This FLOP count consists entirely of $m(m+1)n/2$ FMAs. Clearly then, these three important level 3 BLAS consist almost entirely of FMAs. Finally, consider our level 3 Cholesky kernel routine of order n. Its FLOP count is n square roots, n divides, $n(n-1)/2$ reciprocal multiplies and $n(n^2-1)/6$ FMAs. Again, Cholesky factorization consists mainly of FMAs as our transformational point of view suggests. Now, for clarity and convenience, let $N = n\mathtt{NB}$. Using point 4 of the Introduction, Cholesky factorization consist of n calls to DPOFU, $n(n-1)/2$ calls each to DTRSM and DSYRK, and $n(n-1)(n-2)/6$ calls to DGEMM. Using our new data structures, each of these kernel routines will execute at a near uniform rate because all of their matrix operands map well into L1 cache (see points 1 to 6 of the Introduction). Thus, the N by N Cholesky problem, occupying space in a non-uniform deep memory hierarchy can be effectively mapped onto a set of submatrices and then executed as a series of $n + n(n-1) + n(n-1)(n-2)/6 = n(n+1)(n+2)/6$ kernel routine calls where each kernel routine call will execute at a near uniform rate in L1 cache.

2.2 Description of Four Kernel Routines

In Section 2.1, we gave operation counts for DGEMM, DTRSM, DSYRK and Cholesky factorization. Let A be a general matrix of order N with a leading dimension $\mathtt{LDA} \geq N$, i.e., $A = \mathtt{A}(0:\mathtt{LDA}-1, 0:\mathtt{N}-1)$. I use colon notation, see [8]. Note that in Fortran $a_{i,j}$ is stored in location $i + \mathtt{LDA}j$ of the array A holding matrix A. We now describe calling sequences of four kernel routines that can be used as building blocks for the above four routines. We will refer to general matrices A, B, C, U stored in arrays A,B,C,U with associated leading dimensions LDA,LDB,LDC,LDU.

DATB4(M,N,K,A,LDA,B,LDB,C,LDC) computes $C = C - A^T B$ where A is k by m, B

```
     1 13 25 37 49 61 73 85  97 109
     2 14 26 38 50 62 74 86  98 110
     3 15 27 39 51 63 75 87  99 111
     4 16 28 40 52 64 76 88 100 112
     5 17 29 41 53 65 77 89 101 113
A=   6 18 30 42 54 66 78 90 102 114
     7 19 31 43 55 67 79 91 103 115
     8 20 32 44 56 68 80 92 104 116
     9 21 33 45 57 69 81 93 105 117
    10 22 34 46 58 70 82 94 106 118
    11 23 35 47 59 71 83 95 107 119
     *  *  *  *  *  *  *  *   *   *
```

Figure 1: Standard Fortran column-major storage order

```
 1  5  9 13|49 53 57 61| 97 101**
 2  6 10 14|50 54 58 62| 98 102**
 3  7 11 15|51 55 59 63| 99 103**
 4  8 12 16|52 56 60 64|100 104**
-----------|-----------|---------
17 21 25 29|65 69 73 77|113 117**
18 22 26 30|66 70 74 78|114 118**
19 23 27 31|67 71 75 79|115 119**
20 24 28 32|68 72 76 80|116 120**
-----------|-----------|---------
33 37 41 45|81 85 89 93|129 133**
34 38 42 46|82 86 90 94|130 134**
35 39 43 47|83 87 91 95|131 135**
 *  *  *  *| *  *  *  *| *    * **
```

Figure 2: New Square blocked full column-major order

is k by n and C is m by n.

`DSLVL4(B,LDB,M,U,LDU,N)` computes $B = BU^{-T}$ where U is order n upper triangular and B is m by n.

`DTATA4(M,N,A,LDA,U,LDU)` computes $U = U - A^T A$ where U is order m symmetric stored in full upper format and A is n by m.

`DPOFU4(A,LDA,N,INFO)` computes $U^T U = A$ where A is order n symmetric positive definite stored in full upper format and $U \leftarrow A$.

3 Square Blocked Full and Hybrid Formats for General Matrices

These new data formats are best described by an example. Let A be a `M` = 11 by `N` = 10 matrix with `LDA` = 12. In Fortran this matrix would be stored as shown in Figure 1; the number in location (i, j) is $a(i, j)$'s storage position in A.

Let `NB` = 4 be the block size and suppose A is stored in column-major block order. Here $m1$ = $n1$ = 3 and A is a 3 by 3 block matrix. Each square block is a `NB` by `NB` and contains a submatrix of A. In the new data format A would be stored as in Figure 2.

Now suppose a user inputs his matrix A in standard Fortran or C order with some additional space in the array holding A directly below A. For example, in Figure 1 the minimum storage for A is `LDA`*n = 120 double words. If the user supplied ns $\geq$ 144 elements (extra storage for $\geq \mathtt{NB}^2 * \mathtt{m1} * \mathtt{n1} - \mathtt{LDA} * \mathtt{n} = 24$ double words directly below A), one could transform Fortran storage order to the new square blocked, full column order. Once this data transformation is completed, one could execute the block equivalent of a standard vanilla code with calls made to standard kernel routines.

3.1 Square Blocked Full Data Format Gaussian Elimination

We briefly describe this procedure for a right-looking Gaussian elimination with partial pivoting. In the vanilla version (`NB` = 1 here), the outer loop is on $j = 0, N - 1$ and for each j one finds the pivot in column j and swaps it with `a(j,j)`. Then `a(j+1:M-1,j)` is scaled by the reciprocal of the pivot to form column j of L. Next, columns $k = j + 1, N - 1$ are processed in two steps. Let k be a generic column. First, a swap of row j and the pivot row is made. Second, a `DAXPY` update is performed : `a(j+1:M-1,k)=a(j+1:M-1,k)-a(j+1:M-1,j)*a(j,k)`. For the block version refer to Figure 4. In the blocked version the outer loop is on $bj = 0, \mathtt{n1} - 1$ and for each bj one factors a block column L*U = P*A(`j:M-1,j:N-1`) by calling kernel routine `RGETF3`. Then columns $j + nb$ to $N - 1$ are processed in three block steps of size `ks`. Let `k:k+ks-1` be the generic block column. First, there is a forward pivot step. This is accomplished by calling a blocked version of the LAPACK routine, `DLASWP`, called `DLASWPB`. Next, there is a `DTRSM` computation whose first four parameters are 'L', 'L', 'N', 'U', done by the kernel routine `DLLNU4`. Finally, there is a `DGEMM` update whose first two parameters are 'N', 'N' which is done by a series of calls to kernel routine `DAB4`. After completion of the k block loop there is a back pivot step. As just mentioned, there are three kernel routines in the block equivalent (see Figure 4). They are a factor a panel of size m by n where $n \leq$ NB kernel called RGETF3, a DTRSM kernel called DLLNU4, and a DGEMM kernel called DAB4. We mention that the factor kernel has the same functionality as LAPACK's DGETF2 routine. However, RGETF3 is a level 3 routine, done recursively, as the suffix 3 and the prefix R, indicate. Note that the vanilla routine, actually the LINPACK routine DGEFA, does not have a back pivot step. So the blocked version does extra work, some of which could be avoided.

Now I turn to how DGEFB might be packaged as a subroutine in standard Fortran. Following LAPACK conventions, we suggest using the input format of DGETRF(M,N,A,LDA,IPIV,INFO). The new routine would have a nearly identical calling sequence: DBGTRF(M,N,A,LDA,IPIV,NSINFO). The new input parameter is NS $\geq$ n*LDA and it is combined with LAPACK's output *only* parameter, INFO; hence, the name NSINFO. If NSINFO is not sufficiently large, DBGTRF just returns by placing in -NSINFO the amount of storage necessary to apply the new block algorithm. If NSINFO is sufficiently large, the input matrix is rearranged into square block format and then DBGTRF is executed using the new block algorithm. Like the LAPACK LWORK parameter, the value returned in -NSINFO will be the value used by DBGTRF for good level 3 performance, namely m1*n1*NB^2.

3.2 Square Blocked Full Hybrid Data Formats for Matrices

Let A be m by n, where $\texttt{LDA} \geq m$; i.e., A is stored in column-major order. Assume $m \geq n$ (a similar result holds for $n > m$). Let $n1 = \lceil n/\texttt{NB} \rceil$ and $n2 = n + \texttt{NB} - n1 * \texttt{NB}$. Partition the column space of A into $n1 - 1$ pieces of size NB and a leftover piece of size $n2 \leq \texttt{NB}$. This new format represents A as a set of $n1 - 1$ rectangles of size $m * \texttt{NB}$ and a last one of size $m * n2$. We partition the row space of A into $m1 - 1$ pieces of size NB and a leftover piece of size $m2 \leq \texttt{NB}$. Here $m1 = \lceil m/\texttt{NB} \rceil$ and $m2 = m + \texttt{NB} - m1 * \texttt{NB}$. Matrix A is a $m1$ by $n1$ block matrix. Each block has a TRANS parameter; i.e., it can be stored in column- or row-major order. In each of the $n1$ rectangles, the last $\texttt{LDA} - m$ rows are stored last. For the original A we assume $\texttt{LDA} \approx m$. In Figure 3 we depict the matrix A that is associated with Figure 1.

```
    1  5  9 13|49 53 57 61|  97 101
    2  6 10 14|50 54 58 62|  98 102
    3  7 11 15|51 55 59 63|  99 103
    4  8 12 16|52 56 60 64|100 104
  -----------|-----------|-------
   17 21 25 29|65 69 73 77|105 109
 A=18 22 26 30|66 70 74 78|106 110
   19 23 27 31|67 71 75 79|107 111
   20 24 28 32|68 72 76 80|108 112
  -----------|-----------|-------
   33 36 39 42|81 84 87 90|113 116
   34 37 40 43|82 85 88 91|114 117
   35 38 41 44|83 86 89 92|115 118
    *  *  *  *| *  *  *  *|  *   *
```

Figure 3: Full Hybrid Block column-major storage order

When $\texttt{LDA} \gg m$, our algorithms will suffer. This is because the useful information in the array A, representing matrix A, is mn while the unused (unnecessary) information is $(\texttt{LDA} - m)n$. Performing the in-place data transformation requires $O(\texttt{LDA}n)$ operations, so the ratio of overhead to useful work is $\texttt{LDA}/m - 1$. For example, if we set LDA = 1100 in Figure 3 this ratio is 99, i.e., there are 99 wasted operations for every useful one. The main advantage of using square blocked full hybrid data format is that it makes those dense linear algorithms which use it completely portable. Further, if $\texttt{LDA} \gg m$, we can still apply the original algorithm. Finally, we mention that a buffer of size $M \times \texttt{NB}$ is required to do the in-place data transformation. This is accomplished by allocating the buffer at the start of processing and freeing it at the end. This is certainly allowable as many LAPACK and BLAS algorithms routinely use extra space of this size.

3.2.1 A Matching BLAS 3 LU = PA Algorithm

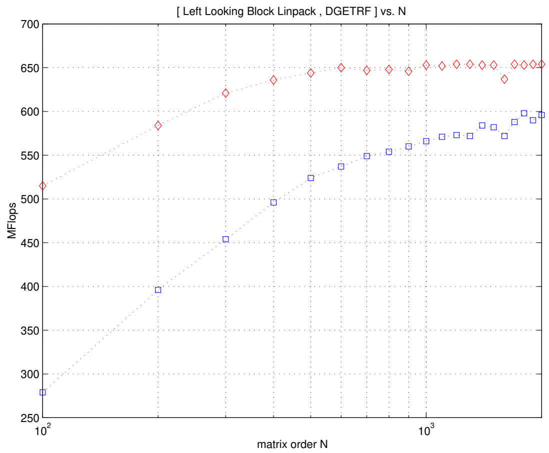
It is almost as easy to code this algorithm as the DGEFB algorithm of Figure 4. However, due to lack of space, we do not include the code for this format.

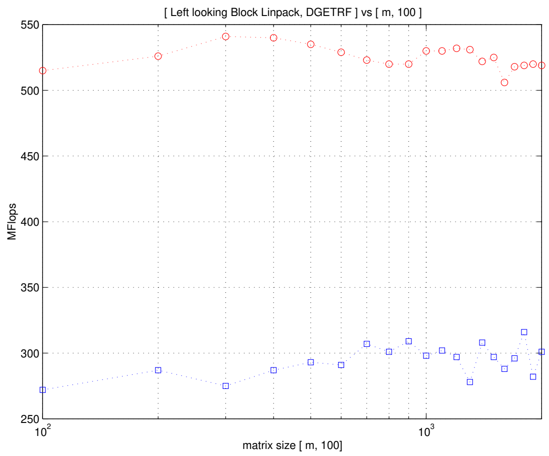
3.3 Programming Notes for Square Blocked Full Formats

As mentioned in the introduction, one addresses the block coordinates (I, J) and the local coordinates (i, j) within that block. So, by using three- or four-dimensional Fortran/C arrays, one can program for these formats. In Figure 4, we use a three-dimensional array; we handle the block (I, J) addressing implicitly, as 1-D addressing in the third dimension. See Section 7 (below) for a discussion regarding the handling of blocks that are stored recursively. The main programming difficulties arise in coding the factor kernel routines. For example, Gaussian elimination with partial pivoting requires working with a column of blocks. Thus, local offsets in a square block are required; see Section 6. Because of this, the number of parameters, and generalization to algorithms for distributed memory processors, I predict that descriptors will eventually be used. In [10], I described some possible descriptor formats.

3.4 Performance for LU = PA

We have tried several variants of solving LU = PA; e.g., left- and right-looking and recursive variants. Also, we have experimented with several variants of the new data structures. Here we show performance results on a 200 MHz IBM Power 3 processor with a peak computational rate of 800 MFLOPS. Results are for Full Hybrid Block format. We do not do the data transformation immediately. This is because Fortran storage (column-major order) is ideally suited for the factor part of the algorithm. After factorization it pays to do a data transformation. Not included here are the results of algorithm DGEFB. We remark that these results are similar. There are two plots. Both compare the block LINPACK algorithm with the LAPACK algorithm DGETRF. Note that the x-axis is log scale; we let n and m of $(m, 100)$ range from 100 to 2000. For square matrices the new code is 90 % to 10 % faster than DGETRF as n ranges from 100 to 2000. Note that, for very large n, these codes are dominated by ESSL's DGEMM code. The second graph is for tall thin matrices; i.e., n is held fixed at size 100. Here, the new code is nearly twice as fast as DGETRF.





4 Square Blocked Packed Formats for Symmetric/Triangular Arrays

These new formats are a generalization of packed format for triangular arrays. They are also a generalization of full format for triangular arrays. The main benefit of the new formats is that they allow for level 3 performance while using about half the

```
      subroutine dgefb (m,n,a,nb ,ipiv,info)
      implicit none
      integer*4 m,n,nb ,info
      real*8 a(0:nb-1,0:nb-1,0:*)
      integer*4 ipiv(0:1,0:*)
      real*8 one,zero,t
      parameter (one=1.d0,zero=0.d0)
      integer*4 nb2,m1,n1,m2,n2,j,k,i,bj,bk,bi,ms,ks,ns
      integer*4 iti,ibi,iai,ici ! pointers
      info=0
      nb2=nb*nb ! size of a square block
      m1=(m+nb-1)/nb ! row order of block matrix
      n1=(n+nb-1)/nb ! col order of block matrix
      m2=m+nb-m1*nb ! row size of last block
      n2=n+nb-n1*nb ! col size of last block
      bj=0 ! block j index
      do j=0,n-1,nb
*
*       factor column swath A(j:j:m-1,j:j+nb-1)
*
        bk=bj
        iti=bj+m1*bj ! (0,0)-th element of block iti is A(j,j)
        ns=nb
        if(bj.eq.n1-1)ns=n2
        call rgetf3(m-j,ns,a(0,0,iti),nb,ipiv(0,j),info)
*       globalize pivot indices
        do i=j,j+ns-1
          ipiv(1,i)=ipiv(1,i)+bj
        enddo
        do k=j+nb,n-1,nb
          bk=bk+1 ! block k index
          ks=nb
          if(bk.eq.n1-1)ks=n2
          ibi=bj+m1*bk ! (0,0)-th element of block ibi is A(j,k)
*         forward pivot A(j:j+m-1,k:k+ks-1) ! a(0,0,bk*m1) -> A(0,k)
          call dlaswpb(ks,a(0,0,bk*m1),nb,j,j+ns-1,ipiv,1)
*         solve L*X = A(j:j+nb-1,k:k+ks-1)
          call dllnu4(nb,ks,a(0,0,iti),nb,a(0,0,ibi),nb)
          bi=bj
          do i=j+nb,m-1,nb
            bi=bi+1
            ms=nb
            if(bi.eq.m1-1)ms=m2
            iai=bi+m1*bj ! (0,0)-th element of block iai is A(i,j)
            ici=bi+m1*bk ! (0,0)-th element of block ici is A(i,k)
*
*           update A(i:i+ms-1,k:k+ks-1) = A(i:i+ms-1,k:k+ks-1)
*                  - A(i:i+ms-1,j:j+nb-1)*A(j:j+nb-1,k:k+ks-1)
*
            call dab4(ms,ks,nb,a(0,0,iai),nb,a(0,0,ibi),nb,
     &                a(0,0,ici),nb)
          enddo
        enddo
        bi=0
        do i=0,bj-1
*         back pivot A(j:j+m-1,i:i+nb-1) ! a(0,0,bi) -> A(0,i*nb)
          call dlaswpb(nb,a(0,0,bi),nb,j,j+ns-1,ipiv,1)
          bi=bi+m1
        enddo
        bj=bj+1 ! next block j col
      enddo
      return
      end
```

Figure 4: `DGEFB` subroutine.

```
       Packed Lower
 1
 2 11
 3 12 20
 4 13 21 28
 5 14 22 29 35
 6 15 23 30 36 41
 7 16 24 31 37 42 46
 8 17 25 32 38 43 47 50
 9 18 26 33 39 44 48 51 53
10 19 27 34 40 45 49 52 54 55

       Packed Upper

 1  2  4  7 11 16 22 29 37 46
    3  5  8 12 17 23 30 38 47
       6  9 13 18 24 31 39 48
         10 14 19 25 32 40 49
            15 20 26 33 41 50
               21 27 34 42 51
                  28 35 43 52
                     36 44 53
                        45 54
                           55
```

Figure 5: Packed Format Arrays

storage of the full array cases. In packed format, the elements of a triangular matrix would be stored as shown in Figure 5; the number in location (i, j) is $a(i, j)$'s storage position in A.

For square blocked packed formats there are two parameters, `NB` and `TRANS`, where, typically, $n \geq$ `NB`. For this format, we first choose a block size, `NB`, and then we lay out the data in squares of size `NB`. Each square block can be in row-major order (`TRANS` = 'T') or column-major order (`TRANS` = 'N'). This format supports both `uplo` = 'U' or 'L'. For `uplo` = 'L', the first vertical stripe is n by `NB` and it consists of $n1$ square blocks where $n1 = \lceil n/\mathtt{NB} \rceil$. It holds the first trapezoidal n by `NB` part of L. The next stripe has $n1-1$ square blocks and it holds the next trapezoidal n - `NB` by `NB` part of L, and so on, until the last stripe consisting of the last leftover triangle is reached. There are $n1(n1+1)/2$ square blocks in all.

An example of Square Blocked Lower Packed Format (SBLP) with TRANS = 'T' is given in Figure 6. Here n = 10, `NB` = 4 and TRANS = 'T' and the numbers represent the position within the array where $a(i, j)$ is stored. Note the missing numbers (e.g., 2, 3, 4, 7, 8, and 12) which correspond to the upper right corner of the first stripe. This square

```
 1  *  *  *
 5  6  *  *
 9 10 11  *
13 14 15 16
-----------|
17 18 19 20|49  *  *  *
21 22 23 24|53 54  *  *
25 26 27 28|57 58 59  *
29 30 31 32|61 62 63 64
-----------|-----------|
33 34 35 36|65 66 67 68|81  * * *
37 38 39 40|69 70 71 72|85 86 * *
 *  *  *  *| *  *  *  *| *  * * *
 *  *  *  *| *  *  *  *| *  * * *
```

Figure 6: Square Blocked Lower Packed Format

blocked, lower, packed array consists of 6 square block arrays. The first three blocks hold submatrices that are 4 by 4, 4 by 4, and 2 by 4. The next two blocks hold submatrices that are 4 by 4 and 2 by 4. The last square block holds a 2 by 2 submatrix. Note the padding, which is done for ease of addressing. Addressing this set of six square blocks as a composite block array is straightforward.

An example of Square Blocked Upper Packed Format (`TRANS = 'N'`) is given in Figure 7. The blocked, upper, packed array consists of 6 square block arrays. The first block holds a 4 by 4 submatrix. The next two block hold 4 by 4 block submatrices. The last three blocks hold 4 by 2, 4 by 2, and 2 by 2 submatrices. Each block is in column-major order. Note the padding, which is done for ease of addressing. Addressing this set of six square blocks as a composite block array is straightforward.

Here is another important point. With extra storage appended directly below a standard packed array, denote here by `AP`, one can move to these new data formats without allocating additional storage. For the examples above, `AP` requires 55 storage elements. If there are 96 - 55 = 41 free locations below `AP`, then one can move the packed array downward into the block packed array by starting at the end of `AP` and moving the square blocks in a block column into a buffer of of size $n1$*$\mathtt{NB}^2$ either in row-major or column order. The entire buffer can then be copied back over the vacated block column.

I believe a main innovation in using the square blocked packed format is to see that one can translate, verbatim, a standard packed factorization algorithm into a square blocked packed algorithm

```
1 5   9 13|17 21 25 29|49 53 * *
* 6  10 14|18 22 26 30|50 54 * *
* *  11 15|19 23 27 31|51 55 * *
* *   * 16|20 24 28 32|52 56 * *
          |-----------|---------
           33 37 41 45|65 69 * *
            * 38 42 46|66 70 * *
            *  * 43 47|67 71 * *
            *  *  * 48|68 72 * *
                      |---------
                       81 85 * *
                        * 86 * *
                        *  * * *
                        *  * * *
```

Figure 7: Square Blocked Upper Packed Format

by replacing each reference to an i, j element by a reference to its corresponding square block submatrix. This is an application of point 4 in the Introduction. Because of the storage layout, the beginning of each block is easily located. Also key is that this format supports level 3 BLAS. Hence, old, packed code is easily converted into square blocked, packed, level 3 code. In a nutshell, I am keeping "standard packed" addressing so the library writer/user can handle his own addressing in a Fortran/C environment.

Now we turn to full format storage. We continue the example with `N` = 10, and `LDA` = 12. Simply set `NB` = `LDA` = 12 and one obtains full format; i.e., square block packed format gives a single block triangle which happens to be full format (see Figure 8).

In Figure 8 we ignore the last $\mathtt{NB} - n$ columns of the square blocked array. Here is an interesting observation. The unused storage, of size $n*(\mathtt{LDA}+\mathtt{LDA}-n-1)/2$, consists of n fragmented vectors of sizes $\mathtt{LDA} - n : \mathtt{LDA} - 1 : 1$. As before, I use colon notation, see [8]. These vectors are interspersed with $1 : n : 1$ vectors of the symmetric matrix A. For `uplo` = 'U' the symmetric matrix consists of ten vectors of sizes 1 to 10 in steps of 1 (55 elements total). The unused storage consists of ten vectors of sizes 11 to 2 in steps of -1 (65 elements total). It is my opinion that users of dense linear algebra codes do *not* utilize this fragmented storage. If this is so, one could convert full format to square blocked packed format, thereby freeing up a contiguous block of storage.

4.1 A Cholesky Algorithm for Square Block Packed Format

Now I turn to programming dense linear algorithms in the new formats. As an example, Figure 9 gives `uplo` = 'L' code for `DPSTRF` (`PS` stands for positive definite symmetric), which produces the lower Cholesky factor for the positive definite symmetric A, where A is in square blocked, packed, lower, transposed format. Algorithm `DPSTRF` is a simple right-looking algorithm, as the code illustrates. One could let `NB` = 1 and then each level 3 square block kernel call becomes a corresponding scalar operation on an i, j element. For `NB` = 1, this routine is a variant of the LINPACK routine, `DPPFA`. Routine `DPOFU4` is a ESSL factor kernel. The corresponding scalar operation is square root. In ESSL, [15], all level 3 BLAS and factorization routines use kernel routines. For example, in ESSL's `DGEMM`, a blocking routine is called to partition the matrix operands, A and B, into submatrices (matrix blocks) and then calls are made to kernel routines that operate on the blocks. Data copying of the operands to the kernel routines is decided on by the ESSL `DGEMM` blocking routine. This data copying is done so that the kernel routines uniformly perform at a high rate of execution. (see point 2a of the Introduction). We remark that data copying is usually done by ESSL's `DGEMM` routine; see page 739 of [9] for more details. Now in `DPSTRF` we can call the kernel routine, `DATB4`, directly, thereby avoiding data copying, since `NB` was selected for good L1 cache behavior. The routine `DSLVL4` is a `DTRSM` kernel routine and routine `DTATA4` is a `DSYRK` kernel routine. The suffix, `4`, on each kernel routine indicates that 4 by 4 register blocking (loop unrolling by four) is being used. These kernels have no clean-up code. So, when the order of A is not a multiple of 4, we pad the leftover blocks (up to 3 rows and columns with zeroes and up to 3 diagonals with ones). Thus the code works of any matrix order, and the kernel routines are short and of very high performance. We mention that the kernel routines are programmed in Fortran. Note that routine `DPSTRF` is just one example of the general schema presented in Points 7 and 8 of the Introduction. In graph 1 of Section 4.4 we give the performance of `DPSTRF` versus LAPACK `DPOTRF`.

Before closing I want to suggest a way to package this data storage in LAPACK. I continue with the current routine for Cholesky factorization, `uplo` = 'L'. For `uplo` = 'U' a similar proce-

```
          uplo = 'L'                              uplo = 'U'

  1  *  *  *  *  *  *  *  *   *         1 13 25 37 49 61 73 85  97 109
  2 14  *  *  *  *  *  *  *   *         * 14 26 38 50 62 74 86  98 110
  3 15 27  *  *  *  *  *  *   *         *  * 27 39 51 63 75 87  99 111
  4 16 28 40  *  *  *  *  *   *         *  *  * 40 52 64 76 88 100 112
  5 17 29 41 53  *  *  *  *   *         *  *  *  * 53 65 77 89 101 113
  6 18 30 42 54 66  *  *  *   *         *  *  *  *  * 66 78 90 102 114
  7 19 31 43 55 67 79  *  *   *         *  *  *  *  *  * 79 91 103 115
  8 20 32 44 56 68 80 92  *   *         *  *  *  *  *  *  * 92 104 116
  9 21 33 45 57 69 81 93 105  *         *  *  *  *  *  *  *  * 105 117
 10 22 34 46 58 70 82 94 106 118        *  *  *  *  *  *  *  *  *  118
  *  *  *  *  *  *  *  *  *   *         *  *  *  *  *  *  *  *  *   *
  *  *  *  *  *  *  *  *  *   *         *  *  *  *  *  *  *  *  *   *
```

Figure 8: Square blocked packed formats when $\mathtt{NB} = n$

dure would be followed. Define a new LAPACK routine, called subroutine `DPSTRF( UPLO, N, AP, NSINFO )`. Input parameters `UPLO, N, AP` have the same meaning as the corresponding parameters of LAPACK routine `DPPTRF` and, hence, do not need any description. The new input parameter, `NS`, stands for the storage the user inputs for `AP`; $\mathtt{NS} \geq n(n+1)/2$. Hence, the name, `NSINFO`. On output, `NSINFO` plays the role of LAPACK output *only* parameter `INFO`. If `NS` is not sufficiently large, `DPSTRF` just returns in −`NSINFO` the amount of storage necessary for good level 3 performance. Like the `LWORK` parameter, the value returned in `-NSINFO` will be the value used by `DPSTRF` for good level 3 performance. If `NS` is sufficiently large, the input storage is rearranged into square block format and then `DPSTRF` is executed giving level three performance.

4.1.1 An Example that Describes Algorithm `DPSTRF`

Let `NB = 88`. This is good block size for the IBM Power 3 L1 cache which holds 8192 double precision words. The `DGEMM` kernel, `DATB4`, whose functionality was described in Section 2.2 places all of A, four columns of B and a 4 by 4 submatrix of C into L1 cache during a computing instance. This requires 7744 + 352 + 16 = 8112 double words. On IBM RS/6000 platforms, only the A matrix fully occupies L1 cache. Additionally, four columns of the B matrix and four lines of the C matrix are required. We do *not* require that all three matrices, A, B, and C, occupy L1 cache. Thus, $mk + 4k + 4LS$ should be less than some large fraction of the size of L1. Here LS stands for line size which is 16 double words for Power 3. We note that ATLAS, see page 18 of [22], uses a very similar L1 cache blocking strategy.

Let N = 250. Routine `DPSTRF` requires N to be a multiple of four. Thus, N will become 252 by padding two extra rows and columns of an identity matrix to A. A conversion routine, called `DLPTUP` standing for Lower Packed To Upper (square block) Padded converts an lower packed format array to square blocked lower format with padding of up to 3 extra rows and columns to make N a multiple of four. The size of the block array will be three so A will look like Figure 6 where now `NB=88` and `n2=76`. Note rows 75 and 76 of blocks 3 and 5 will be padded with zeroes and also rows 77 to 88 of these blocks will consist of unused space. Block 6 will contain an lower triangular matrix stored by row of order 76 where the last two rows are rows of an identity matrix.

Note that we store our submatrices in SBLP format with `TRANS` parameter = 'T'. This is because we emulate factoring A in full upper format as we want all our calls in the four kernel routines to do their dot products stride one. A proof that this is okay is alluded to on pages 626 and 628 of [13]. Now we are ready to follow algorithm, `DPSTRF`. We know that we will factor a 3 by 3 block matrix consisting of six submatrices each residing in a subarray of size $\mathtt{NB2} = \mathtt{NB}^2$. These subarrays reside in memory locations $1 + \mathtt{NB}^2\mathtt{i}$, $0 \leq i \leq 6$ = 1, 7775, 15489, 23233, 30977, 38721. Again see Figure 6 and adjust the starting positions of the blocks by 88*88 = 7744 instead of 4*4 = 16. In what follows we will identify each of the six submatrices by their starting position of the global array that holds them. Now, according to Section

```
!       Square Blocked Lower Packed Transposed Format Cholesky Factor     !
      subroutine dpstrf(uplo,n,nb,a,info)
      implicit none
      character*1 uplo ! only uplo = 'L' and trans = 'T' is handled
      integer*4 n,nb,info ! mod(n,4) = 0 is assumed
      real*8 one,a(*)
      integer*4 j,k,i,kb,ib,jj,jk,kk,ji,ki,nb2,n1,n2,nn
      parameter (one=1.d0)
      info=0
      nb2=nb*nb ! size of a square block
      n2=(n+nb-1)/nb ! order of block matrix
      n1=n2*nb2 ! block lda
      jj=1 ! -> a(j,j), j=1
      do j=1,n-nb,nb
*       factor a(j:j+nb-1,j:j+nb-1) = u**T*u, u = a.
        call dpofu4(a(jj),nb,nb,info) ! factor kernel
        if(info.gt.0)goto 30
        ji=jj
        do i=j+nb,n,nb
          ji=ji+nb2
          ib=min(n-i+1,nb)
*         solve a(j:j+nb-1,j:j+nb-1)**T*u(j:j+nb-1,i:i:ib-1) =
*               a(j:j+nb-1,i:i+ib-1) for u, u = a.
          call dslvl4(a(ji),nb,ib,a(jj),nb,nb) ! trsm kernel
        enddo
*       initialize pointers for the k loop
        kk=jj ! -> a(k,k), k=j
        nn=n1     ! lda of k block, k=j
        jk=jj ! -> a(j,k), k=j
        do k=j+nb,n,nb
*         update pointers for the k loop
          jk=jk+nb2! -> a(j,k)
          kk=kk+nn      ! -> a(k,k)
          kb=min(n-k+1,nb)
*         update a(k:k+kb-1,k:k+kb-1) = a(k:k+kb-1,k:k+kb-1)
*         - a(j:j+nb-1,k:k+kb-1)**T*a(j:j+nb-1,k:k+kb-1)
          call dtata4(kb,nb,a(jk),nb,a(kk),nb) ! syrk kernel
          ji=jk ! -> a(j,i) i=k
          ki=kk ! -> a(k,i) i=k
          do i=k+nb,n,nb
            ji=ji+nb2 ! -> a(j,i)
            ki=ki+nb2 ! -> a(k,i)
            ib=min(n-i+1,nb) ! block a(k,i) has size nb by ib
*           update a(k:k+kb-1,i:i+ib-1) = a(k:k+kb-1,i:i+ib-1)
*           - a(j:j+nb-1,k:k+kb-1)**T*a(j:j+nb-1,i:i+ib-1)
            call datb4(kb,ib,nb,a(jk),nb,a(ji),nb,a(ki),nb) ! gemm kernel
          enddo
          nn=nn-nb2 ! lda for next time through loop
        enddo
*       update pointers for the j loop
        jj=jj+n1      ! -> a(j,j), j=j+nb
        n1=n1-nb2    ! lda for next j block
      enddo
*     factor a(j:n,j:n)
      call dpofu4(a(jj),nb,n-j+1,info) ! factor kernel
      if(info.gt.0)goto 30
      return
   30 info=info+j-1
      return
      end
```

Figure 9: DPSTRF subroutine.

2.1 there will be three calls to DPOFU4, three calls each to DSLVL4, DTATA4 and a single call to DATB4. Algorithm DPSTRF is a right-looking algorithm, [4]. The outer block j loop factors and scales a trapezoidal panel and then uses the factored block column to update the remaining block columns to its right. There will be three passes through the outer block j loop. It suffices to give the calling parameters of the ten kernel routine calls. Knowing their values makes it easy to follow the program flow of the example. For block pass one (j = 1, jj = 1) there is one factor call, two scale calls and three update calls:

```
  CALL DPOFU4(A(1),NB,NB,INFO)
  CALL DSLVL4(A(7775),NB,NB,
A(1),NB,NB)
  CALL DSLVL4(A(15489),NB,76,
A(1),NB,NB)
  CALL DTATA4(NB,NB,A(7775),NB,
A(23233),NB)
  CALL DATB4(NB,76,NB,A(7775),NB,
A(15489),NB,A(30977),NB)
  CALL DTATA4(76,NB,A(15489),NB,
A(38721),NB)
```

For block pass two (j = 89, jj = 23233) there is one factor call, one scale call and a single update call:

```
  CALL DPOFU4(A(23233),NB,NB,INFO)
  CALL DSLVL4(A(30977),NB,76,
A(23233),NB,NB)
  CALL DTATA4(76,NB,A(30977),NB,
A(38721),NB)
```

For block pass three (j = 177, jj= 38721) there is a single factor call:

```
  CALL DPOFU4(A(38721),NB,76,INFO)
```

Note that in the above ten calls all LDAs were NB; so we were obeying point 2a of the Introduction.

4.1.2 Performance and Overhead of the Example

The purpose of this section is to quantify how Algorithm DPSTRF spends its execution time. We shall see an experimental verification of Section 2.1 and also see very small overhead. We first state the performance in MFLOPS of the ten subroutine calls in Section 4.1.1 in the order of the calls. They are 609, 715 711, 749, 775, 750, 609, 711, 750, 582. The peak MFLOP rate is 800 and all measurements made here were via the wall clock. The DGEMM and DSYRK rates of 775 and 750 are very good. The DTRSM rate of 711 to 715 would have been higher because each multiply is really an FMA. The peak rate is 4 times the cycle time; there are two FPUs and each cycle produces 2 FLOPs from each FPU. The DPOFU rates of 609 and 582 are also lower than they should be. Again, there are multiplies done by the FMA unit and each multiply only gives one, not two, FLOPs. Also, DPOFU performs divides and square roots. On the Power 3 these instructions consume 18 and 27 cycles, respectively, for a total of 45 cycles. In 45 cycles, the same chip can perform 90 FMAs or 180 FLOPs. Yet, we count each divide and square root as only one FLOP. The ten subroutine calls consume 231,132, 681,472, 588,544, 689,216, 1,117,088, 514,976, 231,132, 588,544, 514,976, 149,302 for a total of 5,366,382 FLOPs. See Section 2.1 where the various FLOP count formulas are given. The FLOP count for $N = 250$ is 5,239,875. The execution time of DPSTRF was 0.007518 seconds. The MFLOP rate is 697. However, we really did a $N = 252$ size problem. The MFLOP rate for that problem is 714. Now we compute the MFLOP rate assuming no overhead for DPSTRF. We take the dot product of the two size ten vectors to get 3,847,004,572. We then divide by 5,366,382 FLOPs to get the no overhead MFLOP rate, which is 716.87. Hence, execution time at this rate is 0.007486 seconds. Thus, the difference is 0.000032 seconds and the overhead is therefore 32/7518 or 0.43%.

4.2 Block Hybrid Formats of Symmetric/Triangular Arrays

This new format is a combination of traditional packed and full arrays. It retains the main benefit of the new formats: it allows for level 3 performance while using exactly the same storage as the packed routines do. Thus, no extra storage is required and so it is possible to obtain *full portability* with the existing packed routines. Its parameters are NB and TRANS. In Block Hybrid Format (BHF), we first choose a block size NB and then we lay out the data in trapezoidal swaths. For both uplo ='U' or 'L' there is also parameter TRANS. Now, for an example, see Figure 10 where $n = 10$, NB=4, uplo = 'L' and TRANS ='T'. In general, the first trapezoidal swath has base $n2$ and sides n and $n - n2 + 1$. It consists of a packed triangle of size $n2$ and a rectangle consisting of $n1 - 1$ blocks where $n1 = \lceil n/\mathtt{NB} \rceil$ and $n2 = n + \mathtt{NB} - n1 * \mathtt{NB}$. The remaining $n1 - 1$ trapezoidal swaths have full base width of size NB. Now each base h by sides $(b, b - h + 1)$ trapezoid consists of a packed triangle of size $nt = h(h + 1)/2$

```
 1
 2  3
-----|
 4  5|20
 6  7|21 22
 8  9|23 23 25
10 11|26 27 28 29
-----|-----------|
12 13|30 31 32 33|46
14 15|34 35 36 37|47 48
16 17|38 39 40 41|49 50 51
18 19|42 43 44 45|52 53 54 55
```

Figure 10: Blocked Hybrid Lower Packed Format

and an appended rectangle of size $b-h$ by h. The trapezoid contains $ntr = h(2b-h+1)/2$ points, the rectangle $nr = h(b-h)$ points, and the triangle nt points. Since $ntr = nr + nt$ no extra storage is required to store a trapezoid as a packed triangle and an appended rectangle. Each triangle and rectangle can be stored either in row- or column-major order (`TRANS` = 'N' or 'T'). Note that the `LDA` of the rectangles (set of "squares") will be either `NB` or $n2$. There appear to be four packed triangles because we have four cases ('L', 'N'), ('L','T'), ('U','N'), ('U','T'). However, the layouts for packed ('L','N') and ('U','T') formats are identical as are the layouts for packed ('U','N') and ('L','T') formats. In the former case we have traditional lower packed format; in the latter traditional upper packed format. Now turn again to Figure 10. There are three trapezoidal swaths. In the first swath of size $n2 = 2$ there is an upper packed triangle of order 2 and a rectangle consisting of two "squares" each of size 4 by 2 stored rowwise (TRANS ='T'). The remaining two trapezoidal swaths, two and three, are each trapezoidal swaths of size `NB=4`. The second trapezoid consists of an upper packed triangle of order 4 and a rectangle consisting of a single square of size four. The last trapezoid consists of upper packed triangle of order 4 and a rectangle consisting of no squares.

4.2.1 A Matching BLAS 3 Cholesky Algorithm

The LINPACK and LAPACK algorithms for `DPPFA` and `DPPTRF` are left-looking when `uplo` = 'U'. The LAPACK algorithm for `DPPTRF` is right-looking when `uplo` = 'L'. LINPACK does not have a Cholesky algorithm when `uplo` = 'L'. However, these algorithms are not suited for our BLAS 3 implementation of Cholesky using BHF. We only describe the `uplo` = 'L' algorithm here. We choose to mimic the `uplo` = 'U' algorithm of LAPACK `DPOTRF`. This algorithm could be called hybrid since it has both right- and left-looking characteristics. It is better to choose the `uplo` = 'U' algorithm because all `DGEMM` computations become 'T', 'N' instead of 'N', 'T' ; see [13] for details.

However, there are stronger reasons to choose the hybrid algorithm. First, we choose it because each block triangle is updated, factored, and does all its scalings in the outer J loop of our block hybrid Cholesky (BHC) algorithm which we now describe briefly. There are $n1 = \lceil n/\mathtt{NB} \rceil$ passes through the outer J loop. To be able to only use BLAS 3 we need each blocked packed triangle to be in FULL format. We use a buffer `T` of size $\mathtt{NB}^2$ to copy a packed triangle to full format in `T`. At the beginning of a pass through the J loop we start a K loop that calls `DSYRK` and `DGEMM` to update `T` and the rectangle (consisting of a set of squares - inner I loop) below `T`. Next, `T` is factored by kernel routine `DPOFU`. After factoring `T`, `DTRSM` is called to scale the rectangle (set of squares - I loop) beneath `T`. The pass through the J loop ends by copying full `T` back to packed format. Note here that the corresponding `DPPFA/DPPTRF` algorithm would copy each `T` back and forth multiple times. The use of `DSYRK` is a kernel routine call. For `DGEMM` and `DTRSM` there is two and one rectangles respectively, each consisting of a set of squares. Hence, a single call on two and one rectangles respectively requires no data copying within the calls. Alternatively, one could call the `DGEMM` and `DTRSM` kernels several times, once for each square in the rectangle set.

Another reason to choose the hybrid algorithm has to do with how its matrix operands enter L1 cache. Consider the kernel routine `DPOFU` and suppose the triangle `T` has order n. Let j be the outer loop variable. During the $j-th$ pass of the loop a rectangle of size $j(n+1-j)$ is accessed. A triangle above the rectangle of size $j(j-1)/2$ is no longer needed and another triangle to the right of the rectangle of size $(n-j)(n-j+1)/2$ has yet to be accessed. Note that these three figures have exactly $n(n+1)/2$ points. Now the rectangle has maximal area $n^2/4$ when $j = n/2$. However, using either the right- or left-looking algorithm leads to a maximal area of size $n^2/2$.

Now let us briefly look at the overhead of BHC. The major cost is the copying of $n1$ packed triangles of size NB to and from full format. A very minor cost is the overhead of calling the kernel routines. First, each kernel routine needs no error checking as it is an internal routine. There are $n1(n1+1)/2$ submatrices and $n1(n1+1)(n1+2)/6$ calls to kernel routines. These calls consist of $n1$ calls to DPOFU, $n1(n1-1)/2$ calls each to DSYRK and DTRSM, and $n1(n1-1)(n1-2)/6$ calls to DGEMM. Let us use an example to illustrate the tiny overhead. Assume, $n = 1000$ and NB=100 which is reasonable on IBM Power 3 machines. Now, $n1 = 10$ and so there are 220 kernel calls. Let $M = 1,000,000$. Each DGEMM call consumes $2M$ FLOPs, each DSYRK and DTRSM call consumes about M FLOPs and each DPOFU call consumes approximately $M/3$ FLOPs. Now $10(M/3) + 90M + 240M = 1000(M/3)$ which is about the FLOP count of Cholesky when $n = 1000$. Clearly, the calling overhead is tiny and so the overhead cost of BHC is the copying cost of packed triangles to full triangle and back (a total of 50,500 matrix elements). Of course, we have not included the cost of Point 7 in the Introduction. This cost consists of moving, in-place, $n(n+1)/2 = 500,500$ matrix elements.

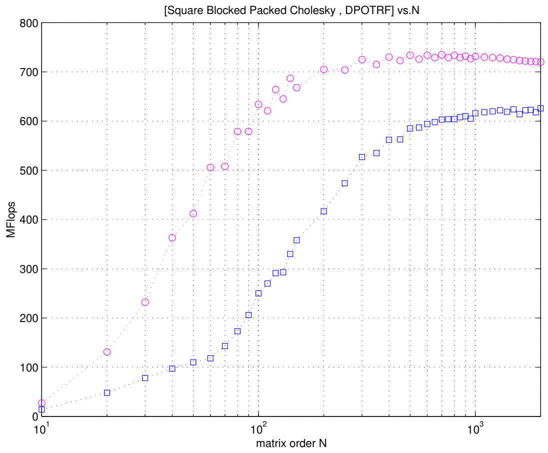
In this section, we discussed a fully portable replacement for LINPACK DPPFA and LAPACK DPPTRF. As stated, in the previous Section and the Introduction, the new algorithm is a direct translation of a vanilla point algorithm where each scalar operation is replaced by level 3 kernel operation that runs at nearly peak performance. And, we only used existing level 3 BLAS and kernel routine DPOFU. Actually, only the kernel routine parts of the level 3 BLAS were required.

4.3 Programming Notes for Square Blocked Packed Formats

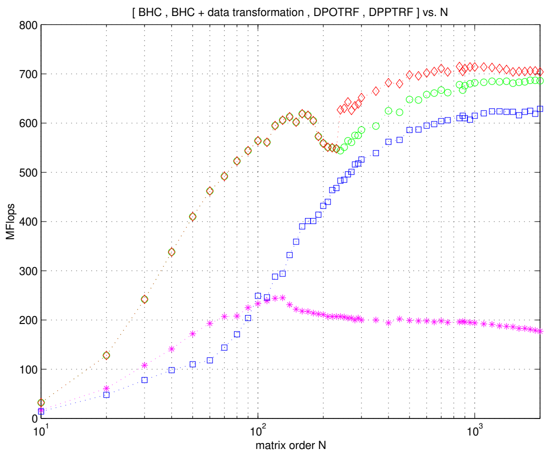
Like ordinary packed formats the implementor of a packed format library code explicitly handles his own addressing; e.g. AP(IJ) points at $a(i,j)$ in the packed array AP representing the symmetric/triangular array A. We do the same thing for square blocked packed format and BHF; e.g. see Figure 9, where AP is dimensioned a(*).

4.4 Performance for Cholesky

There are two plots. Both plot MFLOPS versus matrix order n. Note that the x-axis is log scale; we let n range from 10 to 2000. In the comparison of Square Blocked Packed Cholesky versus DPOTRF we do not include the cost of transforming the data format. This is perhaps unfair. Nonetheless, we did it to demonstrate what type of performance is possible. Note that DPSTRF shows some choppy behavior especially when n is small. The matrix orders where this occurs are not multiples of four. For example, when $n = 70$ the performance is about the same as $n = 60$. This is because DPSTRF is solving an order $n = 72$ problem, while the MFLOPS calculation is being done for $n = 70$. However, the kernel routines are much simpler when there is no fixup code. Note that DPSTRF is always faster than DPOTRF by as much as a factor of four when $n = 60$ and at least 15 % for n = 2000.



In the comparison for BHC versus LAPACK we give four graphs: BHC, BHC + data transformation, DPOTRF, and DPPTRF; we name these curves 1, 2, 3, and 4. For small n we do not use the data transformation. In fact, we wrote a packed format Cholesky factorization kernel for uplo = 'L' when n is small. Note that the L1 cache on Power 3 holds 8192 double words and that this factor kernel peaks at 620 MFLOPS for $n > 160$. In Section 4.4 we showed that $n^2/4$ is the effective cache size and so, $n \approx 180$. Note, the initial data transformation does cost something. The actual crossover happened at $n = 230$. For $n \leq 230$ the curves 1 and 2 are identical. For $n > 230$, it pays to use the data transformation and the curves 1 and 2 separate. A fair comparison would be curve 2 versus curve 4. Curve 2 is, on the average, three times faster than curve 4. Now we address the unfair comparison of curve 2 versus curve 3. Curve 2 is much faster than curve 3 for small n (up to four times faster) and more than 10 % faster for large n.



5 Lower Packed Blocked Overlapped Format for Symmetric Indefinite Factorization

In this Section we consider the solution of a linear system, $Ax = b$, where A is symmetric indefinite. The most popular algorithm for the solution of this problem uses the Bunch-Kaufman pivoting [8, §4.4], [14, §10.4.2] for the LDL^T decomposition of a matrix, A . There are two types of subroutines in LAPACK which utilize this method. In the first, the matrix is stored in a two-dimensional array, see Figure 8. As previously mentioned, conserving memory is sometimes an important issue. Clearly, full storage uses almost twice the necessary memory. Packed storage, see Figure 5, is the second type of storage scheme. Using this storage format, a one-dimensional array is used to store only the essential part of the matrix.

We introduce Lower Packed Block (LPB) storage as a generalization of packed storage. In LPB format, the columns of the matrix are divided into blocks which are stored successively in memory. Several successive columns of the matrix are kept inside each block as if they were in full storage. The result of this storage is that it allows the usage of level 3 BLAS. Of course, this format requires slightly more memory than does the packed format. However, for problems of practical interest, the memory overhead is only about 5%. Thus, the new storage scheme combines the two advantages of the storage formats used in LAPACK: the smaller memory size of LAPACK's `DSPSV` and the superior performance of LAPACK's `DSYSV`.

This new storage layout is a generalization of both packed and full format. In fact, we modify this new format slightly to produce Lower Packed Blocked Overlapped (LPBO) format. This was done in order to handle efficiently the 1×1 or 2×2 blocks that occur during symmetric indefinite factorization.

5.1 A Generalization of the Traditional Symmetric Formats

Let $1 \leq NB \leq N$ where NB is the block size. Let $n_1 = \lceil N/NB \rceil$ and $n_2 = N + NB - n_1 * NB$ The new LPB format partitions the N columns of A into $(n_1 - 1)$ block columns, each consisting of NB successive columns of A, and a last block consisting of n_2 columns. The `LDA` of block i is `LDA` $- (i-1)NB$. An example of LPB format with `NB` $= 4$ is given in Figure 11a. Note that when `NB` $= N$ one gets full format (see Figure 8a), whereas when `NB` $= 1$ and `LDA` $= N$ one gets lower packed format. Hence, the LPB format generalizes both traditional formats.

5.2 A Very Brief Description of Symmetric Indefinite Factorization

First note that a block factorization stage produces

$$PAP^T = LDL^T = WL^T = LW^T; W = LD,$$

where P is a permutation matrix, L is unit lower triangular, and D is block diagonal consisting of 1×1, or 2×2 blocks. More detailed descriptions can be found in [8, §4.4], [14, §10.4.2]. Our algorithm produces results identical to those of LAPACK's `_SYTRF` routine.

5.3 A Modification of LPB Format Called Lower Packed Block Overlapped Format (LPBO)

Symmetric Indefinite Factorization produces either 1×1 or 2×2 pivot blocks in an arbitrary order, depending on the numerical values of A. The LPB data format does *not* allow for the efficient handling of the either kind of pivot block when they occur at the boundary between blocks of columns. To overcome this inefficiency, we introduce the LPBO format. To obtain the new LPBO format (see Figure 11b) we pad, at the end of the first $n_1 - 1$ block columns, an extra `NB` storage locations.

We now illustrate how this padding allows us to handle efficiently a block factorization and its level 3 update. The problem arises when we factor column j and j is multiple of NB. If column

```
      LPB format                               LPBO format

  1  *  *  *                              1  #  *  *  *
  2 14  *  *                              2 14  #  *  *
  3 15 27  *                              3 15 27  #  *
  4 16 28 40                              4 16 28 40  #
  5 17 29 41 49  *  *  *                  5 17 29 41 53  #  *  *  *
  6 18 30 42 50 58  *  *                  6 18 30 42 54 62  #  *  *
  7 19 31 43 51 59 67  *                  7 19 31 43 55 63 71  #  *
  8 20 32 44 52 60 68 76                  8 20 32 44 56 64 72 80  #
  9 21 33 45 53 61 69 77 81  *            9 21 33 45 57 65 73 81 89  #
 10 22 34 46 54 62 70 78 82 86           10 22 34 46 58 66 74 82 90 94
  *  *  *  *  *  *  *  *  *  *            *  *  *  *  *  *  *  *  *  *
  *  *  *  *  *  *  *  *  *  *            *  *  *  *  *  *  *  *  *  *
```

Figure 11: Lower Packed and Lower Packed Block Overlapped Format when $\mathtt{NB} = 4$

j starts a 1×1 pivot then there will be NB or NB-1 columns in the factor block. If column j starts a 2×2 pivot then there will be NB+1 or NB columns in the factor block. This means that the first update block will have NB or NB-1 columns. To illustrate, let $j = \mathtt{NB} = 4$. Column j starts either a 1×1 or a 2×2 pivot block. Let us first assume case 2, i.e., the pivot block is 2×2. We shall update columns 6:9 of block 2 and block 3 with columns 1:5, consisting of block 1 and column 1 of block 2. The updates are
$A(6:9, 6:9) = A(6:9, 6:9) - W(6:9, 1:5) * L^T(6:9, 1:5)$ and
$A(10:10, 10:10) = A(10:10, 10:10) - W(10:10, 1:5) * L^T(6:9, 1:5)$

In case 1, we need to update blocks 2 and 3 with block 1. This is the easier case. The updates are:
$A(5:8, 5:8) = A(5:8, 5:8) - W(5:8, 1:4) * L^T(5:8, 1:4)$ and
$A(9:10, 5:8) = A(9:10, 5:8) - W(9:10, 1:4) * L^T(5:8, 1:4)$

In either case it should be clear from studying the LPBO storage layout that both the above block computations are laid out in full storage format. Therefore, level 3 BLAS can be applied. Also, in both cases above, we also must update the last block, either $A(10:10, 10:10)$ or $A(9:10, 9:10)$. Again, it should be clear how this can be done using LPBO format.

The total memory we need for LPBO storage is exactly

$$\mathtt{S_{LPBO}} = \mathtt{A_{LPBO}} + \mathtt{WORK}$$

where

$$\mathtt{A_{LPBO}} = \mathtt{A_{N1}} + (\mathtt{LDA} - \mathtt{N} + \mathtt{N2}) * \mathtt{N2}$$

$$\mathtt{A_{N1}} = (\mathtt{N1} - 1) * (\mathtt{LDA} * \mathtt{NB} + \mathtt{NB})$$

$$-((\mathtt{N1} - 1) * (\mathtt{N1} - 2)/2) * \mathtt{NB}^2$$

$$\mathtt{N1} = (\mathtt{N} + \mathtt{NB} - 1)/\mathtt{NB}$$

$$\mathtt{N2} = \mathtt{N} + \mathtt{NB} - \mathtt{N1} * \mathtt{NB}$$

$$\mathtt{WORK} = (\mathtt{LDW} + 1) * \mathtt{NB}$$

$$\mathtt{LDW} \approx \mathtt{LDA}$$

Here, `WORK` plays the same role as the `WORK` array in LAPACK's `_SYTRF` routine. `WORK` holds the W matrix, where $W = LD$. Note that for large `N`, $\mathtt{A_{LPBO}}$ is only slightly greater than the size required for packed storage.

5.4 New LAPACK Algorithms for Symmetric Indefinite Factorization in ESSL

We have packaged our algorithms in ESSL. We now discuss the issues of portability and migration as they relate to these algorithms. Our algorithms are called `DBSSV`, `DBSTRF` and `DBSTRS`. They provide the same functionality as the LAPACK algorithms `DSPSV`, `DSPTRF`, and `DSPTRS`, respectively. In fact, the calling sequences of `DBSSV`, `DBSTRF` and `DBSTRS` are identical to those of `DSPSV`, `DSPTRF`, and `DSPTRS`. Thus, it suffices to describe `DBSTRF`: `SUBROUTINE DBSTRF(UPLO, N, AP, IPIV, NSINFO)`. The first four arguments of `DBSTRF` have meanings identical to those of `DSPTRF`'s first four arguments. `DSPTRF`'s fifth, and last, argument is the output only argument, `INFO`. For `DBSTRF`, `NSINFO` is both an input and an output argument. Hence, the differences between the two algorithms are BS (Blocked Symmetric) versus SP (Symmetric Packed) storage and `NSINFO` being both an input and an output variable, whereas `INFO` is output only.

5.4.1 Overview of `DBSTRF` and `DBSTRS`

Let `NT=N(N+1)/2` be the size of array `AP`. `DBSTRF` accepts lower or upper packed format `AP` (exactly like `DSPTRF`) depending on value of `UPLO`. After argument checking, `DBSTRF` computes `NSO`, the size of the LPBO format array plus the size of the `W` buffer. The `W` buffer is used by both `DBSTRF` and `DSYTRF`. If `NSINFO<NSO` then `NSINFO` is set to `-NSO` and no computation is done. Otherwise `DBSTRF` moves, in-place, packed `AP(1:NT)` to LBPO format. Then `DBSTRF` performs a level 3 Bunch-Kaufman pivoting on `APBO = AP(1:NSO)`. `DBSTRS` and `DSPTRS` have identical calling sequences:

```
SUBROUTINE DBSTRS(UPLO, N,
NRHS, AP, IPIV, B, LDB, INFO)
```

The only difference between `DBSTRS` and `DSPTRS` is the name (`BS` instead of `SP`) and the data format of `AP`. Depending on N, `AP` is either in packed format or in LPBO format. Note that `DBSTRS` is only called if `DBSTRF` successfully completes the factorization of AP. The crossover value that determines whether to employ packed format or LPBO format is decided by the routine `DBSTRF`.

5.4.2 Migration and Portability Consideration

We designed our subroutines `DBSTRF` and `DBSTRS` to minimize the effort required to migrate existing codes that call `DSPTRF` and `DSPTRS`. Changes are indeed minimal. For each call to `DSPTRF` and `DSPTRS`, one needs to change "SP" to "BS." Also, `DBSTRF` requires that the size of the array, AP, be input in argument `NSINFO`. If necessary, one can use the return of `-NSO`, in `NSINFO`, to allocate additional storage.

5.5 Innovations Leading to Program Efficiencies

Using the Algorithms and Architecture approach, we describe five innovations regarding our symmetric indefinite factorization algorithms that use LPBO format. These five are briefly described in the five subsections below.

5.5.1 Intermediate Back Pivoting

A difference between LINPACK and LAPACK is that LAPACK usually uses full back pivoting, whereas LINPACK does no back pivoting. For `DBSTRF`, we have introduced intermediate or block back pivoting. Intermediate pivoting saves pivoting operations and has almost the same effect on efficiency (speed) as does full back pivoting. In the context of using LPBO storage this is especially true. Block back pivoting is necessary for a level 3 factorization and is also sufficient to achieve level 3 performance for multiple solve.

5.5.2 The LPBO Format Allows for a level 3 Multiple Solve

We remark that LAPACK's `DSYTRS` algorithm is not a level 3 implementation. A 2×2 pivot apparently causes a storage problem with full format. However, the LPBO format does not have this problem. To see why this is so, consider a 2×2 pivot block, D, and its associated block of L :

$$D = \begin{pmatrix} a & b \\ b & c \end{pmatrix}, L = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In LAPACK $A(2,1)$ cannot simultaneously hold both b and zero and $A(1,2)$ is forbidden storage. However, with LPBO format, we are free to use both $A(2,1)$ and $A(1,2)$ (see the '#'s in Figure 11b, illustrating LPBO format). Thus, we represent D (UPLO='L') as:

$$D = \begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$$

In any case LAPACK's full `DSYTRS` code uses LINPACK type pivoting and hence is not a level 3 implementation.

5.5.3 WL^T is More Efficient Than LW^T

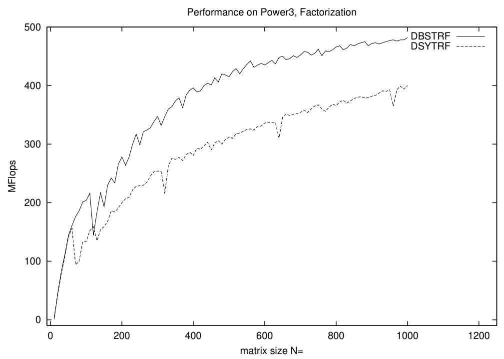
The level 3 part of `DSYTRF` and `DBSTRF` is the update computation $A \leftarrow A - LDL^T$. A buffer or work array W is used to compute $W = LD$. Now $WL^T = LW^T$ as $D = D^T$. LAPACK computes $A \leftarrow A - LW^T$ as opposed to $A \leftarrow A - WL^T$. The reason WL^T is to be preferred over LW^T (at least on IBM platforms and platforms using ATLAS, [22]) is that (during `DGEMM`) the left operand matrix (W in this case) fills the L1 cache. If LW^T were used, then L would fill the L1 cache. Now, the size of W is under the control of the algorithm implementor, as he sets its `LDW`, whereas L is an input array. Thus WL^T is to be preferred over LW^T.

5.5.4 The Workspace Size of W Can Be $\mathtt{NB}^2$ and Not `N*NB`

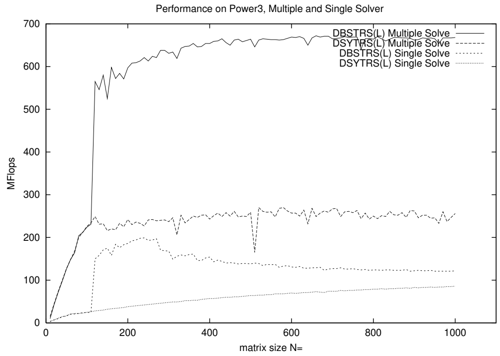
During the factor part of `DBSTRF`, we only need to compute the diagonal block of L and not the entire panel of L. However, we must store W on top of A since the buffer size is only $\mathtt{NB}^2$. This means the benefit mentioned in Section 5.5.3 might be lost. When factorization is complete we overwrite the diagonal part of W with L from the buffer. Also, during update $A \leftarrow A - WL^T$ we can update a block panel at a time. Again, we only need to compute a square block of L to do this. After a block panel of A is updated the square L block created in the buffer is copied over the corresponding part of W which is occupying A's storage.

5.5.5 The UPLO='U' Case

Using a stride of one gives the best performance. The UPLO='L' case uses stride-one for the most part. Stride-`LDA` can be very costly, especially on IBM processors. This very costly case occurs when UPLO='U' and one factors A in the usual "top-down, left-right" manner. However, one can just as well factor A "bottom-up, right-left". When this is done, the UPLO='U' case "converts to" the UPLO='L' case. Now both LINPACK and LAPACK almost follow this strategy. Our reason to nearly follow them is based on performance; we have not seen this reasoning for doing so presented in the literature.



(a) Factorization



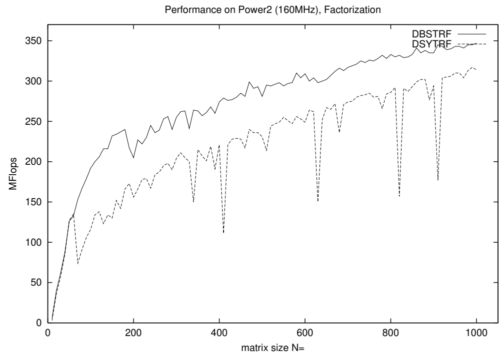
(b) Solver

Figure 12: Performance on IBM Power3

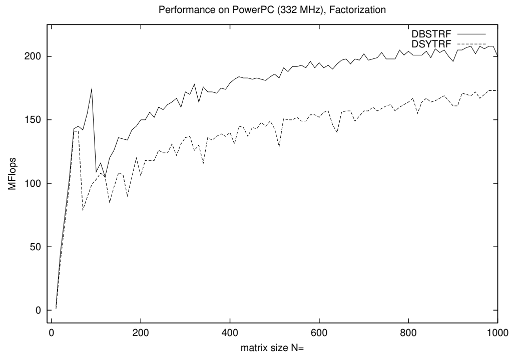
5.6 Performance

We tested our algorithms on three recent IBM platforms: Power3, Power2, and PowerPC 604e. In each figure we compare our new `BS` routines to LAPACK's `SY` routines. All plots measure MFLOPS versus matrix order, which runs from 10 to 1000 in steps of 10.

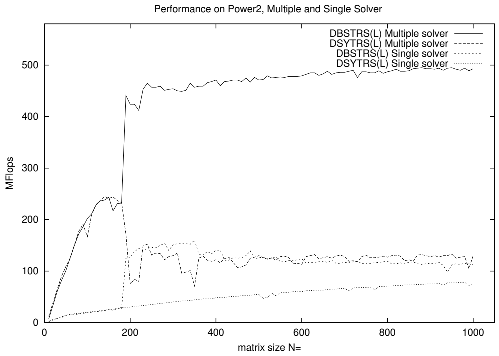
In Figure 12(a) we consider factorization, performed on a 200 MHz Power3. Initially both programs have identical performance because the algorithms are identical. There is crossover point where `DBSTRF` switches from packed to LPBO format. The first big drop (caused by program loading) in Figure 12(a) is where this crossover occurs. It can be seen that `DBSTRF` outperforms `DSYTRF` by up to approximately 40% and this includes the cost of converting from packed to LPBO format. Also, note that for $N > 300$ there are three dips in `DSYTRF`'s performance (see Section 5.5.3 for an explanation). In Figure 12(b), we compare `DBSTRS` to `DSYTRS` for multiple and single right hand sides. Here we see a dramatic improvement of using intermediate block factorization format over using LINPACK factorization format. Figure 13(a) shows factorization performance on an IBM 160 MHz Power2 machine. The new features to note in this graph are the large performance dips that occur for the `DSYTRF` algorithm. These dips occur when the `LDA` of A (chosen here to equal N) does not produce a good data mapping into the four-way set associative L1 cache. Note that `DBSTRF` does not have these dips (see Section 5.5.3) because we set `LDW` $\approx$ `LDA` so as to produce a good L1 data mapping. In Figure 13(b), we did not choose `LDB` $= N$ because doing so might have produced similar dips. Figure 14(a) shows factorization performance on a 332 MHz PowerPC 604e machine. Relative to Power3 and Power2 this machine evinces poor floating point performance. Figure 14(b) shows how good level 3 performance can be, relative to the level 2.5 performance of `DSYTRS` (about a factor of 5).



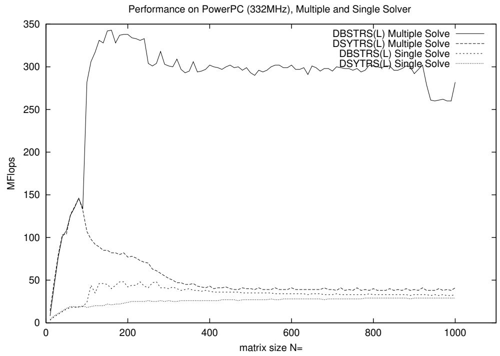
(a) Factorization



(a) Factorization



(b) Solver



(b) Solver

Figure 13: Performance on IBM Power2

Figure 14: Performance on IBM PowerPC 604e

6 Vectors

Vectors generalize as follows. Each vector is a collection of subvectors. The subvectors have the same format as Fortran 77 vectors: `X(0:NB-1:INCX)`. The "stride" between two subvectors (either constant or variable) has to be determined. This vital information becomes part of the definition of the vector as a collection of subvectors.

7 Recursion

It is not the purpose of this paper to cover how recursion interacts with linear algebra algorithms. The purpose of this section is to alert the reader to the fact that recursion can be very useful. Recursion helps improve the performance of dense linear algebra algorithms in at least two ways. First, it provides for automatic variable blocking, see [9, 12]. Hence, it provides a heuristic to block for the various higher levels of the memory hierarchy: L2, L3, main memory, out-of-core. This paper describes block based data structures stored in the conventional column-major or row-major order manner. However, recursion provides or requires a new way to store the blocks. To be able to do that we can address the blocks through the use integer tables. Since each of the blocks contain $\mathtt{NB}^2$ elements, the number of blocks will be relatively small. This means that the additional storage for the tables will be tiny.

The second way that recursion helps is more theoretical. It provides for new algorithms. For example, in [4], the IJK way of producing variants of a given linear algebra algorithm was described. Now, recursion introduces another way. In particular, we have mentioned factor kernels in this paper and the fact that many libraries, e.g., LAPACK, produce only level 2 implementations of these algorithms. By using recursion these kernel routines, e.g., LAPACK `___TF2` routines, are automatically converted to level 3 routines. In fact, instead of LAPACK having both `___TRF` and `___TF2` routines, a single routine will now suffice.

8 How the BLAS Change

The main thing to note is that data copying has been removed. A new set of BLAS, namely factor kernels, must be defined. However, the conventional BLAS become simpler to write as there is no data copying nor data allocation to be considered. Only a blocking routine is required to schedule the various kernel routine calls. This blocking routine should take into account the underlying memory system consisting of L2 cache, L3 cache, and main memory. Now, using the recursion heuristic of Section 7, it is possible to write platform independent BLAS.

As a generalization, one could envision extending the BLAS concept to dense linear algebra algorithms themselves. For example, consider Cholesky factorization, which is described in Section 4. In Sections 4.1 and 4.2, two different blockings, algorithm `DPSTRF` and the algorithm of Section 4.1.2 (the current LAPACK blocking algorithm), are given for Cholesky factorization. Hence, making a choice, we will produce a "Cholesky BLAS" algorithm. In essence, this is the approach used in producing algorithms for the ESSL library, [15].

9 Kernel Routines

A kernel routine for a level 3 BLAS or for a factorization routine is that piece of code that performs the floating point operations. Vanilla codes for these routines are simple scalar codes consisting of three nested loops and are found in some text books. For example, the 'N', 'N' case of vanilla `DGEMM` has a statement `T = T + A(i,k)*B(k,j)` in the inner, `k`, loop and initially `T = C(i,j)`, etc. For the 'T', 'N' case, the inner loop statement is `T = T + A(k,i)*B(k,j)`, etc. We would name these codes `DAB` and `DATB`. We use the convention that the suffix `4` means that the outer `j,i` loops are both unrolled by four. Hence, 16 independent dot products, instead of one, are being done by the high performance production versions of these vanilla codes. Thus, `DAB4` and `DATB4` used in Figures 4 and 9 are the suffix 4 codes briefly described above. Currently, they are being implemented by hand. However, in principle, the kernel routines can be automatically produced by a compiler and/or pre-processor for IBM Power 3 processors. In [1], pages 569 and 570, further details are given. Note however, that now we use 4 by 4 unrolling instead of the 4 by 2 unrolling in [1]. The other kernel routines, `DLLNU4` in Figure 4 and `DPOFU4, DSLVL4, DTATA4` in Figure 9, are implemented in a similar fashion. Routines `DLLNU4` and `DSLVL4` are `DTRSM` kernel routines. `DTATA4` is a `DSYRK` kernel and `DPOFU4` is a Cholesky factor kernel. As we saw in Section 2.2, the majority of the operations performed by these four routines are, in fact, FMAs. And by the principle of linear superposition we can re-arrange the computations in all of these routines by unrolling their two outer loops by four. Thus, for the most part, the inner loop will consist of the same computation of 16 independent dot-products that was described above. In summary, we are again exhibiting the pervasiveness of matrix multiplication in dense linear algebra algorithms. Only the factor kernel routine, `RGETR3`, of Section 3.1 is more complex. It involves a combination of recursion and kernel routines along with logic to handle the partial pivoting aspects.

10 Summary and Conclusions

I have described several novel data formats for dense linear algebra and have described some simple, novel algorithms that utilize these new data structures. I have relied on a heuristic that is the key factor governing performance on processors with deep memory hierarchies, namely blocking or tiling. To be able to use this heuristic we have made use of the following fact from linear algebra: "Some point algorithms have a submatrix block formulation." What is lacking are many more concrete demonstrations that these data formats improve the performance of standard linear algebra software, as typified by ESSL and LAPACK. This paper shows that the result is true for Cholesky and Symmetric Indefinite factorization and for LU factorization with partial pivoting. Not presented are results on QR factorization and Linear Least Squares via QR factorization. However, in [5, 6], we present evidence that indicates that these results are similar. Finally, there is agreement that the new software that is being developed can become part of LAPACK and ESSL if sufficient gains in performance and or storage utilization are demonstrated. These preliminary results indicate that this will happen.

11 Acknowledgements

The work described here is the outcome of my collaboration with HPCN at the University of Umea, Sweden and Uni-C in Lyngby, Denmark. At Umea my collaborators are Erik Elmroth, Andre Henriksson, Isak Jonsson, Bo Kagstrom, and Per Ling. At Uni-C they are Bjarne-Stig Andersen, Alex Karaivanov, Minka Marinova, Jerzy Wasniewski, and Plamen Yalamov. In some sense they should be co-authors. Certainly, this work would not have progressed to this point without our collaboration. A catalyst for our progress is Carl Tengwall of IBM Sweden who believed from the beginning that these collaborations would be an excellent match between Academia and Industry. Finally, I would like to thank John Gunnels of IBM Research for carefully reading the paper and suggesting several ways to improve its exposition.

References

[1] R. C. Agarwal, F. G. Gustavson, M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, Vol. 38, No. 5, Sep. 1994, pp. 563,576.

[2] G. Birkhoff and S. MacLane. A Survey of Modern Algebra, revised edition 1953, Book. *Macmillan Company* New york.

[3] R. K. Brayton, F. G. Gustavson and R. A. Willoughby. Some Results on Sparse Matrices. *Mathematics of Computation*, Vol. 24, No. 112, Oct. 1970, pp. 937,954.

[4] J. J. Dongarra, F. G. Gustavson, A. Karp. Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine. *SIAM Review*, Vol. 26, No. 1, Jan. 1984, pp. 91,112.

[5] E. W. Elmroth and F. G. Gustavson. Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. *IBM Journal of Research and Development*, Vol. 44, No. 4, July 2000, pp. 605,624.

[6] E. W. Elmroth and F. G. Gustavson. A New Much Faster and Simpler Algorithm for LAPACK DGELS. *BIT*, Vol. xx, No. y, AAAA 200x, pp. aaa,bbb.

[7] B. M. Fleisher. Private Communication. IBM, Yorktown Heights. Sep. 2001.

[8] G. Golub, C. VanLoan. Matrix Computations, Book. *Johns Hopkins Press* Baltimore, 1996

[9] F. G. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, Vol. 41, No. 6, Nov. 1997, pp. 737,755.

[10] F. G. Gustavson. Notes sent to Umea on data formats and descriptors. March 1998.

[11] F. G. Gustavson. Notes on Blocked Packed Format and Square Blocked Format for Symmetric/Triangular Arrays. Sep. 1999

[12] F. G. Gustavson, A.Henriksson, I. Jonsson, B. Kagstrom, P. Ling. Recursive Blocked Data Formats and BLAS Dense Linear Algebra Algorithms. In B. Kagstrom et.al., editors *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, No. 1541, 1998, pp. 195,206.

[13] F. G. Gustavson and I. Jonsson. Minimal Storage High Performance Cholesky via Blocking and Recursion *IBM Journal of Research and Development*, Vol. 44, No. 6, Nov. 2000, pp. 823,849.

[14] N. J. Highham. Accuracy and Stability of Numerical Computations, Book. *SIAM* Philadelphia, 1996.

[15] IBM. IBM Engineering and Scientific Subroutine Library for AIX Version 3, Release 3. *IBM Pub. No. SA22-7272-04* Dec. 2001.

[16] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK Users' Guide Release 3.0*, SIAM, Philadelphia, 1999, (http://www.netlib.org/lapack/lug/lapack_lug.html).

[17] J. J. Dongarra, C. B. Moler, J. R. Bunch, G. W. Stewart. *LINPACK Users' Guide Release 2.0*, SIAM, Philadelphia, 1979.

[18] L. Mirsky. An Introduction to Linear Algebra, revised edition 1972, Book. *Dover Publications* Mineola, New york.

[19] R. K. Montoye, E. Hokenek, S. L. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, Vol. 34, No. 1, Jan. 1990, pp. 59,70.

[20] R. K. Montoye. Private Communication. IBM, Austin. Sep. 2001.

[21] S. Toledo., A survey of out-of-core algorithms in numerical linear algebra. Book, *External Memory Algorithms*, eds., J. M. Abello and J. S. Vitter, pp. 161,179, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999.

[22] R. Clint Whaley, Antoine Petitet, Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *LAWN Report #147*, Sept., 2000, pp. 1-33.