High-performance linear algebra algorithms using new generalized data structures for matrices

F. G. Gustavson

We present a novel way to produce dense linear algebra factorization algorithms. The current state-of-the-art (SOA) dense linear algebra algorithms have a performance inefficiency, and thus they give suboptimal performance for most LAPACK factorizations. We show that using standard Fortran and C two-dimensional arrays is the main source of this inefficiency. For the other standard format (packed one-dimensional arrays for symmetric and/or triangular matrices), the situation is much worse. We show how to correct these performance inefficiencies by using new data structures (NDS) along with so-called kernel routines. The NDS generalize the current storage layouts for both standard formats. We use the concept of Equivalence and Elementary Matrices along with coordinate (linear) transformations to prove that our method works for an entire class of dense linear algebra algorithms. Also, we use the Algorithms and Architecture approach to explain why our new method gives higher efficiency. The simplest forms of the new factorization algorithms are a direct generalization of the commonly used LINPACK algorithms. On IBM platforms they can be generated from simple, textbook-type codes by the XLF Fortran compiler. On the IBM POWER3 processor, our implementation of Cholesky factorization achieves 92% of peak performance, whereas conventional SOA full-format LAPACK `DPOTRF` *achieves 77% of peak performance. All programming for our NDS can be accomplished in standard Fortran through the use of three- and four-dimensional arrays. Thus, no new compiler support is necessary. Finally, we describe block hybrid formats (BHF). BHF allow one to use* ***no*** *additional storage over conventional (full and packed) matrix storage. This means that new algorithms based on BHF can be used as a backward-compatible replacement for LAPACK or LINPACK algorithms.*

1. Introduction

The Basic Linear Algebra Subroutines (BLAS) were introduced to make the algorithms of dense linear algebra (DLA) performance portable [1–3]. Starting with LINPACK [4] and progressing to LAPACK [5], the level 1, 2, and 3 BLAS were introduced. The different BLAS levels are distinguished by the number of nested "do loops" required to perform the indicated computation. Almost all of the

0018-8646/03/$5.00 © 2003 IBM

floating-point operations of DLA algorithms are performed through the use of BLAS calls. If performance were equal to operation count, performance would be truly portable. However, with today's deep memory hierarchies, this is no longer the case. To understand the performance inefficiency of LAPACK algorithms, it suffices to discuss the level 3 BLAS, `DGEMM` (Double precision GEneral Matrix Matrix). All LAPACK codes are written in terms of level 1, 2, and 3 BLAS, and `DGEMM` is the most important BLAS (see Section 2). The `DGEMM` interface definition requires that its matrix operands be stored as Fortran (column-major) or C (row-major) two-dimensional arrays. We refer to these storage schemes as *standard* two-dimensional arrays.

Design principles for producing a high-performance level 3 `DGEMM` BLAS are given in [6–9]. A key design principle for `DGEMM` is to partition its matrix operands into submatrices and then call an L1 kernel routine multiple times on its submatrix operands. Here L i stands for level i cache. L i is not to be confused with level i BLAS. By "kernel" routine, we mean a routine that performs matrix-multiply-type operations on matrix operands that are contiguous and of a form and size that permits optimal use of the L1 cache. Another key design principle is to change the data format of the submatrix operands so that each call to the L1 kernel can operate at or near the peak Million FLoating-point OPerations per Second, or MFLOPS, rate. The "optimal" data format for the L1 kernel routine is, essentially, the NDS advocated in this paper. This format change and the subsequent change back to standard data format is a cause of the performance inefficiency in `DGEMM` implementations. Any LAPACK factorization routine of a matrix, A, calls `DGEMM` *multiple* times, with all of its operands being submatrices of A. For each call, data copy is done; the principal inefficiency is therefore multiplied by this number of calls. However, this inefficiency can be removed by adopting the NDS and by creating a substitute for `DGEMM`, e.g., its L1 kernel routine, which does *not* require the aforementioned data copy.

In [10, 11], recursive blocked data formats were introduced as a replacement for standard Fortran or C array storage. One of the key insights was 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. We demonstrated that recursion (i.e., divide and conquer) should be used to order these blocks in storage. This storage arrangement leads to L2, L3, and memory blocking automatically. However, the ordering of the blocks is nonlinear, and tables are needed to properly address these blocks. A simpler way to order the blocks is standard Fortran or C order; i.e., store the blocks either in column-major or row-major order. The main benefit of the simpler data layout is that addressing of an arbitrary element $a(i, j)$ of matrix A can easily be handled by a compiler and/or a programmer. We call the NDS *simple* if the ordering of the blocks follows the standard data structure order.

For level 3 algorithms, the basis of the IBM Engineering and Scientific Subroutine Library (ESSL) [12] is a set of kernel routines that achieve peak performance when the underlying arrays fit into L1 cache [6, 12]. If one were to adopt these new, simple NDS, BLAS and LAPACK-type algorithms become almost trivial to write. Also, the combination of using the NDS with kernel routines is general, and for matrix factorization it helps to overcome the current performance problems introduced by having a nonuniform, deep memory hierarchy. We use the Algorithms and Architecture approach [6] to elucidate what we mean. The results are based on the eight points below. Points 1 to 3 are commonly accepted architecture facts about many of today's processors. Points 4 to 6 are facts about dense linear algebra algorithms that are easily demonstrated or proven. Points 7 and 8 are an obvious conclusion based on the Algorithms and Architecture approach.

1. Floating-point arithmetic cannot be done unless the operands involved reside in the registers of the floating-point unit.
2. Standard two-dimensional Fortran and C arrays do *not* always map well into the L1 cache.
   (a) The best case occurs when the array is contiguous and properly aligned.
   (b) At least a three-way set-associative cache is required when a matrix multiply operation is being performed.
3. For peak performance, all matrix operands must be used multiple times when they enter the L1 cache.
   (a) This ensures that the initial cost of bringing an operand into cache is amortized by its level 3 multiple reuse.
   (b) Multiple reuse of all operands can occur only if all matrix operands map well into the 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 (the LAPACK library [13]).
5. Some submatrix representations are both contiguous and fit into the L1 cache.
6. Dense matrix factorization is a level 3 computation.
   (a) Dense matrix factorization, in the context of point 4, is a series of submatrix computations.
   (b) Every submatrix computation (executing any kernel routine) is a level 3 computation, performed in the L1 cache.
   (c) A level 3 L1 computation is one in which each matrix operand is used multiple times.

From points 1–6, we conclude points 7 and 8:

7. Map the input Fortran or C array (matrix A) to a set of contiguous submatrices, each fitting into the L1 cache.
   (a) For portability (using BHF), perform the inverse map after applying point 8 (below).
8. Apply the appropriate submatrix algorithm.

In the block submatrix codes of the LAPACK library, input matrices are in standard or packed formats, so point 5 does *not* hold for LAPACK algorithms. (See the subsection on performance inefficiencies of LAPACK factorization algorithms and p. 739 of [10] for more details.) Point 5 does hold for the NDS described here. Assuming that 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. Point 7 is pure overhead for the new algorithms, and adopting the new data formats eliminates this overhead. By doing only point 8 we see that we can obtain near-peak performance, because every subcomputation of point 8 is a point 6(b) computation.

We now discuss the use of kernel routines in concert with NDS. Take any standard linear algebra factorization code, say Gaussian elimination with partial pivoting or the QR factorization of an `M` by `N` matrix, A. By "standard" we mean, loosely speaking, an element-wise matrix algorithmic description, in the spirit of the algorithms presented in [13]. These are often referred to as "vanilla" or pseudo codes. It is quite easy to derive the block equivalent code from the standard code. In the standard 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 a call to either a `DTRSM` or a `DTRMM` kernel. In the simple block equivalent codes we are led to one of the variants of IJK order [14]. For these types of new algorithms, the level 3 BLAS become simply calls to kernel routines. It is important to note that no data copying is being done. Also, performance portability is ensured, since different platforms would produce specific implementations of these kernel routines instead of the current level 3 BLAS.

One type of kernel routine that deserves special mention is the factor kernel. Neither LAPACK nor the research literature treats factor kernels in sufficient depth. For example, the factor part of LAPACK level 3 factor routines (those named with the suffix `TRF`) are level 2 routines; they are named with the suffix `TF2`, and they call level 2 BLAS repetitively. On the other hand, ESSL [6], and more recently [10, 11, 15], where recursion is used, have produced level 3 factor routines that employ level 3 factor kernels to yield level 3 factor components.

The above three paragraphs indicate that new algorithms can be obtained from simple DLA codes if one first introduces the simple NDS. 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, in place, to the NDS, column-major, square block format. Next, the block equivalent of the standard code (with calls to standard ESSL-type kernels) could be performed on the transformed NDS. 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 more than 720 MFLOPS, reaching 735 MFLOPS at $N = 500$. This measurement did not include the cost of transforming the data to square blocked packed format. Using conventional full-format LAPACK `DPOTRF` with ESSL BLAS, performance reaches 600 MFLOPS at $N \geq 600$ and achieves a peak of only 620 MFLOPS. Similar performance results are obtained for general and symmetric indefinite matrix factorization.

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. Since one can use only level 1 and packed level 2 BLAS with packed storage, performance suffers drastically on RISC and other systems. Thus, many users choose 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. Using the NDS instead of the standard packed format, we describe 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. We present new algorithms and performance results for Cholesky and symmetric indefinite factorization.

The simple NDS described in the above five paragraphs are *not* compatible with the existing standard data formats (i.e., the storage layout for a two-dimensional array). There is great resistance to changing data formats that have been used for a long time, and this is especially so for major programming languages such as Fortran and C. Also, libraries for dense linear algebra (e.g., LAPACK, LINPACK, and ESSL) all support packed-format symmetric/triangular arrays. Again, the simple NDS are *not* compatible. A compelling reason for this resistance is portability: If one changes the input layout used by a library subroutine/function, existing software which calls that library subroutine/function will not be operational. Therefore, we introduce modifications to our simple NDS: block hybrid format (BHF), which uses no additional storage over the standard data formats of dense linear algebra. In BHF we store the triangular part of a

trapezoidal matrix in packed format and the rectangular part as a general matrix (full format). In most cases, using BHF solves the portability replacement problem, since a new routine would then accept standard formats, convert them to BHF, compute on BHF, and convert BHF back to the standard formats.

In Section 2 we describe some basic algorithmic and architectural results as a rationale for the work we are presenting. 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 POWER3 processor for the hybrid format and compare them with the LAPACK `DGETRF` algorithm. This latter algorithm is a backward-compatible replacement for the LAPACK `DGETRF` routine. In Section 4 we describe column-major, 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 describe a BHF version of this simple NDS. For both types of NDS we describe an associated Cholesky factorization algorithm. Also, performance results on an IBM POWER3 processor for both algorithms and LAPACKs `DPOTRF/DPPTRF` are compared. The BHF algorithm is a backward-compatible replacement for the LAPACK `DPPTRF` routine. In Section 5 we describe a new storage layout for symmetric indefinite factorization and 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 greater than that of the LAPACK packed storage algorithms. Our new algorithms, called `DBSSV`, `DBSTRF`, and `DBSTRS`, are now part of ESSL [12]. Sections 6 to 8 (on vectors, recursion, and BLAS) briefly describe how the NDS affect or are affected by these three subjects. In Section 9 we briefly describe kernel routines for IBM platforms. In Section 10 we give a summary and present our conclusions.

2. Rationale and underlying foundations of our approach

We describe some basic DLA algorithmic and architectural results as a rationale for why the combination of NDS and kernel routines works so well. The idea is to elucidate, somewhat, the Algorithms and Architecture approach [6]. We see that for DLA algorithms, an architecture with an FMA instruction has a significant advantage over one using a floating-point multiply followed by a floating-point add. First, we describe the linear transformation approach to producing DLA algorithms and use it as a foundation for producing their high-performance implementations as follows. Every simple DLA code has an associated block (submatrix) algorithm. This latter code consists entirely of level 3 L1 kernel routine calls. Each call of every L1 kernel routine runs at or near the peak performance rate when the NDS are used. In fact, the main part of each level 3 L1 kernel routine consists of performing a series of independent FMAs. Second, we reintroduce operation counts as an accurate measure of algorithm performance. This metric becomes meaningful again largely because of the manner in which our new data structures interact with the memory hierarchy. We demonstrate this by considering the example of Cholesky factorization. The entire algorithm consists of level 3 L1 kernel routine calls in which no data copy occurs when NDS are used. For each call, operation count is a good performance indicator of the pertinent level 3 L1 kernel. Thus, under these conditions, operation count is a good performance indicator for the entire Cholesky factorization. Next, we clarify three aspects of the LAPACK design that cause it to have performance inefficiencies. And finally, we show that the combination of NDS and kernel routines corrects all performance inefficiencies attributed to the LAPACK design.

The linear (coordinate) transformation approach for producing DLA algorithms

The LAPACK library is a collection of about 500 DLA algorithms. In this section we attempt to describe all of these DLA algorithms by using a single central idea from linear algebra, namely the combined use of elementary matrices and linear transformations to produce a DLA algorithm. We state two theorems, without proof, as a means of indicating how we think the combination of NDS and kernel routines can be put on a rigorous foundation for all DLA algorithms. In fact, the same argument could be used for LAPACK and its use of level 3 BLAS. As an illustration of the first theorem, we demonstrate how two different LAPACK DLA factorization algorithms (DLAFA), `DQRTRF` and `DGETRF`, relate to this general approach. Next, we break the multiplication of an elementary matrix by a general matrix into its component parts and show that the FMA is a fundamental computational unit and, hence, an excellent choice for realization in a computer architecture. Then a discussion about the first theorem leads to the conclusion that DLAFA are little more than a repeated application of general matrix multiplication, `DGEMM`, on various submatrices of the matrix, A, that is being factored. The second theorem states that the way we now do matrix multiplication [6–9] subject to a cache hierarchy is optimal to within the constants in the big-O and Ω notations. This means that our current implementation practice is on a sound theoretical footing.

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 because it demonstrates some reasons why the Algorithms and Architecture approach [6] is so effective. Briefly, the Algorithms and Architecture approach states that the key to performance is to understand the interaction of the algorithm and the architecture. 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 [16], pp. 170–173, matrices and row equivalence are discussed. In particular, elementary row operations of three types are cited on p. 172. The most important operation is the addition of any multiple of one row to any other row of a matrix. For another treatment, see Chapter 6, Elementary Operations and the concept of Equivalence, in [17].

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: $E = I + \sigma u v^T$, 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, $1 \le i \le k$, represent k linear transformations where each T_i is elementary. Let $T = T_k T_{k-1} \ldots T_1$ be their product. 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 [15, 18]. We remark that Section 2, pp. 939–942 of [19] shows that the 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, pp. 606–615 of [15].

We now 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, `D=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 separately costs about 1.7 times more than an FMA[1,2] [20].

Next we claim that matrix multiplication is pervasive in the algorithms of dense linear algebra; for example, see [13] 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) can be viewed as *defining* matrix multiplication. In fact, in the 1840s Cayley first described a matrix as a rectangular two-dimensional array. According to Meyer [21], Cayley also defined matrix multiplication as the result of the composition of two linear coordinate transformations. We take the same view here. Our dual points of view let us describe dense linear algebra algorithms as a series of coordinate transformations with the aim of finding a set of basis vectors where the matrix A is now represented as upper-triangular. And for each such composition of transformations to be linear, we *must* perform matrix multiplication. For more details, see Chapter 8, pp. 209–214 of [16]. We also note that matrix multiplication is by *definition* a series of parallel dot-product operations and hence just a series of FMAs, which can be done independently.

We have just seen why matrix multiplication shows up repeatedly in, say, LAPACK algorithms. In fact, the level 3 BLAS, `DGEMM`, is considered 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.

[1] B. M. Fleisher, private communication, IBM Research Division, Yorktown Heights, NY, September 2001.
[2] R. K. Montoye, private communication, IBM Research Division, Austin, TX, September 2001.

Theorem

Any algorithm that computes $a_{i,k}b_{k,j}$ for all $1 \le i, j, k \le n$ must transfer between memory and an M word cache $\Omega(n^3/\sqrt{M})$ words if $M < n^2/5$. See [22].

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, since our current algorithms achieve the lower bound complexity measure. Also, practical experimental evidence (e.g., the ESSL `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 high-speed storage and use it multiple times to amortize the cost in performance of moving the data. In doing so, we satisfy points 1 and 3 of the Introduction. We briefly mention the translation lookaside buffer (TLB), cache, and register blocking. The TLB contains a finite set of pages which are known as the *current working set* of the computation. If the computation addresses only memory in the TLB, there is no penalty. Otherwise, a TLB miss occurs, resulting in a large performance penalty; see [6]. 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 [6].

Operation counts measure 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 valid because the memory hierarchy was uniform. In fact, it was during this time that the FLOP (floating-point operation) and the MFLOPS (million floating-point operations per second) became measures of floating-point computations. The MFLOPS, still widely used today, is now employed primarily as a measure of peak performance, since the memory hierarchies are no longer uniform.

Our introduction of new data structures, along with their use by kernel routines, brings operation counts back as an indicator of peak L1 cache performance 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. Also, by considering level 3 BLAS and factor kernels, we concretely demonstrate the pervasiveness of matrix multiplication. 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` ($C = C - A^TB$) 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^2n. 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` ($C = C - A^TA$) 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 $N = n$`NB`. Using point 4 of the Introduction, Cholesky factorization consists 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 factorization problem, occupying space in a nonuniform 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 uniform near-peak rate in L1 cache.

Description of four kernel routines

In the previous section, we gave operation counts for `DGEMM`, `DTRSM`, `DSYRK`, and Cholesky factorization. Let A be a general matrix of order N with a leading dimension `LDA` $\ge N$, i.e., A = `A(0:LDA - 1, 0:N - 1)`. We use colon notation; see [13]. Note that in Fortran, $a_{i,j}$ is stored in location $i + j \cdot$ `LDA` 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 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 $C = C - A^TB$, where A is k by m, B 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 $U = U - A^TA$, where U is order m symmetric stored in full upper format and A is n by m.
- `DPOFU4 (A,LDA,N,INFO)` computes U, where $U^TU = A$, A is order n symmetric positive definite, stored in full upper format, and $A \leftarrow U$.

We comment briefly. When the dimensions of the operand are all one, the first kernel computes $c = c - a * b$, the second computes $b \leftarrow b/a$, the third computes $u = u - a^2$, and the fourth calculates $u \leftarrow \sqrt{a}$. These four operations are the scalar counterparts of the four level 3 kernels listed above. These four level 3 L1 kernels perform at or near peak only when their operands are represented in the NDS format.

Performance inefficiencies of LAPACK factorization algorithms

A basis for LAPACK was to cast its DLAFA in terms of level 3 BLAS. In the early 1990s this was an excellent strategy because most processors had only a single cache level, L1. For IBM POWER1 and POWER2 processors, the latency to memory was less than 20 cycles. Additionally, POWER2 featured QUAD load and store operations; it had exceptional memory bandwidth. This was evidenced by the fact that, on the POWER2, ESSL general matrix factorization, Cholesky factorization, and `DGEMM` achieved 90%, 92%, and 96% of the peak rate of the machine for $n = 1000$. In those days operation count and the level 3 BLAS concepts were still reasonable guides for portability performance. Nonetheless, in producing level 3 BLAS that gave smooth high performance, it was still necessary to employ data copy for the L1 kernel routines. Today, with deep memory hierarchies, operation count and the use of BLAS are no longer a reliable performance guide for LAPACK algorithms. In this section, we give three reasons why this is so.

The first reason has to do with LAPACK packed routines. With packed storage one can use only level 1 and packed level 2 BLAS, and performance consequently suffers drastically on RISC systems. A typical reduction from level 3 is about a factor of 3. The BLAS Technical Forum [23] recommended packed level 3 BLAS to correct this inefficiency, while we recommended the adoption of NDS instead of producing a new set of packed level 3 BLAS.[3]

The second reason has to do with the LAPACK choice to make the factorization part of level 3 DLAFA into level 2 computations. The performance inefficiency incurred by this choice can be very great. An example of this occurs in general or QR matrix factorization when the input matrix is tall and narrow, since the entire factorization then becomes level 2. Note that recursion can generally be used to produce level-3-type factorization routines (see Sections 7 and 9).

The third reason was briefly outlined in the Introduction; we continue that description now. A BLAS routine has no knowledge about how it is being used by a calling routine. However, a LAPACK DLAFA of matrix A and its calls to the BLAS are related. Level 3 BLAS were proposed, in part, to support DLAFA, so an implicit relationship exists between level 3 BLAS and their use by LAPACK routines. In fact, every DLAFA calls the level 3 BLAS several times. Each of these multiple BLAS calls has every one of its matrix operands equal to submatrices of the matrix A that the DLAFA is trying to factor. Can one exploit this relationship? An answer lies in examining what the current BLAS do: They try to exploit architectural design characteristics while maintaining the functionality of the BLAS. In a given level 3 BLAS call, its data operands are copied to a new data format so that repeated calls to the L1 kernel routines of the BLAS will all run at or near their potential peak rate. Since the level 3 BLAS are called by DLAFA multiple times, data copying occurs multiple times. However, all of these matrix operands are related because they are all submatrices of the matrix, A, being factored. Now, the manner in which one can exploit this relationship becomes clear: Change the data format of the matrices that are input to the level 3 BLAS routines from standard format to a format conformal to that used by the level 3 BLAS kernel routines and thereby eliminate the need for all repetitive data copying. This new data format is, in fact, the NDS that we are advocating in this paper.

Mathematical dimension theory can be used to shed some light on the subject of this section, and we briefly mention it here. Every object has a unique dimension. The laws of science and engineering relate to two- and three-dimensional objects. Note that linear algebra is repetitively used to solve the problems of engineering and science via computer modeling and simulation. The standard storage layouts for multidimensional arrays are one-dimensional. The dimension, d, of an object is the number of parameters necessary to describe any small, ϵ, neighborhood of the object. A fundamental theorem of dimension theory states that it is impossible to maintain closeness between points in a neighborhood unless the two objects have the same dimension. In the present context we are dealing with matrices, which are two-dimensional objects. However, they are stored in the computer as one-dimensional objects. By applying this theorem, we can see that matrix elements cannot be ordered to be uniformly close together. The standard formats keep either the row or the column dimension close, but the other dimension, column or row, is necessarily far apart. A space-filling curve helps to ameliorate this 1D to 2D problem because one can order 2D blocks in a recursive (divide and conquer) manner. In [10, 11] we do just that and claim that we have a heuristic for automatically blocking all levels of the memory hierarchy. Now return to the

[3] F. G. Gustavson, notes on blocked packed format and square blocked format for symmetric/triangular arrays, September 1999.

```
        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.

individual submatrices of our NDS. Even though these contiguous submatrices are stored in the same standard one-dimensional manner, these two-dimensional submatrices fit nicely into L1 cache. Any data that remains in the L1 behaves as if it were in a random-access memory, and the floating-point execution proceeds at the same high rate as long as all floating-point operations are independent. Fortunately, for DLA algorithms and matrix multiplication in particular, this is the case.

Correcting the LAPACK program inefficiencies

One should use BHF versions of the NDS so that any new DLA code produced will be a backward-compatible replacement for its corresponding LINPACK and/or LAPACK code. We write a DLA code as the block (submatrix) counterpart of its standard code. This is easily done. Now we cover the three inefficiencies of the preceding section for this code. First, the NDS for packed arrays are in a format used by the level 3 L1 BLAS kernel routines. Hence, the new code will consist entirely of a series of level 3 L1 kernel routine calls, and its performance will be better than that of a conventional packed LAPACK code, since these packed codes cannot use level 3 BLAS. Second, level 3 factor kernels are building blocks for producing level 3 implementations of LAPACK TF2 routines. As such, their use produces codes that almost always outperform any LAPACK level 2 implementation. Third, LAPACK full-format routine `DPOTRF` requires repetitive data copying when it calls level 3 BLAS. Using the NDS avoids all data copying.

To demonstrate our approach in more detail, the following sections contain some text and figures originally presented at the Working Conference on the Architecture of Scientific Software, October 2–4, 2000, in Ottawa, Canada, and published in the conference proceedings [24]. This material is reproduced by permission of the publisher.

3. Square blocked full and hybrid formats for general matrices

These new data formats are best described by an example. Let A be an `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 the storage position of $a(i, j)$ in A. Note that locations $(12i, i = 1, 10)$ are indicated by an asterisk. These locations of array `A` are required storage. However, they are neither used nor needed. Nonetheless, a library writer is not allowed to alter their contents. In the rest of Section 3 and Sections 4 and 5, an asterisk will have the same meaning.

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 `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 directly below A in the array holding A. For example, in Figure 1 the minimum storage for A is `LDA` $* n = 120$ doublewords. If the user supplied $ns \geq 144$ elements (extra storage for $\geq$ `NB`$^2 *$ `m1` $*$ `n1` $-$ `LDA` $*$ `n` $= 24$ doublewords 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.

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 blocked version, refer to **Figure 3**. 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 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 3). They are a factor panel kernel of size m by n where $n \leq$ NB called RGETF3, a DTRSM kernel called DLLNU4, and a DGEMM kernel called DAB4. We mention that the factor kernel has the same functionality as the LAPACK DGETF2 routine. However, RGETF3 is a level 3 routine, done recursively, as indicated by the suffix 3 and the prefix R. 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.

We now turn to the way in which 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 ≥ n*LDA, and it is combined with the 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 matrix is rearranged into square blocked 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 $m1*n1*NB^2$.

Square blocked full hybrid data formats for matrices

Let A be m by n, where LDA ≥ m; i.e., A is stored in column-major order. Assume $m \geq n$ (a similar result holds for $n > m$). Let $n1 = \lceil n/NB \rceil$ and $n2 = n + NB - n1 * NB$. 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 represents A as a set of $n1 - 1$ rectangles of size $m * NB$ 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 $m1 = \lceil m/NB \rceil$ and $m2 = m + NB - m1 * NB$. Matrix A is an $m1$ 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 LDA − m rows are stored last. For the original A we assume LDA ≈ m. In **Figure 4** we depict the matrix, A, that is associated with Figure 1.

When LDA ≫ m, our algorithms will suffer because the useful information in the array A, representing matrix A, is mn, while the unused (unnecessary) information is (LDA − m)n. Performing the in-place data transformation requires O(LDAn) operations, so the ratio of overhead to useful work is LDA/m − 1. For example, if we set LDA = 1100 in Figure 4, 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 a high-performance replacement for LAPACK routines. Further, if LDA ≫ m, we can still apply the original algorithm. Finally, we mention that a buffer of size $M \times$ 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, since many LAPACK and BLAS algorithms routinely use extra space of this size.

A matching BLAS 3 LU = PA algorithm

It is almost as easy to code this algorithm as the DGEFB algorithm of Figure 3. However, because of lack of space, we do not include the code for this format.

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 or C arrays, one can program for these formats. In Figure 3, we use a three-dimensional array; we handle the block (I, J) addressing implicitly as 1D addressing in the third dimension. See Section 7 for a discussion regarding the handling of blocks that are stored recursively. The main programming difficulties arise in coding the factor kernel

```
      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 3

Subroutine DGEFB.

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, we predict that descriptors will eventually be used.

Performance for LU = PA

We have examined 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 POWER3 processor with a peak computational rate of 800 MFLOPS. The peak rate is four times the cycle time; there are two FPUs, and each cycle produces two FLOPS from each FPU. Results are for full hybrid block format. The performance timings include the cost of data transformation from standard format to BHF. 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. **Figure 5** shows two plots in which the block LINPACK algorithm is compared 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 [Figure 5(a)] 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 `DGEMM` code. Figure 5(b) shows tall, thin matrices; i.e., n is held fixed at size 100. Here, the new code is nearly twice as fast as `DGETRF` for all m values. Note that `DGETRF` has suboptimal performance because a large part of its factorization time is spent in level 2 `DGETF2` (see the subsection on performance inefficiencies of LAPACK factorization algorithms).

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 storage of the full-array cases. In packed format, the elements of a triangular matrix would be stored as shown in **Figure 6**; the number in location (i, j) is the storage position of $a(i, j)$ 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 `uplo` = `'U'` or `'L'`. For `uplo` = `'L'`, the first vertical stripe is n by `NB` and it consists of $n1$ square blocks where

```
          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 4

Full hybrid block column-major storage order.

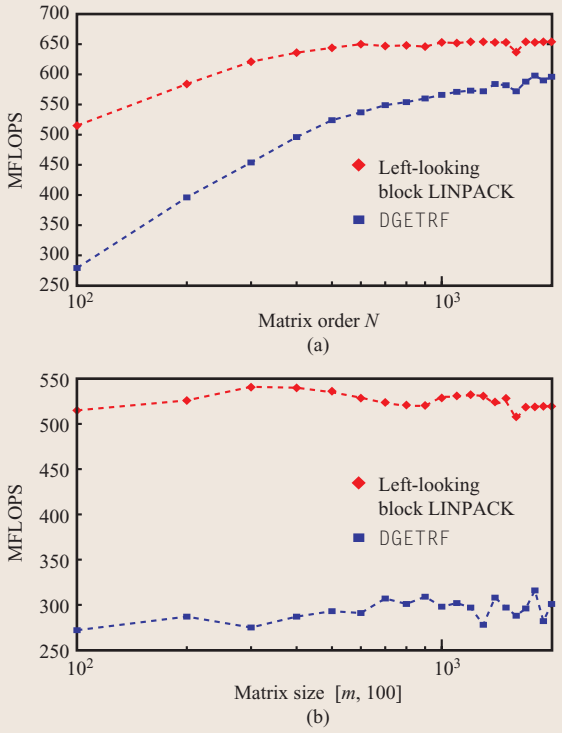

Figure 5

Performance for LU = PA: (a) $m = n$; (b) $100 < m < 2000$, $n = 100$.

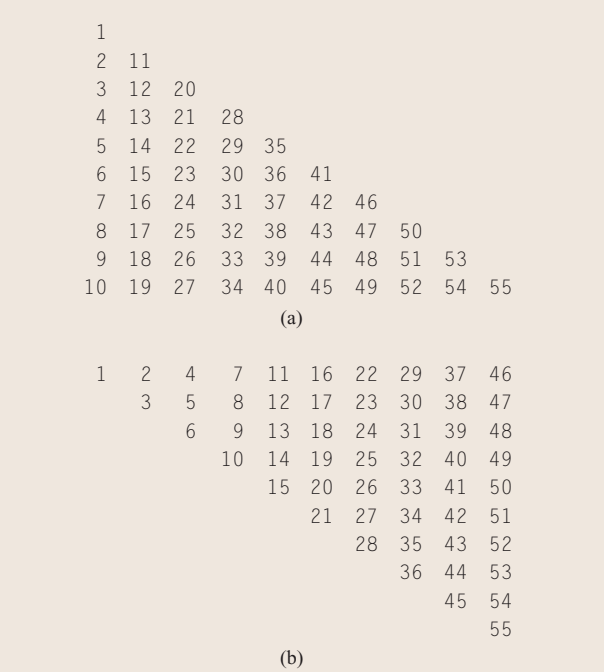

Figure 6

Packed-format arrays: (a) Lower packed; (b) upper packed.

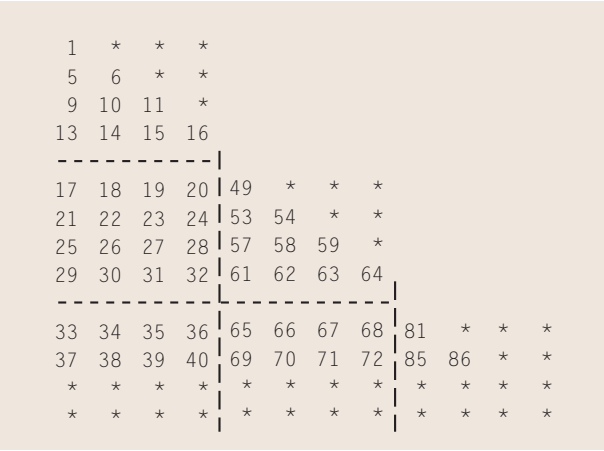

Figure 7

Square blocked lower packed format.

$n1 = \lceil n/\mathsf{NB} \rceil$. 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 $n - \mathsf{NB}$ 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 (SBLP) format with TRANS = 'T' is given in **Figure 7**. 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 six 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 8**. The square blocked upper packed array consists of six square block arrays. The first block holds a 4 by 4 submatrix. The next two blocks 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, denoted 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, one can move the packed array downward into the blocked packed array by starting at the end of AP and moving the square blocks in a block column into a buffer of size $n1{*}\mathsf{NB}^2$ either in row-major or column-major order. The entire buffer can then be copied back over the vacated block column.

The 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 with 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, "standard packed" addressing is used so that the library writer/user can handle his own addressing in a Fortran or C environment.

We now 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 blocked packed format gives a single block triangle which happens to be full format (see **Figure 9**).

In Figure 9 we ignore the last $\mathsf{LDA} - n$ columns of the square blocked array. Here is an interesting observation.

The unused storage, of size $n * (\texttt{LDA} + \texttt{LDA} - n - 1)/2$, consists of n fragmented vectors of sizes $\texttt{LDA} - n : \texttt{LDA} - 1 : 1$. As before, colon notation is used [13]. 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). If this fragmented storage is *not* used, one can convert full format to square blocked packed format, thereby freeing up a contiguous block of storage.

A Cholesky factorization algorithm for square blocked packed format

We now turn to programming dense linear algebra algorithms in the new formats. As an example, **Figure 10** gives code for DPSTRF (PS stands for positive definite symmetric) where the data is stored in the lower triangular part of the full array (`uplo = 'L'`). This 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 can 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 an ESSL factor kernel. The corresponding scalar operation is square root. In ESSL [12] all level 3 BLAS and factorization routines use kernel routines. For example, in ESSL 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 2(a) of the Introduction.] We remark that data copying is sometimes done by the ESSL DGEMM routine; see the subsection on performance inefficiencies of LAPACK factorization algorithms and p. 739 of [10] for more details. 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 4) is being used. (Calling sequences for these four routines were given in Section 2.) 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 zeros and up to three diagonals with ones). Thus, the code works for any matrix order, and the kernel routines, which are programmed in Fortran, are short and of very high

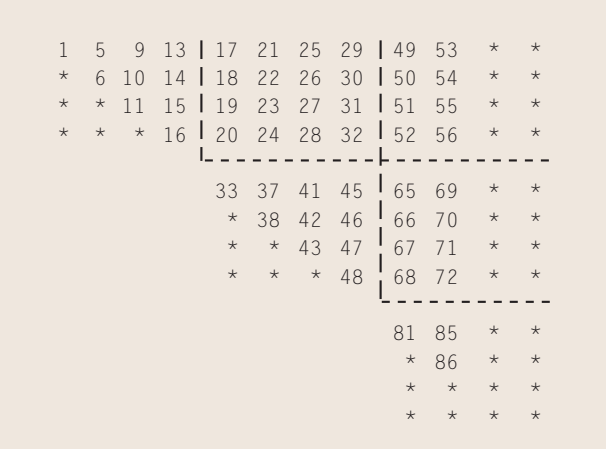

Figure 8

Square blocked upper packed format.

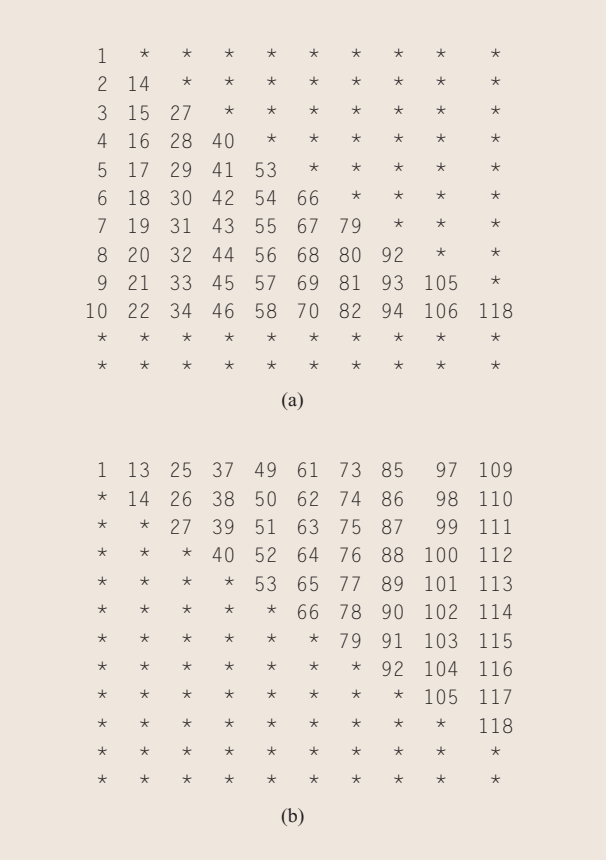

Figure 9

Square blocked packed formats when `NB` = n: (a) `uplo = 'L'`; (b) `uplo = 'U'`.

```
!       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: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 10

DPSTRF subroutine.

performance. Note that routine DPSTRF is just one example of the general schema presented in Points 7 and 8 of the Introduction.

Before closing this section, we wish to suggest a way to package this data storage in LAPACK. We continue with the current routine for Cholesky factorization, uplo = 'L'; a similar procedure would be followed for uplo = 'U'. 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 need no description. The new input parameter, 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 the 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 3 performance.

An example that describes the algorithm DPSTRF

Let NB = 88. This is a good block size for the IBM POWER3 L1 cache, which holds 8192 double-precision words. The DGEMM kernel, DATB4, whose functionality was described in Section 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 doublewords. On most 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. All three matrices, A, B, and C, are not required to 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 doublewords for POWER3. Note that Automatically Tuned Linear Algebra Software (ATLAS) uses a very similar L1 cache-blocking strategy (p. 18 of [8]).

Let $N = 250$. Routine DPSTRF requires N to be a multiple of 4. Thus, N will become 252 by padding two extra rows and columns of an identity matrix to A. A conversion routine called DLPTUP [lower packed to upper (square block) padded] converts a lower packed format array to square blocked lower format with padding of up to three extra rows and columns to make N a multiple of 4. The size of the block array will be 3, so A will look like Figure 7, where now NB=88 and n2=76. Note that rows 75 and 76 of blocks 3 and 5 will be padded with zeros, and rows 77 to 88 of these blocks will consist of unused space. Block 6 will contain a lower triangular matrix of order 76 stored by row, where the last two rows are rows of an identity matrix.

Note that we store our submatrices in SBLP format with the TRANS parameter = 'T'. This is because we emulate factoring A in full upper format, since we want all of our calls in the four kernel routines to do their dot products stride one. A proof that this is correct is given on pp. 826 and 828 of [25] for recursive packed format. 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 NB2 = NB^2. These subarrays reside in memory locations 1 + NB^{2}i, $0 \leq i \leq 6 = 1$, 7775, 15489, 23233, 30977, and 38721. Again see Figure 7 and adjust the starting positions of the blocks by 88 ∗ 88 = 7744 instead of 4 ∗ 4 = 16. In what follows we identify each of the six submatrices by their starting position in the global array that holds them. Now, according to the subsection on the linear transformation approach for producing DLA algorithms, there are three calls to DPOFU4, three calls each to DSLVL4 and DTATA4, and a single call to DATB4. Algorithm DPSTRF is a right-looking algorithm [14]. 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 are three passes through the outer block j loop, sufficient 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 are 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 are 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 2(a) of the Introduction.

Performance and overhead of the example

The purpose of this section is to quantify how Algorithm DPSTRF spends its execution time. The results provide a verification of the subsection on the linear transformation approach for producing DLA algorithms and also show very small overhead. We first state the performance in

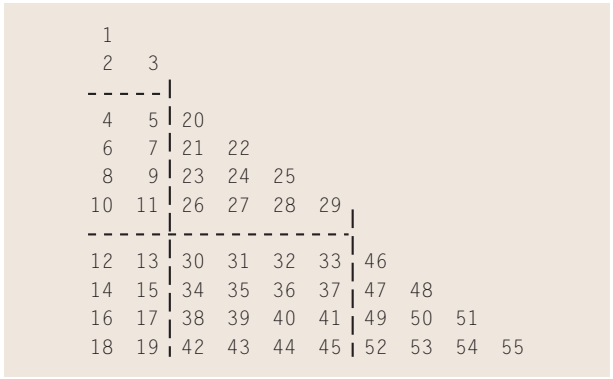

Figure 11

Blocked hybrid lower packed format.

MFLOPS of the ten subroutine calls (in the subsection on an example describing DPSTRF) in the order of the calls. They are u = 609, 715, 711, 749, 775, 750, 609, 711, 750, and 582. The peak MFLOP rate is 800, and all measurements made here were by way of the wall clock. The DGEMM and DSYRK rates of 775 and 750 are very good. 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 equals only one, not two, FLOPS. Also, DPOFU performs divides and square roots. On the POWER3, 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 v = 231132, 681472, 588544, 689216, 1117088, 514976, 231132, 588544, 514976, and 149302, for a total of 5366382 FLOPS. This is the Cholesky FLOP count for an order N = 252 matrix. (See the subsection on the linear transformation approach for producing DLA algorithms, where the various FLOP-count formulas are given.) The FLOP count for N = 250 is 5239875. The execution time of DPSTRF was 0.007518 seconds. The MFLOPS rate is 697. However, we really did an N = 252 size problem. The MFLOPS rate for that problem is 714. Now we compute the MFLOPS rate assuming no overhead for DPSTRF. We take the dot product of the two size-ten vectors $u \cdot v$ to get 3847004572. We then divide by 5366382 FLOPs to get the no-overhead MFLOPS 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%.

Block hybrid formats of symmetric/triangular arrays

These new formats are a combination of traditional packed and full arrays. They retain the main benefit of the new formats presented in the previous section: They allow for level 3 performance while using exactly the same storage as the packed routines. Since no extra storage is required, these routines become backward-compatible replacements of the existing LAPACK packed routines. Their parameters are NB and TRANS. In block hybrid format (BHF), we first choose a block size NB and then lay out the data in trapezoidal swaths. For uplo = both 'U' and 'L', there is also the parameter TRANS. For an example, see **Figure 11**, in which 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 $n1 = \lceil n/\mathrm{NB} \rceil$ and $n2 = n + \mathrm{NB} - n1 * \mathrm{NB}$. The remaining $n1 - 1$ trapezoidal swaths have full base width of size NB. Now each trapezoid with base h and sides (b, $b - h + 1$) 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") is 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 the traditional lower packed format; in the latter, the traditional upper packed format. Now turn again to Figure 11. 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 in row-major order (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 4. The last trapezoid consists of an upper packed triangle of order 4 and a rectangle consisting of no squares.

A matching BLAS 3 Cholesky factorization 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 describe only 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 [25] for details).

There are stronger reasons to choose the hybrid algorithm. First, we choose it because each block triangle is updated, factored, and does all of its scalings in the outer j loop of our block hybrid Cholesky (BHC) algorithm, which we now describe briefly.[4] There are $n1 = \lceil n/\mathtt{NB} \rceil$ passes through the outer j loop. To be able to use only 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 − middle 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 are two and one rectangles, respectively, 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 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 jth 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. The overhead of calling the kernel routines is a very minor cost. First, the kernel routines require no error checking, as they are internal routines. 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 why the overhead is tiny. Assume that $n = 1000$ and NB = 100, which is reasonable on IBM POWER3 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 factorization when $n = 1000$. Clearly, the calling overhead is tiny, and so the overhead cost of BHC is the cost of copying packed triangles to full triangles and back (a total of 50500 matrix elements). Of course, we have not included the cost of Point 7 from the Introduction. This cost consists of moving, in-place, $n(n + 1)/2 = 500\,500$ matrix elements.

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

Programming notes for square blocked packed formats

As in 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; for example, see Figure 10, in which AP is dimensioned AP(*).

Performance for Cholesky factorization

Figure 12 shows performance (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 and DPOTRF, we do not include the cost of transforming the data format. This is perhaps unfair; nonetheless, we did it to demonstrate the limits of possible performance. Note that DPSTRF shows some choppy behavior, especially when n is small. The matrix orders where this occurs are not multiples of 4. For example, when $n = 70$, the performance is about the same as when $n = 60$. 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

[4] For full details, see B. S. Andersen, J. A. Gunnels, F. G. Gustavson, J. K. Reid, and J. Wasniewski, "A Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm"; to be submitted to *ACM Transactions on Mathematical Software*.

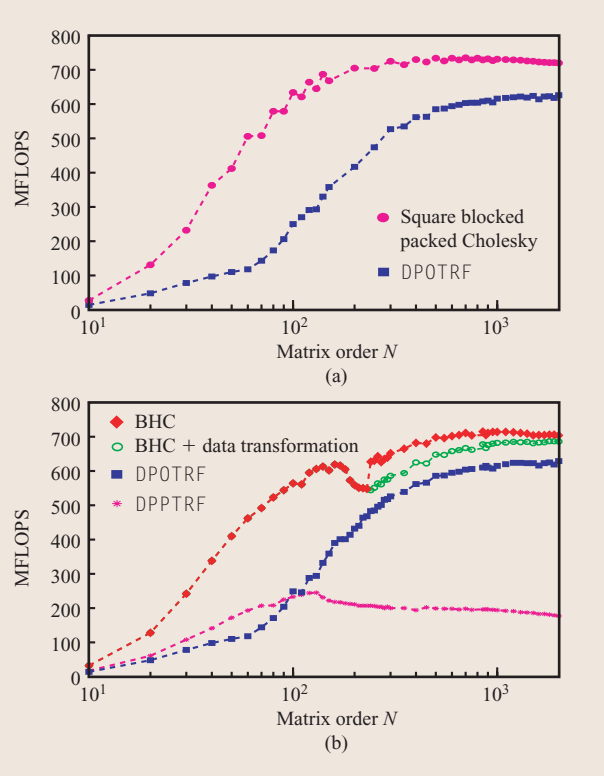

Figure 12

Performance for Cholesky factorization as a function of matrix order: (a) Square blocked packed Cholesky and DPOTRF; (b) BHC, BHC and data transformation, DPOTRF, and DPPTRF.

DPSTRF is always faster than DPOTRF by as much as a factor of 4 when $n = 60$ and at least 15% for $n = 2000$.

In the comparison of BHC and LAPACK shown in Figure 12(b), there are four curves: 1) BHC, 2) BHC + data transformation, 3) DPOTRF, and 4) DPPTRF. 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 POWER3 holds 8192 doublewords and that this factor kernel peaks at 620 MFLOPS for $n > 160$. In this subsection we have shown that $n^2/4$ is the effective cache size, and thus $n \approx 180$. Note that the initial data transformation does cost something. The actual crossover happened at $n = 230$. For $n \leq 230$, curves 1 and 2 are identical. For $n > 230$, it pays to use the data transformation, and 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 [13, §4.4] and [26, §10.4.2] for the LDL^T decomposition of a matrix, A. There are two types of subroutines in LAPACK which use this method. In the first, the matrix is stored in a two-dimensional array (Figure 9). As previously mentioned, conserving memory is sometimes an important issue. Clearly, full storage uses almost twice the necessary memory. Packed storage (Figure 6) is the second type of storage scheme. With this storage format, a one-dimensional array is used to store only the essential part of the matrix.

We introduce lower packed blocked (LPB) storage as another 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 use of level 3 BLAS. Of course, this format requires slightly more memory than 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 the LAPACK DSPSV and the superior performance of the LAPACK DSYSV [27].

This new storage layout is another 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.

A generalization of the traditional symmetric formats

Let $1 \leq \mathtt{NB} \leq N$, where NB is the block size. Let $\mathtt{N1} = \lceil N/\mathtt{NB} \rceil$ and $\mathtt{N2} = \mathtt{N} + \mathtt{NB} - \mathtt{N1} * \mathtt{NB}$. The new LPB format partitions the N columns of A into (N1 − 1) block columns, each consisting of NB successive columns of A, and a last block consisting of N2 columns. The LDA of block i is $\mathtt{LDA} - (i - 1)\mathtt{NB}$. An example of LPB format with $\mathtt{NB} = 4$ is given in **Figure 13(a)**. Note that when $\mathtt{NB} = N$, one gets full format [see Figure 9(a)], whereas when $\mathtt{NB} = 1$ and $\mathtt{LDA} = N$, one gets lower packed format. Hence, the LPB format generalizes both traditional formats.

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 a block diagonal consisting of 1×1 or 2×2 blocks. More detailed descriptions can be found in [13, §4.4] and [26, §10.4.2]. Our algorithm produces results identical to those of the LAPACK _SYTRF routine.

A modification of LPB format called lower packed blocked 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 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 13(b)**], we introduce, at the end of the first `N1−1` block columns, an extra `NB` storage locations as padding.

We now illustrate how this padding allows us to efficiently handle a block factorization and its level 3 update. The problem arises when we factor column j and j is a multiple of `NB`. If column j starts a 1×1 pivot, there will be `NB` or `NB−1` columns in the factor block. If column j starts a 2×2 pivot, 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 = `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 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 of the above block computations are laid out in full storage format. Therefore, level 3 BLAS can be applied. Also, in both cases above, we 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.

```
 1   *   *   *
 2  14   *   *
 3  15  27   *
 4  16  28  40
 5  17  29  41  49   *   *   *
 6  18  30  42  50  58   *   *
 7  19  31  43  51  59  67   *
 8  20  32  44  52  60  68  76
 9  21  33  45  53  61  69  77  81   *
10  22  34  46  54  62  70  78  82  86
 *   *   *   *   *   *   *   *   *   *
 *   *   *   *   *   *   *   *   *   *
```

(a)

```
 1   #   *   *   *
 2  14   #   *   *
 3  15  27   #   *
 4  16  28  40   #
 5  17  29  41  53   #   *   *   *
 6  18  30  42  54  62   #   *   *
 7  19  31  43  55  63  71   #   *
 8  20  32  44  56  64  72  80   #
 9  21  33  45  57  65  73  81  89   #
10  22  34  46  58  66  74  82  90  94
 *   *   *   *   *   *   *   *   *   *
 *   *   *   *   *   *   *   *   *   *
```

(b)

Figure 13

Lower packed and lower packed block overlapped format when `NB` = 4: (a) LPB format; (b) 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 the LAPACK _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.

New LAPACK algorithms for symmetric indefinite factorization in ESSL

We have packaged our algorithms in ESSL. We now discuss the issues of exact replacement and migration

of these routines for LAPACK 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 the first four arguments of DSPTRF. The fifth and last argument of DSPTRF 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.

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 the 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, 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 called only 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.

Migration and portability considerations

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 must 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.

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 are briefly described in the five subsections which follow.

Intermediate back pivoting

A difference between LINPACK and LAPACK is that LAPACK usually uses full back pivoting, whereas LINPACK does not use 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.

LPBO format allows for a level 3 multiple solve

We remark that the LAPACK 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 # symbols in Figure 13(b) that illustrate LPBO format). Thus, we represent D (UPLO = 'L') as

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

In any case, the LAPACK full DSYTRS code uses LINPACK-type pivoting and hence is not a level 3 implementation.

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 [8]) is that (during DGEMM) the left operand matrix (W in this case) fills the L1 cache. If LW^T were used, L would fill the L1 cache. Now, the size of W is under the control of the algorithm implementor, since he sets its LDW, whereas L is an input array. Thus, WL^T is to be preferred over LW^T.

The workspace size of W can be NB² and not N * NB

During the factor part of DBSTRF, we need only 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 NB^2. This means that the benefit mentioned in the preceding subsection 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 need only 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 the storage of A.

The UPLO = 'U' case

Using a stride of 1 gives the best performance. The UPLO = 'L' case uses stride 1 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. 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).

Performance

We tested our algorithms on three recent IBM platforms: POWER3, POWER2, and PowerPC 604e. In each of the following figures we compare our new BS routines to the LAPACK SY routines. All plots measure MFLOPS versus matrix order, which runs from 10 to 1000 in steps of 10.

In **Figure 14(a)** we consider factorization, performed on a 200-MHz POWER3. Initially both programs have identical performance because the algorithms are identical. There is a crossover point at which DBSTRF switches from packed to LPBO format. The first big drop (caused by program loading) in Figure 14(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 the DSYTRF performance (see the subsection on the relative efficiencies of WL^T and LW^T for an explanation). In **Figure 14(b)**, we compare DBSTRS to DSYTRS for multiple and single right-hand sides. Here we see the dramatic improvement of using intermediate block factorization format over using LINPACK factorization format. **Figure 15(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

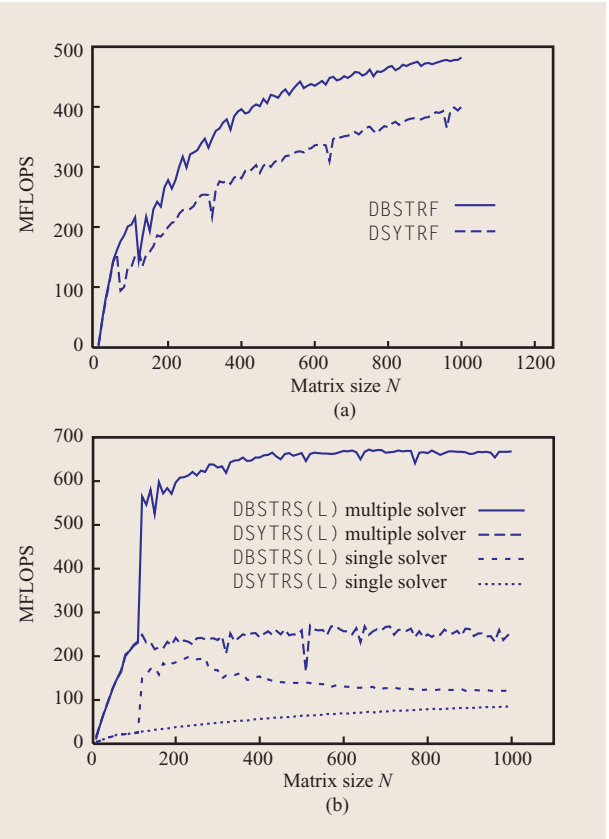

Figure 14

Performance on IBM POWER3: (a) Factorization; (b) multiple and single solver.

a good data mapping into the four-way set-associative L1 cache. Note that DBSTRF does not have these dips because we set LDW ≈ LDA so as to produce a good L1 data mapping. In **Figure 15(b)**, we did not choose LDB = N, because doing so might have produced similar dips. **Figure 16(a)** shows factorization performance on a 332-MHz PowerPC 604e machine. Compared to POWER3 and POWER2, this machine evinces poor floating-point performance. **Figure 16(b)** shows how good level 3 performance can be in comparison to the level 2.5 performance of DSYTRS (about a factor of 5).

6. Vectors

The purpose of this section is to briefly show how vectors could be handled by the NDS and kernel routine combination. 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

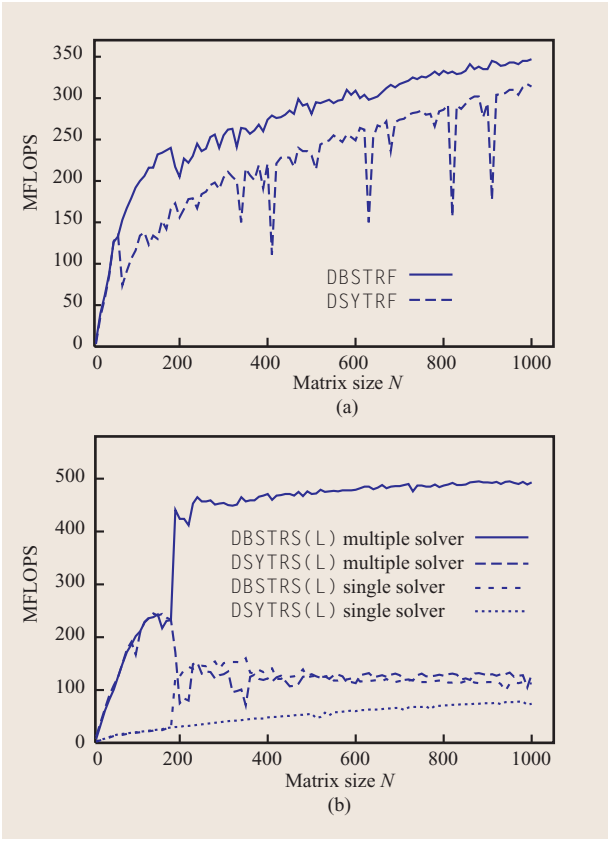

Figure 15

Performance on IBM POWER2: (a) Factorization; (b) multiple and single solver.

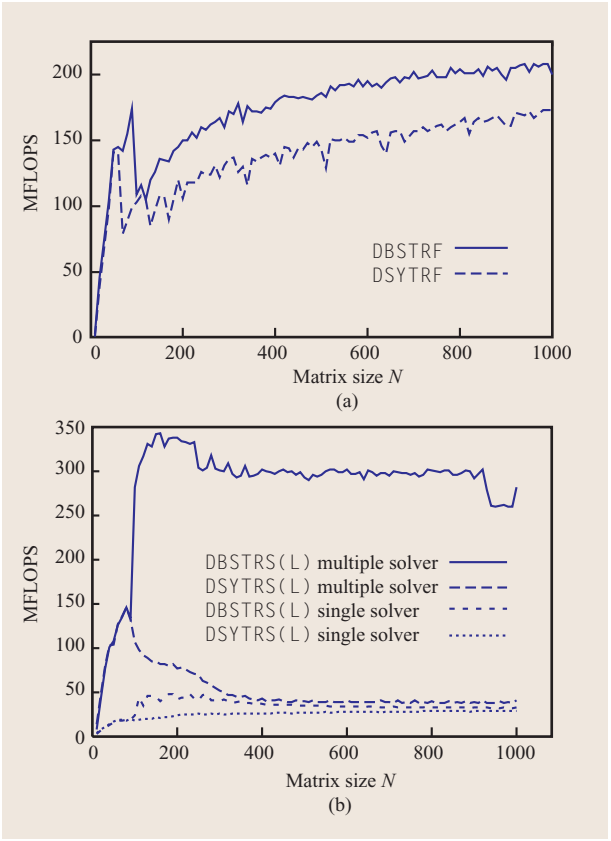

Figure 16

Performance on IBM PowerPC 604e (332 MHz): (a) Factorization; (b) multiple and single solver.

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. Its purpose is to indicate 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 [10, 11, 28–30]. Hence, it provides a heuristic to block for the various higher levels of the memory hierarchy (L2, L3, main memory, disk memory). 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 of integer tables. Since each of the blocks contains 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 [14], 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 is removed when one adopts the NDS as an input/output format. A new set of BLAS, namely factor kernels, must be defined. However, these new BLAS become simpler to write, since 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. Using the recursion heuristic of Section 7, it is possible to write platform-independent BLAS. Nonetheless, the L1 kernels are platform-dependent.

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, where two different algorithms, DPSTRF and the current LAPACK blocking algorithm, are given for Cholesky factorization. These algorithms are differentiated by the manner in which they block the operands. Hence, making a choice between them, we could produce a "Cholesky BLAS" algorithm. This is the approach used in producing algorithms for the ESSL library [12].

9. Kernel routines

A kernel routine for a level 3 BLAS or for a factorization routine is the 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 textbooks. 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 4. 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 3 and 10 are the suffix-4 codes briefly described above. Currently, they are being implemented by hand. However, in principle, the kernel routines can be produced automatically by a compiler and/or preprocessor for IBM POWER3 processors. Further details are given in [6], pp. 569 and 570. Note, however, that we now use 4 by 4 unrolling instead of the 4 by 2 unrolling in [6]. The other kernel routines, DLLNU4 in Figure 3 and DPOFU4, DSLVL4, and DTATA4 in Figure 10, 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 the subsection on operation counts as a measure of kernel routine performance, almost all of the operations performed by these four routines are FMAs. And by the principle of linear superposition we can rearrange the computations in all of these routines by unrolling their two outer loops by 4. Thus, for the most part, the inner loop consists of the same computation of 16 independent dot-products that was described above. In summary, we have further demonstrated the pervasiveness of matrix multiplication in dense linear algebra algorithms. Only the factor kernel routine, RGETR3, of Section 3 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

This paper has described several novel data formats for dense linear algebra and some simple, novel algorithms that utilize these new data structures. The paper relies 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 simple fact from linear algebra: Some point algorithms have a submatrix block formulation. Using the principle of Equivalence, we have shown that a combination of NDS and kernel routines is general and can be applied to a whole class of DLA algorithms. This result is true for Cholesky and symmetric indefinite factorization and for LU factorization with partial pivoting. For all three DLAFA algorithmic examples, we have presented performance results experimentally verifying that the current SOA LAPACK algorithms have performance inefficiencies. The combination of NDS and kernel routines overcomes all of these performance inefficiencies. We have presented performance results for these same three DLAFA, using NDS and kernel routines, demonstrating that they usually perform much better than their LAPACK counterparts.

Acknowledgments

The work described here is partially the outcome of collaboration with researchers at HPCN at the University of Umea, Sweden, and at Uni-C in Lyngby, Denmark. They are Erik Elmroth, Andre Henriksson, Isak Jonsson, Bo Kagstrom, Per Ling Bjarne-Stig Andersen, Alex Karaivanov, Minka Marinova, Jerzy Wasniewski, and Plamen Yalamov. I also thank Carl Tengwall of IBM Sweden and John Gunnels of IBM Research.

References

1. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Software* **5,** No. 3, 308–323 (September 1979).
2. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **14,** No. 1, 1–17 (March 1988).
3. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **16,** No. 1, 1–17 (March 1990).

4. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide Release 2.0*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.
5. 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*, Society for Industrial and Applied Mathematics, Philadelphia, 1999; see *http://www.netlib.org/lapack/lug/lapack_lug.html*.
6. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms," *IBM J. Res. & Dev.* **38,** No. 5, 563–576 (September 1994).
7. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable High-Performance ANSI C Coding Methodology," *Proceedings of the International Conference on Supercomputing*, Vienna, 1997, pp. 340–347.
8. R. C. Whaley, Antoine Petitet, and Jack J. Dongarra, "Automated Empirical Optimization Software and the ATLAS Project," *Parallel Computing* **27,** No. 1–2, 3–35 (January 2001).
9. J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, "A Family of High-Performance Matrix Multiplication Algorithms," *Computational Science—ICCS 2001*, *Part I*, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renna, and C. K. Tan, Eds., *Lecture Notes in Computer Science*, No. 2073, 2001, pp. 51–60.
10. F. G. Gustavson, "Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms," *IBM J. Res. & Dev.* **41,** No. 6, 737–755 (November 1997).
11. F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kagstrom, and P. Ling, "Recursive Blocked Data Formats and BLAS Dense Linear Algebra Algorithms," *Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, B. Kagstrom et al., Eds., *Lecture Notes in Computer Science*, No. 1541, 1998, pp. 195–206.
12. IBM Corporation, *Engineering and Scientific Subroutine Library for AIX Version 3, Release 3*; *Order No. SA22-7272-04*, December 2001.
13. G. Golub and C. VanLoan, *Matrix Computations*, Johns Hopkins Press, Baltimore, 1996.
14. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26,** No. 1, 91–112 (January 1984).
15. E. Elmroth and F. G. Gustavson, "Applying Recursion to Serial and Parallel *QR* Factorization Leads to Better Performance," *IBM J. Res. & Dev.* **44,** No. 4, 605–624 (July 2000).
16. G. Birkhoff and S. MacLane, *A Survey of Modern Algebra*, Revised Edition, Macmillan Publishing Co., New York, 1953.
17. L. Mirsky, *An Introduction to Linear Algebra*, Revised Edition, Dover Publications, Mineola, L.I., New York, 1972.
18. E. W. Elmroth and F. G. Gustavson, "A Faster and Simpler Recursive Algorithm for LAPACK Routine DGELS," *BIT* **41,** No. 5, 936–949 (2001).
19. R. K. Brayton, F. G. Gustavson, and R. A. Willoughby, "Some Results on Sparse Matrices," *Math. Comp.* **24,** No. 112, 937–954 (October 1970).
20. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. & Dev.* **34,** No. 1, 59–70 (January 1990).
21. C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, 2001.
22. S. Toledo, "A Survey of Out-of-Core Algorithms in Numerical Linear Algebra," *External Memory Algorithms*, J. M. Abello and J. S. Vitter, Eds., *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, 1999, pp. 161–179.
23. BLAS Technical Forum, 1995; see *http://www.netlib.org/blas/blast-forum/*.
24. F. G. Gustavson, "New Generalized Matrix Data Structures Lead to a Variety of High-Performance Algorithms," *The Architecture of Scientific Software*, R. P. Boisvert and P. T. P. Tang, Eds., Kluwer Academic Publishers, Boston, 2001, pp. 211–232.
25. F. G. Gustavson and I. Jonsson, "Minimal Storage High Performance Cholesky via Blocking and Recursion," *IBM J. Res. & Dev.* **44,** No. 6, 823–850 (November 2000).
26. N. J. Highham, *Accuracy and Stability of Numerical Computations*, Society for Industrial and Applied Mathematics, Philadelphia, 1996.
27. F. Gustavson, A. Karaivanov, M. Marinova, J. Wasniewski, and P. Yalamov, "A Fast Minimal Storage Symmetric Indefinite Solver," *Applied Parallel Computing New Paradigms for HPC in Industry and Academia, Fifth International Workshop PARA 2000 Proceedings*, *Lecture Notes in Computer Science*, No. 1947, 2001, pp. 103–112.
28. A. Gupta, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Yalamov, "Experience with a Recursive Perturbation Based Algorithm for Symmetric Indefinite Linear Systems," *Euro Par '99, Parallel Processing; Fifth International Euro Par Conference Proceedings*, *Lecture Notes in Computer Science*, No. 1685, 1999, pp. 1096–1103.
29. B. S. Andersen, F. Gustavson, and J. Wasniewski, "A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage," *ACM Trans. Math. Software* **27,** No. 2, 214–244 (2001).
30. B. S. Andersen, J. A. Gunnels, F. Gustavson, and J. Wasniewski, "A Recursive Formulation of the Inversion of Symmetric Positive Definite Matrices in Packed Storage Data Format," *Applied Parallel Computing New Paradigms for HPC in Industry and Academia, Seventh International Workshop PARA 2002 Proceedings, Lecture Notes in Computer Science*, No. 2367, 2002, pp. 287–296.

Received October 18, 2001; accepted for publication July 11, 2002

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gustav@us.ibm.com).* Dr. Gustavson manages the Algorithms and Architectures group in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. degree in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Invention Achievement Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.