D R A F T
Document for the Basic Linear Algebra Subprograms (BLAS) Standard

Basic Linear Algebra Subprograms Technical (BLAST) Forum

December 6, 1997

Contents

The discussion in each chapter is not limited only to the points listed, any related topics can be included at the discretion of the chapter authors. Final acceptance of each section will be decided at the next meeting using *Robert's Rules*. See BLAST Forum homepage for more detailed rules.

Chapter 5

Extra Precise BLAS (XBLAS)

5.1 Section 1: Introduction

We propose two kinds of extensions to the current BLAS, one that allows mixing of operand matrices of different precisions, and a second kind that allows arithmetic inside the BLAS to be carried out to higher precision than the operands, when doing so confers some advantage. These extensions are motivated by the following facts:

- A number of important linear algebra algorithms can become more accurate and sometimes faster if internal computations carry more range and/or precision than used for the input and output arguments. These include linear system solving, least squares problems, and eigenvalue problems. Often the benefits of wider arithmetic cost only a small fractional addition to the total work.

- For single precision input, the computer's native double precision is a way to achieve these benefits easily on all commercially significant computers, at least when only a few extra-precision operations are needed. Crays, Convex, and their emulators implement 64-bit single in hardware and much slower 128-bit double in software, so if many double precision operations are needed, these machines will slow down significantly.

- The overwhelming majority of computers on desktops, containing Intel processors or their AMD and Cyrix clones, are designed to run fastest performing arithmetic to the full width, wider than double precision, of their internal registers. These computers confer some benefits of wider arithmetic at little or no performance penalty. Some BLAS on these computers already perform wider arithmetic internally but, without knowing this for sure, programmers cannot exploit it.

- All computers can simulate quadruple precision or something like it in software at the cost of arithmetic slower than double precision by at worst an order of magnitude. Less slowdown is incurred for a rough double-double precision on machines (IBM RS/6000, PowerPC/Mac, SGI/MIPS R8000, HP PA RISC 2.0) with special fused multiply-accumulate instructions. Since some algorithms require very little extra precise arithmetic to get a large benefit, the slowdown is practically negligible.

In this proposal we shall explain our motivation in more detail and present detailed specifications for our proposed extra precise BLAS, which we call XBLAS. The greatest challenge to our design is dealing with too many possible combinations of input types (including precisions), output type, and internal precision, not all of which are useful. We must name all useful possibilities, provide reasonable defaults so they are easier to use, make them callable from different languages, and arrange these numerous routines into a smaller number of groups each of which may be automatically generated, tested and even optimized from a few prototypes.

Here are three examples to illustrate the opportunities and challenges.

First, extra-precisely accumulated matrix-vector products provide an inexpensive way to refine solutions of linear systems so well that the error is small independently of the system's condition number, unless the condition number is extremely big. The cost of the extra-precise arithmetic is negligible for moderate sized and larger matrices.

Second, multiplying a complex matrix by a real matrix is so much faster than multiplying two complex matrices that a special LAPACK auxiliary routine CLACRM was invented to do this instead of promoting real to complex and calling CGEMM. This routine is usually the bottleneck in what is currently the fastest Hermitian tridiagonal eigensolver, CSTEDC. Thus allowing a mixture of input types enhances speed. Accuracy can be enhanced too if extra-precise accumulation is allowed.

Third, the algorithm in Example 2.1 below needs extra precision in just a few operations, and could obtain that at negligible cost from the 80-bit extended format on Intel/AMD/Cyrix platforms. For portability, it should permit other platforms to use double-double instead of extended. The extra precision need only be used inside the BLAS, so no user variables need to be declared in the high precision format. On the other hand, if the user's input data is double, the algorithms in Examples 2.4, 2.5 and 2.7 would require running a high level LAPACK subroutine with variables declared to be in the high precision format; not all compilers support such declarations. Given this diversity, what is the right tradeoff between portability and functionality for libraries like LAPACK?

To accommodate all these possibilities, we have chosen a simple naming scheme that permits *all* possible combinations of types and precisions to be specified, even uninteresting ones. We do not expect all combinations to be implemented. The naming scheme would be no simpler if we were to explicitly limit the combinations. Our specification prioritizes the most important routines to be implemented, and leaves open the possibility of implementing others as well. In addition, we specify an environmental inquiry function to determine the properties of extra precise BLAS, and whether they exist at all. (For example, the user might be told that no double-double BLAS exist on an Intel platform.) Optional arguments are used judiciously so that the user can specify as little or as much detail about precisions of input arguments and internal precision as desired.

For example, a call to matrix multiplication could appear as

```
XDGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC,
        PREC)
```

where the leading X indicates that this is an XBLAS routine, and the following D indicates that the precision of the output floating point variable C is double. The only difference between this and the standard BLAS calling sequence is the optional string PREC. If PREC left blank, this would indicate that all input variables are also double, and that the widest indigenous floating point format should be used internally (extended on Pentiums, and double on most other machines). If PREC='E', it indicates that the input variables are also double, and that a format wider than double should be used; on Pentiums this would be extended, and on other machines probably double-double. If PREC='ES' it indicates in addition that all the input variables are in fact single precision.

If it is simpler or cheaper to carry more precision internally than the user requested, the BLAS implementor is free to do so provided the environmental inquiry routine returns the actual precision used.

In this spirit we invite implementors of the standard BLAS library to carry extra precision internally whenever this is insignificantly slower than not carrying extra precision, since this is permitted by the BLAS specification now and usually gives users better results. We also propose that our environmental inquiry report the precision actually used in the standard BLAS, along with other information like the use of Strassen's matrix multiply. Left unsolved is the problem of obtaining from one platform exactly the same result as was generated by another, since this problem is already intractable in the face of BLAS optimized for speed on platforms with diverse memory management, opportunities for concurrency (pipelines), and heterogeneous collections of processors.

Though we concentrate on extending the dense BLAS to accommodate extra precision, we believe our general scheme will accommodate other extensions, for example to sparse or parallel BLAS, to interval BLAS, to other routines adopted by the BLAST Forum, and to other routines we discuss below.

The rest of this proposal is organized as follows.

Section 2 lists compelling examples to motivate our proposal. Most involve extra-precise or mixed-type versions of existing BLAS, and some involve new routines.

Section 3 surveys implementation technicalities for the higher-than-double precisions supported by current hardware.

Section 4 describes the naming scheme and F77 calling sequences, including optional arguments and defaults.

Section 5 describes the environmental inquiries used to describe the high-precision BLAS, including the accuracy model they must satisfy.

Section 6 presents the prioritized list of extra precision BLAS that we propose all developers implement.

Section 7 will eventually describe how one can automatically generate these routines.

Section 8 will eventually describe how one can automatically test these routines.

Section 9 will eventually give sample implementations of some basic routines.

Section 10 will eventually give sample applications showing how useful these routines are. It currently just discusses how to pick applications, and what we should expect from compiler writers.

Section 11 compares this to a previous proposal, which was part of the BLAS 2 specification [25].

5.2 Section 2: Example Applications

We present a variety of examples that illustrate different advantages of our proposal, including improved accuracy, improved speed, and improved reliability. Not all features of our proposal are used by all examples:

Examples 2.1, 2.2, 2.5, 2.6, 2.7 and 2.9 require higher precision.

Examples 2.3 and 2.4 require wider range. They can also take additional advantage of IEEE exception handling internally, but do not require it.

Examples 2.7 and 2.8 require mixed-type versions of existing BLAS; 2.8 requires no extra precision or range.

Most examples have well-understood benefits, but examples 2.6, 2.9 and the last part of example 2.2 are research questions.

Most examples use mixed or higher precision versions of existing BLAS, but examples 2.4, 2.5 and 2.7 would require new BLAS (actually the promotion of certain LAPACK or ScaLAPACK routines to BLAS status). They are easy to implement when the input precision is single and the extra precision is double. We include them to illustrate the broader utility of mixed-precision computations.

5.2.1 Example 2.1: Iterative Refinement of Linear Systems

Given an LU or other stable factorization of a matrix A, we can solve $Ax = b$ by forward and back substitution. The relative error of this algorithm is bounded by

$$\frac{\|x_{computed} - x_{true}\|}{\|x_{true}\|} \leq O(N \cdot cond(A) \cdot macheps)$$

where $cond(A) \geq 1$ is the condition number of A and N its dimension, and $macheps$ is the machine epsilon for the input and output data, which we call input/output precision. This error bound is not excessively pessimistic; errors about that big are encountered occasionally.

The accuracy of the computed solution x may be improved by iterative refinement roughly as follows:

repeat
    Compute the residual $r = A \cdot x - b$
    Solve $A \cdot d = r$ for d using the existing factorization
    Update the solution $x = x - d$
until {the residual is small enough or stops decreasing, or
    a maximum iteration count is exceeded}.

LAPACK currently implements this algorithm entirely in the input/output precision, in routine xGERFS. What it accomplishes, according to a well understood error-analysis, is essentially to replace the condition number $cond(A)$ in the error bound by another possibly smaller one, which can be as small as $min_D cond(D \cdot A)$, where the minimum is over all diagonal matrices D. In effect, this algorithm compensates, up to a point, for bad row-scaling The residual is never worsened but the solution x, though usually improved, frequently gets worse if this last condition number is very big.

An improved version of iterative refinement differs from LAPACK's in two places. First, the residual $r = A \cdot x - b$ is accumulated to at least twice the input/output precision and then, after massive cancellation has taken place, is rounded to input/output precision. Second, refinement is stopped after x appears to have settled down about as much as it ever will. Now another classical error analysis says that the ultimate relative error will be bounded by $O(macheps)$, essentially independent of the condition number and dimension of A. This assumes that $N \cdot cond(A)$ is not comparable with or bigger than $1/macheps$ (in which case A could be made exactly singular by a few rounding errors in its components). In short, the improved version of iterative refinement almost always produces the correct solution x, and almost always gives fair warning otherwise. This is a much simpler and stronger result than LAPACK provides now.

If the improved algorithm's residual r is accumulated to sufficiently more than input/output precision but less than twice as much, its improvement over what LAPACK obtains currently becomes proportionately weakened. In extensive experiments on machines with 80-bit registers, i.e. 11 more bits of precision than double, iterative refinement either returned the solution correct to all but the last few bits, or with at least 10 more correct bits than without refinement [2].

The current LAPACK algorithm does not have to change much to benefit from a more accurate residual: *macheps* has to be replaced in a few places by a roundoff threshold that reflects the possibly extra-precise accumulation of r; and if this latter threshold is sufficiently smaller than *macheps* then the iteration's stopping criterion should be changed from testing r to testing x and d, and an estimate of the error in x should be obtained not from the current condition estimator but instead (at lower cost) from the last few vectors d.

The indispensable requirement is an extra-precise version of GEMV in the Level 2 BLAS. One iteration of refinement costs only $4 \cdot N^2$ flops per right-hand side column b. If, as usual, there are only a few columns b and at most a few iterations suffice, their cost is asymptotically negligible compared to the $O(N^3)$ operations required to compute the LU or other factorization in the first place, even if extra-precise operations run somewhat slowly. Therefore, we want to insist that GEMV use extra precision, even if it is expensive.

5.2.2 Example 2.2: Iterative refinement of other linear algebra problems

The availability of extra-precise residuals would make it possible to solve many other linear algebra problem more accurately as well. The simplest example is the overdetermined least squares problem: choosing x to minimize $\|A \cdot x - b\|_2$. This can be formulated as the linear system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \cdot \begin{bmatrix} -r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

and refined using the QR decomposition of A from which the initial solution x was obtained. As in Example 2.1, the final error depends much less upon the condition number of A (though the situation is a bit more complicated than a simple linear system). Again, an extra precise GEMV is indispensable, and its cost is asymptotically small provided that A is not too far from square.

The foregoing is most appropriate for double precision data A and b, since it only requires extra precision hidden within BLAS. For single precision data Example 2.7 provides an attractive alternative, where we use more mixed precision.

Iterative refinement for eigenproblems is still a topic for research, but recent recent results are encouraging. The indispensable requirement is a matrix residual

$$R = A \cdot X - Y \cdot B = \begin{bmatrix} A & -Y \end{bmatrix} \cdot \begin{bmatrix} X \\ B \end{bmatrix}$$

accumulated extra-precisely and rounded back to input/output precision after massive cancellation has taken place. Sometimes $Y = X$; sometimes B is diagonal or triangular. Such residuals can be computed from one call to a GEMM that accumulates extra-precisely, but a lot of waste motion can be avoided if versions of GEMM that accept and produce extra-precise matrices are available too. Initially these versions of GEMM could be simulated by calls upon extra-precise versions of GEMV instead, just to get the ball rolling sooner.

Iterative refinement of nonsymmetric eigenproblems, other than Schur decompositions, depends crucially upon how well iterative refinement works for extremely nearly singular linear systems. For best results, the LU factorization should ideally be performed by Crout factorization with extra-precisely accumulated scalar products, but how much good this would do is not yet clear. Exception handling is important too because infinity and 0/0 turn up in the better mathematical algorithms mentioned above unless entirely algebraic computations are replaced in spots by invocations of transcendental functions elementwise upon arrays. The last question remaining is how much precision would be needed to perform an entire eigencalculation by conventional means, without refinement, to obtain results as good and as soon as are obtained from refinement with a little extra-precise arithmetic in the program; experience indicates that a high precision conventional algorithm without refinement is much more expensive than with refinement.

5.2.3 Example 2.3: Solving Ill-conditioned Triangular Systems

The LAPACK subroutine SLATRS solves a triangular linear system $T \cdot x = scale \cdot b$ with careful scaling to prevent over/underflow. It is used in place of the Level 2 BLAS routine STRSV when we expect very ill-conditioned triangular systems, and want to get the solution vector (with a scale factor) despite the fact that it may be very large or very small. This is the case with condition estimation (SGECON, SPOCON, STRCON) and inverse iteration to find eigenvectors of nonsymmetric matrices (SLAEIN). SLATRS does careful scaling in the innermost loops to prevent over/underflow, and is significantly slower than an optimized STRSV. Also, it contains about 320 executable statements, (versus 160 for the unoptimized version of STRSV) a measure of the complexity of producing such code. There are three ways to accelerate SLATRS.

The first way is to use extra exponent range to guarantee that over/underflow cannot occur for a given number of iterations of the outermost loop of (lower) triangular solve:

Solve $L \cdot x = b$ ... L is lower triangular
  for $i = 1$ to n
    $x(i) = [b(i) - (\sum_{j=1}^{i-1} L(i,j) \cdot x(j))]/L(i,i)$

For example, if the entries of L and b are represented in IEEE single, and no diagonal entry of L is denormalized, and indigenous 80-bit Pentium arithmetic is used to implement this algorithm, testing the scaling is needed only once every 64 values of i. In particular, this could be implemented by calling an optimized SGEMV on blocks of size 64, which would get much of the benefit of the optimized SGEMV. Double is not so good, since we could only do blocks of 8. Alternatively, we could use scaling code like that in SLATRS to compute the largest block size we could get away with. This requires information about the norms of columns of L. In the case of SLAEIN, where we call SLATRS many times, the cost of this information is small. In the case of condition estimation, the cost is larger.

The second way to accelerate SLATRS was explored in the paper "Faster Numerical Algorithms via Exception Handling" [20]. The IEEE exception flags can be used to hide over/underflow problems as follows: We first run using the optimized STRSV, and then check the exception flags. Since over/underflow is rare, this will almost always indicate that the solution is fine. Only in the rare cases that an exception occurs does one need to recompute more carefully. This is definitely the way to go on Pentium. Speedups reported in [20] ranged from 1.4 to 6.5.

The third way to accelerate SLATRS, which could complement the above ways, is to more cleverly estimate how big a block we can call STRSV on, based on some data about sizes of off-diagonal entries of the matrix. SLATRS currently does this in a naive way, and we have explored better alternatives as well.

The complex analog CLATRS is used in the above routines as well as in some other LAPACK routines like CTREVC, which computes eigenvectors of triangular matrices. Speedups of CTREVC using exception handling reported in [20] ranged from 1.38 to 1.65. The overall routine CGEEV for the complex nonsymmetric eigenproblem sped up by 8%, since CTREVC is just part of the calculation.

5.2.4 Example 2.4: Eigenvalue calculation using Bisection

This is an inner loop in the ScaLAPACK routine PDLAIECT to compute eigenvalues of symmetric matrices. The simplest version is as follows:
Count the number of eigenvalues less than x of the symmetric tridiagonal matrix T with diagonal entries $a(1:n)$ and off diagonals $b(1:n-1)$ (we assume $b(0)=0$).

$$
\begin{array}{l}
count = 0 \\
d(0) = 1 \\
\text{for } i = 1 \text{ to } n \\
\qquad d(i) = a(i) - x - b(i-1)^2/d(i-1) \\
\qquad \text{if } d(i) < 0,\ count = count + 1
\end{array}
$$

In practice $d(i)$ overwrites $d(i-1)$, and we replace the inner loop by $d = a(i) - x - b(i-1)^2/d$. The trouble with this is that if d is small or zero, overflow occurs. The standard fix is to test to see if d is too small, and explicitly set it to a tiny safe value if it is:

$$
\begin{array}{l}
\text{for } i = 1 \text{ to } n \\
\qquad d = a(i) - x - b(i-1)^2/d \\
\qquad \text{if } (|d| < too\_small),\ d = -too\_small \\
\qquad \text{if } d < 0,\ count = count + 1
\end{array}
$$

A faster way used by PDLAIECT if the user is on an machine implementing IEEE infinity arithmetic (i.e. most of the time) is to go ahead and compute an infinite d, since the next time through the loop $b(i-1)^2/d$ will be zero, and the recurrence will continue uneventfully. This also requires updating count slightly differently, to account for -0s. Speedups reported in [20] range from 1.18 to 1.47.

$$
\begin{array}{l}
\text{for } i = 1 \text{ to } n \\
\qquad d = a(i) - x - b(i-1)^2/d \\
\qquad count = count + \text{signbit}(d)
\end{array}
$$

One can further accelerate this recurrence by reorganizing it to remove the division, which can cost 6 times or more than the other floating point operations. This loop looks like

$$
\begin{array}{l}
\text{for } i = 1 \text{ to } n \\
\qquad p(i) = (a(i) - x) \cdot p(i-1) - b(i-1)^2 \cdot p(i-2) \\
\qquad \text{if } (p(i) \text{ and } p(i-1) \text{ have opposite signs}),\ count = count + 1
\end{array}
$$

The relationship between these loops is that $d(i) = p(i)/p(i-1)$. Again we can remove the subscript on $p(i)$ if we use two temporaries:

$$
\begin{array}{l}
\text{for } i = 1 \text{ to } n \text{ step } 2 \\
\qquad p = (a(i) - x) \cdot p1 - b(i-1)^2 \cdot p2 \\
\qquad \text{if } (p \text{ and } p1 \text{ have opposite signs}),\ count = count + 1 \\
\qquad p1 = (a(i+1) - x) \cdot p - b(i)^2 \cdot p1 \\
\qquad \text{if } (p \text{ and } p1 \text{ have opposite signs})\ count = count + 1 \\
\qquad p2 = p
\end{array}
$$

This loop has no division, and so may be much faster than the first one. Not all machines have such slow division, so this will be machine dependent. However this loop is much more susceptible to over/underflow, because $p(i)$ is the determinant of the leading i-by-i submatrix of $T - x \cdot I$, and determinants easily over/underflow. For example $\det(x \cdot eye(n)) = x^n$ in Matlab notation, and neither x or n has to be very large for overflow to occur. We again have two approaches as in Example 2.3: use extra exponent range in the inner loop (which can reduce the number of scaling tests in the inner loop to one every 128 iterations in single or 16 in double), or IEEE exception handling.

There are various ways to accelerate bisection by using Newton's method or related ideas, but all involve inner loops similar to the one above.

This computation could not be done with extra precise versions of existing BLAS alone, but would require new code with a small number of explicitly declared extra precise variables. If PDLAIECT is "promoted" to a BLAS, these could be hidden from the user.

5.2.5 Example 2.5: Orthogonal Eigenvector calculation with Inverse Iteration

Given the eigenvalues computed from bisection, the standard algorithm to find their corresponding eigenvectors is inverse iteration, which involves solving a tridiagonal system of linear equations. In exact arithmetic, this algorithm would cost $O(n)$ work per eigenvector, i.e. an optimal $O(1)$ work per solution component, and be embarrassingly parallel. Until very recently, roundoff (from reorthogonalization) has made this algorithm cost as much as $O(n^2)$ per eigenvector and be serial, depending on the eigenvalue distribution. Recent work by Parlett and Dhillon promises to attain the promised complexity of the infinitely precise algorithm, and using only input/output precision. Currently their algorithm works better than they can explain, but a complete proof may remain elusive for a while. In the meantime, it is straightforward to drastically reduce the incidence of the "hard cases" (when eigenvalues are tightly clustered) by using higher intermediate precision. Briefly, if computations are done in b bits, and an eigenvalue shares $s < b$ bits with its nearest neighbor, one can very quickly compute its eigenvector with an accuracy of $b - s$ bits. Therefore adding, say, 10 bits to b means eigenvalues must be 2^{10} times closer together before difficulties set in. Since the incidence of having eigenvalues share s leading bits decreases rapidly with s, increasing b decreases the incidence of numerical difficulty rapidly.

This computation could not be done with extra precise versions of existing BLAS, but would require new code, most conveniently written with a compiler that lets us declare extra-precise variables. This would also require extra-precise eigenvalues computed as described in Example 2.4, or by using another algorithm (called dqds) with a similar nonlinear recurrence.

5.2.6 Example 2.6: Eigenvalue calculation using Lanczos

Lanczos is a standard algorithm for finding eigenvalues of a large sparse symmetric matrix A. It requires matrix-vector multiplication by A, and saxpys, dot products, and scaling of vectors. A major cost of the algorithm is reorthogonalization among the sequence of so-called Lanczos vectors x_i computed by the algorithm; in exact arithmetic this sequence consists of orthonormal vectors, but in practice extra work is needed to explicitly reorthogonalize them. Existing error analyses lead us to believe that computing the matrix-vector products with A in higher precision, and returning and computing with the resulting products in higher precision, might make reorthogonalization much more rarely needed. This could result in both a faster and more accurate algorithm. This is a subject of research.

This computation would require higher precision sparse matrix-vector multiply (or a sparse dot product), as well as higher precision dense BLAS1 and BLAS2.

5.2.7 Example 2.7: Using Normal Equations instead of QR for Least Squares

This example requires a bit more than extra precise BLAS, but could easily be implemented easily by using double precision for single precision input. We include it to illustrate the broader utility of mixed precision computation.

The fastest way to solve an overdetermined least squares problem of minimizing $\|Ax - b\|_2$ is to solve the normal equations:

$$(A^T \cdot A) \cdot x = (A^T) \cdot b$$

This requires about half as many flops as the alternative QR-based approach when the number of rows m is much larger than the number of columns n. The trouble is that using the normal equations instead of QR can effectively square the condition number of the problem, and can lose twice as much accuracy. The remedy is to form and solve the normal equations in twice the input/output precision. Unless the condition number is near $1/macheps$, this can be much more accurate than the QR-based approach in single precision. If we use more than input/output precision but less than twice as much, the error bound is proportionately more.

If m is sufficiently larger than n, and the cost of extra precise arithmetic is less than twice the cost of arithmetic in the input/output precision, then this algorithm will be faster than the existing QR based algorithm.

Here is why this cannot be implemented solely with BLAS. We need to be able to compute, return, and continue computing with $A^T \cdot A$ and $(A^T) \cdot b$ in higher precision. This means that we have to be able to declare variables of these types, not just have anonymous variables. Some languages do not permit declaring such high precision variables. Furthermore, Cholesky requires a square root. But if A and b are in single precision, then this could all be done in double precision; the difficulty arises if A and b are in double precision.

5.2.8 Example 2.8: Mixed Real and Complex Arithmetic

LAPACK auxiliary routine CLACRM multiplies a complex matrix A by a real matrix B. It is called by the LAPACK routines for computing eigenvalues and eigenvectors of Hermitian matrices using divide-and-conquer (CSTEDC, CLAED0 and CLAED7), currently the fastest available algorithm for large matrices. It is similarly used by the complex SVD based on divide-and-conquer. The alternative of promoting B to complex and call the BLAS CGEMM would cost 3 times as many floating point operations (4 floating point multiplications and 2 additions per solution component, instead of 2 multiplications). Since this routine is the most time-consuming part of the divide-and-conquer algorithm, it is important that it go fast. Currently CLACRM is implemented by copying the real part of A to contiguous workspace, calling SGEMM, copying the result back, and repeating this for the imaginary part.

LAPACK auxiliary routines CLASR and CSROT apply one or more real 2-by-2 rotations R_i to a complex matrix A. They are called by the LAPACK routines for computing eigenvalues and eigenvectors of Hermitian matrices using QR iteration (CSTEQR), and by the complex SVD, currently the fastest available algorithms for small matrices. The alternative of promoting the R_i to complex would again triple the cost of the most time-consuming part of QR iteration. Currently these routines are implemented in straightforward (unoptimized) Fortran 77.

5.2.9 Example 2.9: Avoiding Pivoting in Gaussian Elimination

This example is speculative, in the sense that it uses extra precision to avoid pivoting, and that this is attractive only if a pivoting algorithm is sufficiently slower or more complicated than a nonpivoting algorithm, since there are time and space costs associated with our scheme. This is likely to be the case with sparse Gaussian elimination on a distributed memory machine, because scheduling, load balancing and memory allocation already uses very complicated and fine-grained synchronization and memory access on a shared memory machine [19], and the only existing distributed memory implementation uses an a priori and sometimes very pessimistic overestimate of the time and space requirement to do static scheduling [50].

A straightforward but expensive way to avoid pivoting in Gaussian elimination applied to A is to replace sufficiently tiny pivots by a tiny number (like $macheps \cdot \|A\|$), and to compute everything to a high enough precision. But since intermediate values have to be computed and stored in high precision, this could eliminate the gains by avoiding pivoting. In particular, loading and storing high precision values could take as much memory bandwidth as pivoting low precision values.

By a careful reexamination of the classical error analysis of Gaussian elimination, one can carefully limit the number of entries of L, U, $y = L^{-1} \cdot b$ and $x = U^{-1} \cdot y$ that must be stored in high precision. The costs appear to be computing all inner products to high precision, and storing as much as one row of U and column of L per tiny pivot. Experiments and error analysis reveal that merely accumulating inner products to high precision usually does not reduce the backward error to $O(macheps)$. Indeed, it is the final rounding error when storing $U(i,j)$ or $L(i,j)$ to memory that is the dangerous one, even if inner products are exact.

5.3 Section 3: High-Precision Arithmetic Implementation Technicalities

Among computers still commercially significant, though a few of them are fading fast, those listed in Table 1 have more than nominal support in hardware for a third floating-point format wider than the nearly ubiquitous REAL*4 (float) and REAL*8 (double) formats.

Table 1: Available High-Precision Hardware Formats

Computer Family	Bytes width	Sig. Dec.	Overflow
Pentiums and earlier Intel-based PCs clones (Cyrix, AMD, ...)	10 or 12	19 - 20	10^{4932}
Motorola 680x0-based Apple Macintosh SUN III, NeXT, ... ; Intel 80960, Merced	12	19 - 20	10^{4932}
IBM RS/6000, Power-PCs -Macs; SGI MIPS 8000, H-P PA RISC 2.0	16	grubby 32	10^{308}
IBM /370, 3090	16	31 - 33	$7 \cdot 10^{75}$
DEC VAX H-format	16	33 - 34	10^{4931}

The 10- and 12-byte formats are the double-extended format of IEEE Standard 754 for Binary Floating-Point Arithmetic. (Two bytes of the 12-byte format, currently blank in memory, mitigate word-alignment penalties. Some day this format may grow 16 additional sig. bits to fill those two bytes; ultimately this extended format may come to occupy 16 bytes. Highest performance for main memory traffic might require 16-byte alignment rather than 12, depending on the machine. Also, a 10-byte alignment might be faster for BLAS 1 and BLAS 2 routines, since there is little data reuse.) In machines that provide it, all floating-point registers have this extended format, so it is the natural format in which to evaluate all floating-point expressions, no matter what the formats of operands. Running never much slower than double does, this extended format is intended primarily to enhance the speed and reliability of software that appears to users to operate upon single and/or double data exclusively. For instance, if MATLAB used this format for more than merely its matrix multiply, the quality of its results, especially for large dimensions and ill-conditioned problems, would distinguish machines that provide this format from those that do not; similarly for LAPACK.

Unfortunately, proper language support for the extended format is limited. In Windows and DOS on Intel-based PCs and clones, this format is supported by C++ and C compilers from Borland, and from Microsoft (but not its latest Visual C++), but by no commercially significant Fortran. On a 680x0-based Macintosh there is one good Fortran compiler from Language Systems Inc. (now Fortner Research LLC), and at least two good C++ and C compilers that support this format; no compilers support it on a SUN III or NeXT. This format is threatened by atrophy, despite Intel's reiterated commitment to its continued support in hardware, partly because of an unenlightened language community's hostility towards it and partly because of political and commercial alliances. Table 2 lists some product lines with strong commitments both to Intel/AMD/Cyrix and 680x0 arithmetic and to some other that lacks its extended format; investments to support this format in one product line would yield no return in the other.

Table 2: Product Lines with Divided Arithmetic Loyalties

Apple:	Old 680x0 Macintosh	vs. Power Mac
DEC:	DEC PC	vs. DEC Alpha and VAX
Hewlett-Packard:	H-P PC	vs. HPUX on PA RISC
IBM:	Old IBM PC and PS2	vs. Power PC, RS/6000
Silicon Graphics:	SGI PC (new!)	vs. MIPS R-x000
Sun:	Solaris on PC	vs. Solaris on SPARC
Microsoft:	Windows NT on PC	vs. NT on DEC Alpha
MathSoft:	MathCAD	on all platforms.
MathWorks:	MATLAB	on all platforms.
Waterloo MAPLE:	MAPLE V	on all platforms.
Wolfram Research:	Mathematica	on all platforms.

The 16-byte format on RS/6000, Power-PC/Mac, etc. is a double-double format constructed by software out of two 8-byte floating-point numbers, one a continuation of the bits of the other. Double-double is supported fully by IBM's Fortran and by Apple's C++ and C compilers, among others. This format is an attractive nuisance because, though rounded in a grubby way, it runs at least twice as fast as true quadruple precision implemented entirely in software. Double-double runs so fast on these machines because of their Fused Multiply-Accumulate operations, which evaluate $A \cdot B + C$, $A \cdot B - C$ or $C - A \cdot B$ for three double operands with only one rounding error.

These operations have now (1997) appeared in the instruction set of the H-P 8000 and, according to word from Ft. Collins CO, will persist with full software support in H-P's PA RISC 2.0 architectural specification. A Fused Multiply-Accumulate instruction exists on the SGI MIPS 8000 too, but not on other MIPS chips; the same instruction on the MIPS 10000 performs Multiply-Accumulate unfused (two rounding errors), which places conundrums in the way of language support.

One initially surprising feature of the double-double format is that the significands in its two double precision words do not have to be contiguous. For example, $2.0^{1000}+0.5^{1000} \approx 10^{301}+10^{-301}$

is an exactly representable double-double number, since both summands are doubles. Violent variations in the density of double-double numbers prevent macheps- like environmental parameters from being used simultaneously as upper bounds for roundoff and lower bounds for the gaps between adjacent double-double numbers. Consequently certain esoteric programming tricks that work for all other hardware floating-point formats cannot be used successfully with double-double. For instance, the stopping criterion in the following loop, which is intended to stop when |errorbound| $<$ *macheps* $\cdot$ |results|, fails:

```
repeat
      compute errorbound for result
until (result + errorbound = result)
```

Also, the accuracy of double-double subtraction can vary from one implementation to another.

The 16-byte formats on IBM 3090s and DEC VAXs are true quadruple precision formats supported fully by the respective manufacturers' compilers, though supported only vestigially in some VAX models' hardware. With full hardware support, these 16-byte formats run at least 1/4 as fast as the 8-byte formats; without hardware support the speed ratio is below 1/10 . Although 3090s and VAXs are not long for this world, their arithmetics are commemorated by a slow but well-behaved quadruple precision format implemented in software in some C++ and C compilers supplied by the manufacturers of SGI MIPS, SUN SPARCs (including gcc), DEC Alphas, and H-P PA-RISCs. Full hardware support for quadruple seems unlikely in the foreseeable future for two reasons. First, chip area is too precious to squander on quadruple's wiring, which uses more area than its transistors do. Second, no computer's sales have ever been shown to depend much upon the speed of its quadruple floating-point, though a few customers clamor for it. Whether these few customers can be satisfied sooner by either better mathematical algorithms or extended than by quadruple is another question that obscures the future, especially since the implemented precision of extended in future chips can creep higher gradually, reaching quadruple no sooner than it has to.

To cope with the vagaries of higher precision arithmetic, a Numerical C Extensions Group (NCEG) put together a proposal now before ANSI X3J11, the committee responsible for the C standard and its C9X revision. Among other things, the proposal would allow programmers to specify a format, presumably either double or extended, in which all subexpressions would be evaluated, and also to specify a pseudonym for a wider-than-double format that, when instantiated at compile time, could become whichever of extended, double-double or quadruple runs fastest on the target machine. By using these options, reusable software could deliver results which, though different on different machines, would be about as good on any particular machine as that machine could deliver at its normal speed. This would provide some incentive to support ever higher precision in hardware as soon as the market's demand for it becomes strong enough, if it ever does.

The vagaries of higher-than-double precision should not prevent BLAS and LAPACK from supporting mixtures of the relatively uncomplicated single/float and double precisions. Most computers nowadays can support the old C-style semantics, evaluating all subexpressions in double (or wider) precision even if they involve exclusively float variables and constants, with no significant performance penalty. On Intel/AMD/Cyrix platforms this is accomplished by setting a control word to round to double precision in the extended registers. On Power PC/Mac and DEC Alpha platforms all floating-point registers already have double width. A few other machines, IBM /370 for instance, have an instruction that produces a double product of two floats exactly. On all these machines the matrix product $D = A \cdot B$ of two float matrices A and B can be accumulated in double about as fast as in float provided the compiler optimizes C statements like $D[i,j] += (double) A[i,k] \cdot B[k,j]$

and Fortran statements like $D(i,j) = D(i,j) + DPROD(A(i,k), B(k,j))$ in the appropriate way. Only Crays, on whose compilers "single" means pretty much what the rest of us mean by "double" would run significantly slower that way.

5.4 Section 4: Naming Scheme and Calling Sequence

We begin with a list of all the possibilities that one could specify, and then describe our simpler scheme and its defaults. We believe that our naming scheme is as simple as possible.

So far we only consider Fortran77. We discuss other language bindings briefly at the end of this section.

We will use matrix multiplication as an example. The calling sequence for the BLAS 3 routine is as follows:

xGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)

which implements

$$C \Leftarrow ALPHA \cdot (op_1(A) \cdot op_2(B)) + BETA \cdot C.$$

Here $op_1(A)$ is M by K, $op_2(B)$ is K by N, C is M by N, and op_1 and op_2 can mean transpose, conjugate transpose, or the identity function, depending on x, TRANSA and TRANSB. Note that we have chosen to insert parentheses around $op_1(A) \cdot op_2(B)$, although other orders of evaluation are possible too; the Fortran BLAS on Netlib insert the parentheses in two ways, depending on input parameters.

The leading character x can be one of the 4 types

Table 3: Existing BLAS Data Types
S (single precision real)
C (single precision complex)
D (double precision real)
Z (double precision complex)

¿From the last section, we can see the possibility of the following 6 additional types (which may or may not be supported by a particular compiler)

Table 4: Possible High Precision Data Types
X (extended precision real)
Y (extended precision complex)
W (double-double precision real)
U (double-double precision complex)
Q (quadruple precision real)
P (quadruple precision complex)

As mentioned in section 3, one can imagine that type X takes 10-bytes to store, but that it would be more efficient for some operations to store it on 12-byte or 16-byte boundaries because of the memory system, but faster to store it on 10-byte boundaries for other operations. The easiest thing to do is to assume that there is only one way to store a 10-byte quantity per machine, and so not distinguish among them, though they could differ from machine to machine. Alternatively, if someone specifies W (or Q) on a machine that best supports X, this could mean variables of type X but stored in 12-bytes (or 16-bytes). The environmental enquiries in section 5 would indicate the actual precision and range in any case.

In principle, the type of each input and output parameter could be specified independently.

Independent of the input and output types, we could specify the internal precision to be used during the matrix multiplication itself. Clearly the following are all possibilities:

Table 5: Possible Internal Precisions - Preliminary List
S (single precision)
D (double precision)
X (extended precision)
W (double-double precision)
Q (quadruple precision)

However, there are several other possibilities we should consider.

First, for most calculations we would be happiest if the internal precision were the widest one done in hardware (i.e. fast) by the machine, which we call "indigenous arithmetic." On the Pentium and its clones, this means precision X (80-bit arithmetic). On other machines (from SUN, HP, IBM and DEC) it means precision D (double precision). So we might want to designate internal precision I for indigenous arithmetic, which would be either X or D depending on the machine.

Second, for iterative refinement and some other applications where extra precision is essential and its cost low, we want to use whatever extra-precise arithmetic is available, and do not care whether it is X (on Pentiums) or W (on most other machines). So we might want to designate E for Extra precision, which would be either X or W, depending on the machine.

But these are still not the only possibilities that one might want. An expression tree for xGEMM is

```
ALPHA*(op_1(A)*op_2(B)) + BETA*C
                    |
                    |
                    +
                   / \
                  /   \
                 /     \
                /       \
               *         *
              / \       / \
             /   \     /   \
         ALPHA    *  BETA   C
                 / \
                /   \
               /     \
           op_1(A)  op_2(B)
```

The traditional way to evaluate this or any other expression in Fortran or C is to use the wider precision of either operand of any binary operator. So if ALPHA, A, B, and C are single precision real, but BETA is double precision, BETA $\cdot$ C and the addition will be done in double precision, and the other two multiplications in single. We denote this so-called "strict" evaluation scheme by T for "traditional" (since S is already taken). We note that the expression tree for the Fortran BLAS on Netlib varies depending on input parameters, sometimes multiplying ALPHA by op_1(B) before doing the matrix multiplication. Other BLAS implementations may vary.

Finally, a perhaps more natural way to evaluate such a mixed precision expression is to do all operations in the widest precision of any type in the expression. In the example of the last paragraph, this means all operations would be in double precision. This presumes the existence of a total ordering on all the precisions in Table 5. A possible difficulty here is that if a machine has both X and W arithmetic, it is conceivable that X might have less precision but a wider exponent range than W, making them hard to order. However, we doubt that any machine would have 3 distinct precisions D, X and W that were in addition not totally ordered, and in any case leave it to the implementation to impose a total order. The natural letter to designate this "evaluate to widest" scheme is W, which is already taken by double-double, so we use M for "evaluate using the Most precise type appearing."

Finally, we note that a few machines like the IBM 4361 actually support exact inner products. This entails simulating a register wide enough (hundreds or thousands of bits) so that the square of the smallest floating point number can be added to the square of the largest floating point number without roundoff. We mention this possibility not because it is common or because we recommend it, but to list all the possibilities that we might want to accommodate. We indicate this internal precision by L for Long, since E (for Exact) is already taken.

Altogether, this yields the following table of possible "internal precisions"

Table 6: Possible Internal Precisions - Final List
S (single precision)
D (double precision)
I (indigenous precision, either D or X)
X (extended precision)
E (extra precision, either X or W)
W (double-double precision)
Q (quadruple precision)
T (traditional)
M (most precise type appearing in expression)
L (long)

Finally, we can present our proposed calling sequence for extended precision matrix multiplication:

XxGEMM (TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC, PREC)

The differences between this calling sequence and the one for xGEMM above are as follows:

1. The first letter is X, and the prefix x can be taken from either Table 3 or Table 4. x specifies the precision of the output variable (in this case the matrix C). Not all precisions from Table 4 may be implemented.

2. The final (optional) parameter PREC is a character string specifying the internal precision and input variable types. It is interpreted as follows.

   (a) If PREC is a null string (or missing), then the types of all the input variables default to be the same as the output variable as specified by x. The internal precision must be AT LEAST as high as the precision associated with x. The actual precision is available from the enironmental enquiry specified below. Note that this assumes a total ordering on the precisions available in Table 6 (except for T and M, which are equivalent to one of the other precisions in this case, since all input and output types are identical to x).

(b) If PREC contains a single character, it must be from Table 6 (otherwise it is an error) and specifies the internal precision. Again, the implementation is free to use at least as much precision as specified, as reported by the environmental enquiry routine. Since the output and input precisions are all the same (x), choosing either PREC=T or PREC=M is equivalent to choosing PREC=x.

(c) If PREC contains more characters, these specify the input types, associating with the input variables from left to right. For example PREC = 'XSDCZQ' means that the internal precision is (at least) X, and the types of ALPHA, A, B, BETA and C are S, D, C, Z and Q, respectively. If there are fewer characters than input variables, then the rightmost character applies to the remaining input variables. Thus XDGEMM(... , 'DS') returns a double precision product from single precision inputs, using (at least) double precision internally.

We believe that this scheme requires the least change (a single extra character) to get most of the desired functionality, but permits detailed control over precision if desired. If a system does not permit the user to declare variables of type Q, say, then this will be obvious at load time because XQGEMM will not exist. If the user asks for internal precision W but the system better supports the wider precision Q, then the implementor is free to use Q instead.

One possible exception to the rule that the system may use precision at least as high as requested by the user might be T. We cannot think of a situation where the user would explicitly specify traditional evaluation, as well as differing individual input precisions, without wanting this specific evaluation scheme.

Other language bindings for F90, C, C++, and Java remain to be determined. We expect to adopt and modify the interface recommendations made in other proposals accepted by the BLAST Forum. For example, in the case of C we anticipate adding a PREC variable of an enumerated type including the above strings (whichever ones are available in a particular implementation). Object oriented languages likely have much better facilities for determining the types of their operands at compiler time and branching appropriately.

5.5 Section 5: Environmental Enquiries

We propose to extend xLAMCH to handle enquiries about the internal precision used. Ideally, one would use the same routine name with extra arguments, which if present, would have the meaning specified below. If this is not possible or desirable, then we would use XxLAMCH instead. The calling sequence is as follows:

xLAMCH(CMACH, PREC, ROUTINE)

CMACH as the same meaning as in the current xLAMCH, except that the arguments

S (safe minimum, smallest number such that 1/safmin does not overflow)
U (underflow threshold)
O (overflow threshold)

are not permitted, because their values may not be representable in precision x (for example, suppose the internal precision is D and x=S). Information about these values is available from arguments M (exponent of U) and L (exponent of O). This way, all data about the widest precision will be representable in the narrowest precision.

PREC has the same meaning as in the last section. If the precision is not implemented, xLAMCH returns -1 for any CMACH.

ROUTINE, if present, is a string with the name of the routine being asked about (eg 'XSGEMM'). This provides a way to ask about the implementation of a particular routine. If the routine is not implemented with the precision PREC, xLAMCH returns -1 for any CMACH.

If ROUTINE is not present, or is the null string, only the first character of PREC is relevant. If ROUTINE is present, the other characters of PREC are interpreted as in the last section.

Here is what the values returned by xLAMCH mean. If xLAMCH(CMACH, PREC, 'XxGEMM') returns EPSz, BASEz, EMINz and EMAXz for internal precision PREC, and xLAMCH(CMACH, 'x') returns EPSx, BASEx, EMINx and EMAXx for output type x, then this means that XxGEMM satisfies the following error bound.

Let

```
Uz = BASEz**(EMINz-1)          ... underflow threshold for internal precision
Ux = BASEx**(EMINx-1)          ... underflow threshold for input/output type
Oz = BASEz**(EMAXz-1)*(1-EPSz) ... overflow threshold for internal precision
Ox = BASEx**(EMAXx-1)*(1-EPSx) ... overflow threshold for input/output type
```

Then the error bound is

```
| computed C(i,j) - true C(i,j) | <=
      (K+2)*EPSz* \sum_{k=1}^K |ALPHA*op_1(A)(i,k)*op_2(B)(k,j)|
     +(K+1)*EPSz* |BETA*initial C(i,j)|
           ... roundoff from internal precision
     +EPSx* |computed C(i,j)|
           ... roundoff from rounding result to output precision
     +(2*K*|ALPHA|+3)*Uz
           ... underflow from internal precision
     +Ux
           ... underflow from rounding result to output precision
```

This bound reflects the following assumptions:

1. Strassen's or other fast methods may not be used, but the terms in the dot products may be accumulated in any order.

2. The parentheses in $ALPHA \cdot (op_1(A) \cdot op_2(B))$ are respected.

3. Gradual Underflow or Flush-to-Zero is permitted.

4. No intermediate value exceeded Oz in magnitude, and the final result does not exceed Ox in magnitude.

Analogous formulas exist for other BLAS.

Since the standard BLAS library is not forbidden from using extra precision, we propose that this enquiry also apply to the standard BLAS. In other words, one could equally well call xLAMCH(CMACH,'SGEMM') as xLAMCH(CMACH,'XSGEMM','X'), say. Furthermore, we encourage developers to use higher precision if the performance penalty is insignificant. This gives us added flexibility in enquiring about the internals of the BLAS. For example, we could have an argument CMACH='A' and return data about the algorithm used. The return argument might be

1 if standard matrix multiply used, with order of summation is guaranteed to be in some particular order.
2,3, ... same as 1 with other orders, or no order guaranteed
4 Strassen's method

5.6 Section 6: Prioritized List of Extra Precision BLAS

In this section we described the most important extended precision BLAS to implement.

`XxDOT(N, X, INCX, Y, INCY, PREC)`

The dot product is the basis of most other high precision operations. Initially just a single character PREC would be supported, but eventually it should be possible to compute and return the result to higher precision than the inputs, to support using the normal equations for least squares problems. Also, permitting one input to be real and the other complex would support Hermitian eigenvalue algorithms based on divide-and-conquer; a simple implementation would simply consist of two calls to the existing BLAS xDOT.

`XxDOTG(TRANS, N, ALPHA, X, INCX, Y, INCY, BETA, C, PREC )`

This implements

$$C \Leftarrow ALPHA \cdot \sum_i op(X(1+i \cdot INCX)) \cdot Y(1+i \cdot INCY) + BETA \cdot C$$

and is intended to be the natural building block for XxGEMV and XxGEMM below. The G in DOTG stands for "general"

`XxGEMV( TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY, PREC )`

Matrix-vector multiplication is the most natural routine with which to implement iterative refinement as described in section 2. It cannot be implemented directly using XxDOT for general ALPHA and BETA, which is why we proposed XxDOTG.

A sparse version of this routine (or of DOT), where A is lower precision than X and Y, would support Lanczos.

`XxTRSV ( UPLO, TRANS, DIAG, N, A, LDA, X, INCX, SCALE, PREC )`

Solution of triangular systems can also be based on XxDOT, but can take special advantage of exception handling as described above. The routine solves op(A) $\cdot$ Y=SCALE $\cdot$ X (where the solution Y overwrites X), and SCALE is chosen to avoid over/underflow. Alternatively, the name and calling sequence of the LAPACK auxiliary routine xLATRS could be used instead.

5.7 Section 7: Automatic Generation of These Routines

We assume that C is our implementation language. In this case, all the floating point variables would be passed in as void*. There would be a big case statement at the beginning based on PREC to branch to right set of nested loops. Each set of loops would differ only in the casts of the void* variables, and could be generated automatically (optimization would require use of PHiPAC [11] or a similar system).

5.8 Section 8: Automatic Testing of These Routines

We could generate test cases whose answers we know exactly as a function of the intermediate precision. These matrices would be chosen to to give different answers as a function of the input precision, and so serve as a signature of the precision. The same examples could be used to implement xLAMCH.

5.9 Section 9: Sample Implementations

We could just do XxDOTG in C assuming single precision input as an illustration. This should be quite short.

5.10 Section 10: Sample Applications

The first example to do is iterative refinement for linear systems and least squares, since the effort is so small and the payoff so large. This is particularly true for single precision input. We would code these without respecting the detailed specifications above, just quick-and-dirty to measure the speed and accuracy.

Some of the greatest benefits described above for least squares and the symmetric eigenvalue problem will require new routines not currently part of the BLAS. Writing them would require a few variables that are higher precision and/or range than the input/output variables. When the input variables are single, we can simply use double, and this should be the next thing we do. When the input is double, we have a problem, because most compilers do not support the declaration of higher precision variables. This is even true on Pentiums, which have hardware extended precision. How should we write such code?

One way is to propose new BLAS like XXMUL(A,B,C) which would compute the product C=A · B of extended precision variables, where A, B and C are arrays of appropriate length. Given similar routines for add, subtract, divide, sqrt and comparison, we could essentially write all our routines in assembly language. This approach has two big drawbacks: it encourages us to produce inscrutable, machine-dependent, hard-to-maintain and hard-to-port code, i.e. life before IEEE 754. Second, it largely removes pressure from compiler writers to accommodate us.

The second way is to just write relatively simple code using double as the extended precision when the input is single, and perhaps one impressive (if painful) benchmark with double precision input, to hold out a carrot to compiler writers to provide appropriate support for extended variables. So we might write XxMUL , XxADD, XxSUB, XxDIV, XxSQRT, XxCMP, XxCOPY for our own use, but not propose them as a standard.

5.11 Section 11: Comparison to a previous proposal

Appendix B of the BLAS2 paper [2] specifies a set of extra-precise BLAS, which the authors called EBLAS. This proposal naturally has a lot in common with this earlier proposal, but there are some important differences: the earlier proposal made several more limiting assumptions about uses of the BLAS, machine architectures, and compilers than we have here. They assumed that it was necessary to have at least one extended precision argument if the internal precision were to be extended. In other words, not all uses of extra precision could be hidden in the BLAS, so the ability to declare REAL*16 and COMPLEX*32 variables was required. In contrast, we can get a lot of use out of extra precision BLAS in cases where no such declarations need to be made (iterative refinement). Furthermore, they did not specify environmental enquiries, which are important to get error bounds. We believe our current updated proposal better matches current understanding of algorithms and the limitations and opportunities of modern architectures and compilers.

Bibliography

[1] *The EM Symmetric Sparse Supernodal Format*. Elegant Mathematics, Inc., Bothell, Washington, 1993.

[2] *The EM Unsymmetric Sparse Supernodal Format*. Elegant Mathematics, Inc., Bothell, Washington, 1993.

[3] M. Aboelaze, N. Chrisochoides, and E. Houstis. The Parallelization of Level 2 and 3 BLAS Operations on Distributed Memory Machines. Technical Report CSD-TR-91-007, Purdue University, West Lafayette, IN, 1991.

[4] R. Agarwal, S. Balle, F. Gustavson, M. Joshi, and P. Palkar. A Three-Dimensional Approach to Parallel Matrix Multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[5] R. Agarwal, F. Gustavson, and M. Zubair. A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[6] E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 2nd edition, 1995. (Also available in Japanese, published by Maruzen, Tokyo, translated by Dr Oguni).

[7] P. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Master's thesis, Mississippi State University, 1995.

[8] R. E. Bank and D. J. Rose. Marching algorithms for elliptic boundary value problems. the constant coefficient case. *SIAM Journal of Numerical Analysis*, 14:792–829, 1977.

[9] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (SMMP). *Advances in Computational Mathematics*, 1, 1993.

[10] B. Bialecki and G. Fairweather. Matrix decomposition algorithms in orthogonal spline collocation for separable elliptic boundary value problems. *SIAM J. Sci. Comput.*, 16:330–347, 1995.

[11] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the International Conference on Supercomputing*. ACM SIGARC, 1997. Vienna, Austria.

[12] R. Bisseling and J. van der Vorst. Parallel Triangular System Solving on a mesh network of Transputers. *SIAM Journal on Scientific and Statistical Computing*, 12:787–799, 1991.

[13] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997.

[14] R. Brent and P. Strazdins. Implementation of BLAS Level 3 and LINPACK Benchmark on the AP1000. *Fujitsu Scientific and Technical Journal*, 5(1):61–70, 1993.

[15] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In J. Dongarra, K. Masden, and J. Waśniewski, editors, *Applied Parallel Computing*, pages 107–114. Springer Verlag, 1995. (also LAPACK Working Note No.100).

[16] J. Choi, J. Dongarra, and D. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994. (also LAPACK Working Note No.57).

[17] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subroutines. *Concurrency: Practice and Experience*, 8(7):517–535, 1996.

[18] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. Technical Report TR95-49, Department of Computer Sciences, UT-Austin, 1995. Submitted to Concurrency: Practice and Experience.

[19] J. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination, February 1997. To appear in SIAM J. Matrix Anal. Appl., special issue on Sparse and Structured Matrix Computations and Their Applications (Also LAPACK Working Note 124).

[20] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43:983–992, 1994.

[21] D. S. Dodson. Corrigendum: Remark on "Algorithm 539: Basic Linear Algebra Subroutines for FORTRAN usage". *ACM Trans. Math. Software*, 9:140, 1983. (See also [38] and [22]).

[22] D. S. Dodson and R. G. Grimes. Remark on algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software*, 8:403–404, 1982. (See also [38] and [21]).

[23] D. S. Dodson, R. G. Grimes, and J. G. Lewis. Sparse extensions to the FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 17:253–272, 1991. (Algorithm 692).

[24] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

[25] 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*, 14:1–32, 399, 1988. (Algorithm 656).

[26] I. S. Duff, A. M. Erisman, and J. K. Reid. Direct methods for sparse matrices. In *Monographs on Numerical Analysis*. Oxford University Press, New York, 1986.

[27] Iain Duff, Michele Marrone, and Giuseppe Radicati. A proposal for user level sparse BLAS. Technical Report TR/PA/92/85, CERFACS, 1992.

[28] A. Elster. Basic Matrix Subprograms for Distributed Memory Systems. In D. Walker and Q. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 311–316. IEEE Press, 1990.

[29] R. Falgout, A. Skjellum, S. Smith, and C. Still. The Multicomputer Toolbox Approach to Concurrent BLAS and LACS. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*. IEEE Computer Society Press, 1992.

[30] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[31] G. Fox, S. Otto, and A. Hey. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 3:17–31, 1987.

[32] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28:505–535, 1974.

[33] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[34] M. Heath and C. Romine. Parallel Solution Triangular Systems on Distributed Memory Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

[35] M. A. Heroux. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90, CERFACS, 1992.

[36] S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone DELTA. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[37] L. Kaufman and D. D. Warner. High-order, fast direct methods for separable elliptical equations. *SIAM Journal of Numerical Analysis*, 21:672–695, 1984.

[38] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979. (Algorithm 539. See also [22] and [21]).

[39] G. Li and T. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9(3):485–502, 1988.

[40] G. Li and T. Coleman. A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 10(2):382–396, 1989.

[41] W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM Journal on Scientific and Statistical Computing*, 14(6):1259–1288, 1993.

[42] K. Mathur and S. L. Johnsson. Multiplication of Matrices of Arbitrary Shapes on a Data Parallel Computer. *Parallel Computing*, 20:919–951, 1994.

[43] O. McBryan and E. van de Velde. Matrix and Vector Operations on Hypercube Parallel Processors. *Parallel Computing*, 5:117–126, 1987.

[44] R. Pozo. Template numerical toolkit (tnt), 1995. `http://math.nist.gov/tnt`.

[45] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Preliminary Version.

[46] P. Strazdins. A High Performance, Portable Distributed BLAS Implementation. In *Proceedings of the Sixth Parallel Computing Workshop*, Fujitsu Parallel Computing Center, 1996.

[47] Thinking Machines Corporation. *CMSSL for Fortran*, 1990.

[48] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, Cambridge, Massachusetts, 1997.

[49] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Technical Report UT CS-95-286, LAPACK Working Note No.96, University of Tennessee, 1995.

[50] T. et al. Yang. Distributed memory parallel sparse lu. Ucsb preprint, University of California, Santa Barbara.