NEW GENERALIZED MATRIX DATA STRUCTURES LEAD TO A VARIETY OF HIGH-PERFORMANCE ALGORITHMS

Fred G. Gustavson
IBM T. J. Watson Research Center
Yorktown Heights NY, USA

Abstract We describe new data structures for full and packed storage of dense symmetric/triangular arrays that generalize both. Using the new data structures one is led to several new algorithms that save "half" the storage for symmetric matrices and outperform the current blocked based level 3 algorithms in LAPACK. We concentrate on the simplest forms of the new algorithms and show they are a direct generalization of LINPACK. This means that level 3 BLAS's are *not* required to obtain level 3 performance. The replacement for Level 3 BLAS are so-called kernel routines, see [1], and on IBM platforms they are producible from simple textbook type codes, by the XLF Fortran compiler. In the sequel I will label these "vanilla" codes. On Power3 with a peak performance of 800 MFlops, the results for Cholesky factorization at order $n \geq 200$ is over 720 MFlops and then reaches 735 MFlops at $n = 400$. Using conventional full format LAPACK DPOTRF with ESSL BLAS's one first gets to 600 MFlops at $n \geq 600$ and only reaches a peak of 620 MFlops. The simple algorithms of LU factorization with partial pivoting for this new data format is a direct generalization of LINPACK algorithm DGEFA. Again, no conventional level 3 BLAS's are required; the replacements are again so-called kernel routines. Programming for squared 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. Also we mention that other more complicated algorithms are possible; e.g., recursive ones. The recursive algorithms are also easily programmed via the use of tables that address where the blocks are stored in the two dimensional recursive block array. Finally, we describe block hybrid formats. Doing so allows one to use *no* additional storage over conventional (full and packed) matrix storage. This means the new algorithms are *completely portable*.

Keywords: generalized data structures, BLAS, ESSL, LINPACK, LAPACK

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35407-1_22
R. F. Boisvert et al. (eds.), *The Architecture of Scientific Software*

1. INTRODUCTION

In [6],[9], Recursive Blocked Data formats were introduced as a replacement for standard Fortran/C array storage. One of the key innovations 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 [6], [9] 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, [6, 9]. 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 $0 \leq i <$ M and $0 \leq j <$ N and write $i = I*NB + i1$ and $j = J*NB + j1$. 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 $i1 + NB*j1$ 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 meaning there will be 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]. If one was to adopt these new simpler data formats then BLAS's and LAPACK type algorithms become almost trivial to write. We will discuss an example in the next paragraph, and in Sections 2 and 3 will show two 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 memory hierarchy. We use the Algorithms and Architecture approach, see [1] to present the argument.

1 Floating point arithmetic can't be done unless it's operands are 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 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 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 counter part `A(I:I+NB-1,J:J+NB-1)` algorithm.

   (a) Golub and Van Loan's "Matrix Computations" book, [5].

   (b) LAPACK library.

5 Some square 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 submatix 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. Point 4 to 6 are dense linear algebra algorithms facts. The book [5] , point 4a, gives a detailed listing of the scalar algorithms and describes (with references to the research literature) their block submatrix counter parts. The LAPACK library, point 4b, gives code for the submatrix counter part algorithms. However, the block submatrix codes of point 4 use Fortran and C to input their matrices and so point 5 does *not*

hold. See page 739 of [6] 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 demonstrates that near peak performance will be achieved. Points 7 and 8 suggest an obvious algorithm 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. Now 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 any depth. For example, LAPACK factor routines are level 2 routines; they are named with suffix TF2, and they call level 2 BLASes repetitively. On the other hand ESSL, [1], and more recently, [3, 6, 9] where recursion is used, have produced level 3 factor kernels. In the above argument we did not metion L2, L3 cache, memory, out-of-core. So, 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 QR factorization of a `M` by `N` matrix A. This code has a block equivalent where the stride one distance is $\texttt{NB}^2$. The row stride for the vanilla code is `LDA`. For the block equivalent it is m*$\texttt{NB}^2$. It is quite easy to write the block equivalent code from the vanilla code. In the vanilla code a floating point operation is usually a FMA ($c = c + ab$) which in the 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, [2]. For these types of algorithms the BLASes are simply calls to kernel routines. In reality, the BLASes disappear entirely. However, more complicated algorithms can be employed; e.g., recursive algorithms. In that case, a BLAS's call access several submatrix blocks, [9, 6]. So, like a traditional BLAS's call, a blocking routine is required to call the kernel routines multiple times on different sub-matrix operands as the blocking is now variable. However, unlike a traditional BLAS's routine no data copying is performed. This means performance improves.

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, is 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 in-place 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's 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 is portability: If one changes the input layout to 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 BHF for block hybrid formats. 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.

Now I turn briefly to SMP algorithms for Dense Linear Algebra Codes. In [3], superb SMP performance 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 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 give performance results on an IBM Power 3 processor for the hybrid format and compare them with the LAPACK `DGETRF` algorithm. Recently, other new block data formats, [8] have been discovered while working on this project, both at Uni-C and IBM Research. These have to do with triangular/symmetric arrays.

```
   1 13 25 37 49 61 73 85  97 109      1  5  9 13|49 53 57 61| 97 101 * *
   2 14 26 38 50 62 74 86  98 110      2  6 10 14|50 54 58 62| 98 102 * *
   3 15 27 39 51 63 75 87  99 111      3  7 11 15|51 55 59 63| 99 103 * *
   4 16 28 40 52 64 76 88 100 112      4  8 12 16|52 56 60 64|100 104 * *
                                      -----------|-----------|-----------
   5 17 29 41 53 65 77 89 101 113     17 21 25 29|65 69 73 77|113 117 * *
A= 6 18 30 42 54 66 78 90 102 114     18 22 26 30|66 70 74 78|114 118 * *
   7 19 31 43 55 67 79 91 103 115     19 23 27 31|67 71 75 79|115 119 * *
   8 20 32 44 56 68 80 92 104 116     20 24 28 32|68 72 76 80|116 120 * *
                                      -----------|-----------|-----------
   9 21 33 45 57 69 81 93 105 117     33 37 41 45|81 85 89 93|129 133 * *
  10 22 34 46 58 70 82 94 106 118     34 38 42 46|82 86 90 94|130 134 * *
  11 23 35 47 59 71 83 95 107 119     35 39 43 47|83 87 91 95|131 135 * *
   *  *  *  *  *  *  *  *  *   *      *  *  *  *| *  *  *  *| *   *  * *
```

Figure 1 Standard (left) and New Square Blocked (right) column major storage order

Again the major difference between these new array formats is they more closely mirror standard storage formats than 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 3 we describe square blocked packed formats for Symmetric/Triangular arrays and show 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 verses LAPACK's `DPOTRF/DPPTRF` are presented. In both Sections 2 and 3 we give brief notes on how to program for these new formats. Equally brief are Sections 4 to 6 (on Vectors, Recursion, and BLAS) which describe how the new formats effect/are affected by these three subjects. Due to lack of space we regret not covering these topics in any detail. In Section 7 we briefly describe what kernel routines are. In Section 8 we give a brief Summary and Conclusions.

2. SIMPLER SQUARE BLOCKED FULL FORMATS FOR MATRICES

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

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 (left) the minimum storage for A is LDA*n = 120 double words. If the user supplied ns $\geq$ 144 elements (extra storage of $\geq$ 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.

2.1. 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, cols $k = j+1, n-1$ are processed in two steps. Let k be a generic column. First, a swap of row j and the pivot row is made. Secondly, 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 $bj = 0, n1-1$ 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 cols $j+nb$ to $n-1$ are processed in three steps. Let k:k+ks-1 be the generic block col. First, there is a forward pivot step. Next, there is a DTRSM computation whose first four parms are 'L', 'L', 'N', 'U', done by kernel routine DLLNU4. Finally, there is a DGEMM update whose first two parms are 'N', 'N' which is done by a series of calls to kernel routine DAB4. After these three steps 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 function as LAPACK routine DGETF2. However, it is a level 3 routine, done recursively, as the prefix R and suffix 3 indicates. Note that the vanilla routine, actually the Linpack routine DGEFA, does not have a back pivot step. So the blocked version does extra work which actually could be avoided.

```
      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 2 Full Hybrid Block column major storage order

Now I turn to how DGEFB might be packaged as a subroutine in standard Fortran. Following LAPACK, 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 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 storage is rearranged into square block format and then DBGTRF is executed using the new block algorithm. Like the LAPACK LWORK parm, the value returned in -NSINFO will be the value used by DBGTRF for good level 3 performance, namely m1*n1*NB^2.

2.2. SQUARE BLOCKED FULL HYBRID DATA FORMATS FOR MATRICES

Let A be m by n where LDA $\geq m$; i.e., A is stored in column major order. Assume $m \geq n$. A similar result holds for $n > m$. Let $n1 = \lceil n/\mathtt{NB} \rceil$ and $n2 = n + \mathtt{NB} - n1 * \mathtt{NB}$. Partition the column space of A into $n1 - 1$ pieces of size NB and a leftover piece of size $n2 \leq$ NB. This new format represent A as a set of $n1 - 1$ rectangles of size $m * \mathtt{NB}$ 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 $m2 \leq$ NB. Here $m1 = \lceil m/\mathtt{NB} \rceil$ and $m2 = m + \mathtt{NB} - m1 * \mathtt{NB}$. Matrix A is a $m1$ by $n1$ block matrix. Each block has a TRANS parm; i.e., it can be stored in column or row major order. In each of the $n1$ rectangles the last LDA $- m$ rows are stored last. For the original A we assume LDA $\approx m$. In Figure 2 we give matrix A associated with Figure 1.

2.2.1 A Matching BLAS 3 LU = PA Algorithm. It is almost as easy to code this algorithm as the DGEFB algorithm of Figure 4. Do to lack of space we do not include the code for this format.

2.3. 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 the third dimension. See Section 5 ahead on how to handle 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 4 ahead. Because of this, the number of parameters, and generalization to algorithms for distributed memory processors, I predict that descriptors will eventually be used. In [7], I described some preliminary descriptor formats.

2.4. 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 tried several variants of the new data structures. Here we show performance results on a 200MHZ IBM Power 3 processor with a peak MFlop rate of 800. Results are for Full Hybrid Block format. We do not do the data transformation immediately. The reason is that 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 LAPACK algorithm DGETRF. Note that the x-axis is log scale; we let n and m of $(m, 100)$ range from 10 to 2000. For square matrices (Figure 2.4, left) the new code is 90 % to 10 % faster than DGETRF as n ranges from 10 to 2000. Note that for very large n these codes are dominated by ESSL's DGEMM code. The second graph (Figure 2.4,right) 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.

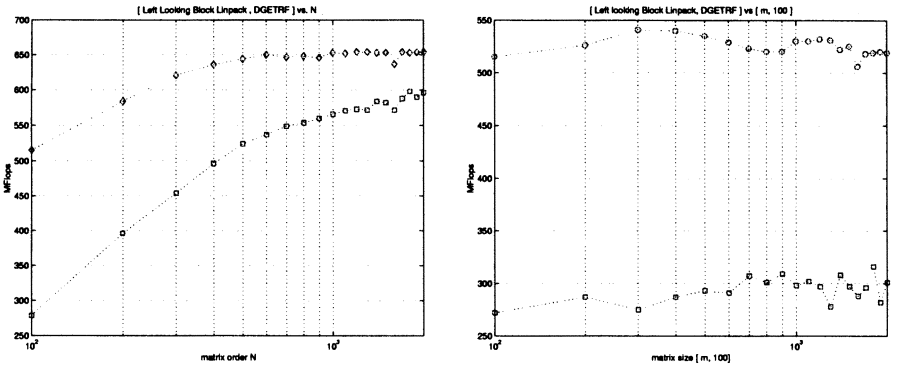

Figure 3 Performance for $LU = PA$

3. SQUARE BLOCKED PACKED FORMATS FOR SYMMETRIC/TRIANGULAR ARRAYS

This new format is a generalization of packed format for triangular arrays. It is 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 storage of the full array cases. In packed format, the elements of a triangular matrix is laid out in memory as follows : (see Figure 5) the numbers represent the position within a data array.

For square blocked packed formats there are two parameters NB and TRANS where usually $n \geq$ 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 $n1 = \lceil n/\text{NB} \rceil$. 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, etc. 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 (with TRANS = 'T') is given in Figure 6(left). 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 blocked lower packed array consists of 6 square block arrays: the first three are 4 by 4, 4 by 4, and 2 by 4. The next two are 4 by 4 and 2 by 4. The last square block is 2 by 2. Note the padding, which is done for ease

```
      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+jb-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

```
 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 * *
 *  *  *  *| *  *  *  *| *  * * *
 *  *  *  *| *  *  *  *| *  * * *
```

```
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 6 Square Blocked Lower (left) and Upper (right) Packed Format

of addressing. Addressing this set of six square blocks as a composite block array is straight forward. An example of Square Blocked Upper Packed Format (TRANS = 'N') is given in Figure 6(right). The blocked upper packed array consists of 6 square block arrays: the first is 4 by 4. The next two are 4 by 4. The last three are 4 by 2 4 by 2, and 2 by 2. 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 straight forward.

Here is another important point. With extra storage appended directly below a standard packed array one can move to these new data formats without extra storage. For the examples above, AP requires 55 storage elements. If there is 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 $n1$*NB^2 either in row major or column order. The entire buffer is then copied back over the vacated block column.

uplo = 'L'

1	*	*	*	*	*	*	*	*	*
2	14	*	*	*	*	*	*	*	*
3	15	27	*	*	*	*	*	*	*
4	16	28	40	*	*	*	*	*	*
5	17	29	41	53	*	*	*	*	*
6	18	30	42	54	66	*	*	*	*
7	19	31	43	55	67	79	*	*	*
8	20	32	44	56	68	80	92	*	*
9	21	33	45	57	69	81	93	105	*
10	22	34	46	58	70	82	94	106	118
*	*	*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*	*	*

uplo = 'U'

1	13	25	37	49	61	73	85	97	109
*	14	26	38	50	62	74	86	98	110
*	*	27	39	51	63	75	87	99	111
*	*	*	40	52	64	76	88	100	112
*	*	*	*	53	65	77	89	101	113
*	*	*	*	*	66	78	90	102	114
*	*	*	*	*	*	79	91	103	115
*	*	*	*	*	*	*	92	104	116
*	*	*	*	*	*	*	*	105	117
*	*	*	*	*	*	*	*	*	118
*	*	*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*	*	*

Figure 7 Square blocked packed formats when NB = n

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 by replacing each reference to an i, j element by its corresponding square block submatrix. 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 turn to full format. We continue the example with N = 10, and LDA = 12. Just 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 7).

In Figure 7 we ignore the last 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. I use colon notation, see [5]. 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.

3.1. A CHOLESKY ALGORITHM FOR SQUARE BLOCK PACKED FORMAT

Now I turn to programming dense linear algorithms in the new formats. As an example, Figure 8 give uplo = 'L' code for DPSTRF (PS stands for positive definite symmetric) which produces the lower Cholesky factor for 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 Linpack routine DPPFA. Routine DPOFU4 is a ESSL factor kernel. The corresponding scalar operation is square root. In ESSL, all level 3 BLAS's 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 operates on the blocks. Data copying of the operands to the kernel routines is decided on by the ESSL DGEMM blocking routine. Now in DPSTRF we can call the kernel routine DATB4 directly, thereby avoiding copying, since NB was chosen 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 the 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, Points 7 and 8, of the introduction. In Figure 3.4(left) we give the performance of DPSTRF verses 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 procedure 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 parm NS stands for the storage the user inputs for AP; NS $\geq n(n+1)/2$. Hence, the name NSINFO. On output NSINFO plays the role of LAPACK output *only* parm INFO. If NS is not sufficiently large, DPSTRF just returns in -NSINFO the amount of storage necessary for good level three performance. Like the LWORK

```
!  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)
        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
          call dslv14(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 8 DPSTRF subroutine.

```
 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 9 Blocked Hybrid Lower Packed Format

parm, 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.

3.2. 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 9 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 $n1 = \lceil n/\texttt{NB} \rceil$ and $n2 = n + \texttt{NB} - n1 * \texttt{NB}$. 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 $nt = h(h + 1)/2$ 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 9. 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.

3.2.1 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 [4], 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, each consisting of a set of squares. Hence, a single call on two and one rectangles requires no data copying within the calls. Alternatively, one could call the DGEMM and DTRSM kernels several times, once for each

square in the 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 sf 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 M Flops and each DPOFU consumes approximately $M/3$ Flops. Now $10(M/3) + 90M + 240M = 1000(M/3)$ which is 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 include the cost of Point 7 in the Introduction. This cost consists of moving inplace $n(n+1)/2 = 500,500$ 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 BLASes were required.

3.3. 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 8 where `AP` is dimensioned `a(*)`.

3.4. PERFORMANCE FOR CHOLESKY

There are two plots. Both plot MFlops verses 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 verses `DPOTRF` (Figure 3.4, left) 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 $n = 60$. The explanation is `DPSTRF` is solving an order $n = 72$ problem and the MFlop computation 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, (see Figure 3.4, right) for BHC verses 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 3.4 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 verses curve 4. Curve 2 is, on the average, three times faster than curve 4. Also, curve 2 is much faster than curve 3 for small n (up to four times faster) and more than 10 % faster for large n.

4. 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.

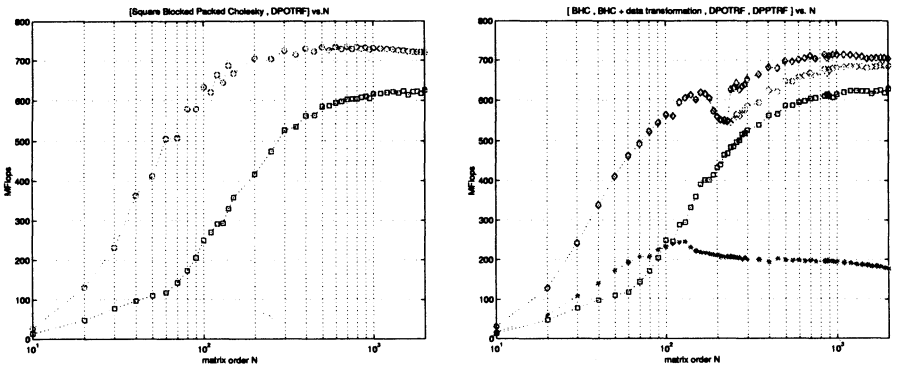

Figure 10 Performance for Cholesky

5. RECURSION

What we have now is block based data structures stored in the conventional column major or row major order. However recursion 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 the blocks contain NB^2 elements the number of blocks will be relatively small. This means that the additional storage for the tables will be tiny.

6. HOW THE BLAS'S CHANGE

The main thing to note is that data copying has been removed. A new set of BLAS's namely factor kernels must be defined. However the conventional BLAS's become simpler to write as there is no data copying nor data allocation to be considered.

7. 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 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`. Now the suffix 4 means 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 8 are the suffix 4 codes briefly described above. Currently, they are being done by hand.

However, in principle, the kernel routines can be automatically produced by a compiler and/or pre-processor for for IBM Power 3 processors. The other kernel routines `DLLNU4` in Figure 4 and `DPOFU4, DSLVL4, DTATA4` in Figure 8 are done in a similar fashion. Only kernel routine `RGETR3` is more complex. It involves a combination of recursion and kernel routines plus logic to handle the partial pivoting aspects. 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].

8. SUMMARY AND CONCLUSIONS

I have described several novel data formats for dense linear algebra and have described some novel simple 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 is many concrete demonstrations that these data formats improve the performance of standard linear algebra software that are typified by ESSL and LAPACK. This paper shows that the result is true for Cholesky factorization and for LU factorization with partial pivoting. Not presented are results on QR factorization. However, in [3] we present result that indicate this result will be true. 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.

Acknowledgments

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.

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] 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.

[3] 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.

[4] 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.

[5] G. Golub, C. VanLoan. *Matrix Computations*, Johns Hopkins Press, Baltimore and London, 1998.

[6] 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.

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

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

[9] F. G. Gustavson, A.Henriksson, I. Jonsson, B. Kagstrom, P. Ling. Recursive Blocked Data Formats and BLAS's 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.

DISCUSSION

Speaker: Fred Gustavson

Richard Hanson : Does this compelling result carry over to architectures other than the Power3?

Fred Gustavson : My collaborators Bo Kagstrom, Erik Elmroth, Isak Jonsson, and Per Ling at the University of Emea, Sweden, and Alexander Karaivanov, Minha Marinova, Jerzy Wasniewski and Plamen Yalamov at Uni-C in Lyngby, Denmark, have obtained similar results. In particular, the Uni-C group worked on recursive packed Cholesky and produced results on the Intel Pentium III, the Compaq Alpha EV6, the SGI R10000, the Sun Ultra Sparc II, and the HP PA-8500.

Kristopher Buschelman : Whose responsibility is it for the construction of these complicated data structures? That is, does the end user need to build a matrix into this format and then pass this data structure into your routines?

Fred Gustavson : I think these new data structures are perhaps simpler. However, the end user does *not* need to construct them. Either a compiler or algorithm writer can support them. For example, the $LU = PA$ algorithm described here accepts standard Fortran/C formats and converts this input into the new format.

Robert van de Geijn : One real opportunity presented by the new hierarchical data structure lies in the possible link that can be made with sparse matrices arising from structured problems. A hierarchical matrix can be used to store the sparse matrix by pruning parts of the hierarchy corresponding to zero submatrices.

Fred Gustavson : My approach works best for level 3 computations; e.g., matrix factorization. I want to assure that each matrix operand that enters L1 cache gets used by the floating point unit(s) multiple times so as to amortize the cost of bringing that operand into L1 cache.

Masaaka Shimasaki : Could you explain the accuracy results of your numerical experiments?

Fred Gustavson : The new algorithms have the same accuracy as the conventional algorithms.

David Walker : Have you compared your approach with a modified LAPACK algorithm in which each of the blocks is contiguous in memory?

Fred Gustavson : In some sense my algorithm is a modified LAPACK algorithm where the blocks are contiguous in memory. However, instead of calling LAPACK TF2 routines, I call level 3 factor kernel routines. And sometimes, I call kernel routines instead of Level 3 BLAS's. And finally, I have changed the LAPACK algorithms in some cases, e.g., I have introduced recursive algorithms.

Vladimir Getov : Your approach is specific for architectures with memory hierarchies. How do you see the suitability of this approach to future architectures on a long-term basis?
Fred Gustavson : I think architectures in the future will still have memory hierarchies and hence the approach will remain valid.