Performance Improvements to LAPACK for the Cray Scientific Library

Edward Anderson *
Cray Research
655F Lone Oak Drive
Eagan, MN 55121

Mark Fahey †
Department of Mathematics
University of Kentucky
Lexington, KY 40506

April 22, 1997

Abstract

This report details local modifications to LAPACK routines for the Cray Scientific Library. Performance of selected routines is evaluated and compared to that of the public domain LAPACK and the equivalent routines from LINPACK or EISPACK. Timing results from a CRAY T90 series computing system are presented.

1 Introduction

The Cray Scientific Library [5] provides an optimized implementation of the public domain software package LAPACK [1] for solving dense linear systems and finding eigenvalues/eigenvectors or singular values/singular vectors of dense matrices. Most of the optimizations come by way of the BLAS [8, 9, 13], a collection of Basic Linear Algebra Subprograms which have been well-tuned for the Cray architectures. Many of the computationally-intensive algorithms in LAPACK, such as the LU, Cholesky, and QR factorizations and the reductions to Hessenberg, tridiagonal, and bidiagonal form, do most of their work in the BLAS and have already been studied in detail [3, 6, 10]. These algorithms are well-suited to the Cray architectures, and their implementation in the Cray Scientific Library differs little from that of the public domain LAPACK.

The LAPACK routines employed after a factorization or reduction has been completed have attracted less attention from performance optimizers, and it is here where most of the local improvements have been made. This report discusses performance improvements for the Cray Scientific Library to a wide range of LAPACK routines, including:

- triangular solve routines in linear system solving,
- auxiliary routines for computing elementary reflections and rotations,
- balancing for the nonsymmetric eigenvalue problem,
- symmetric and nonsymmetric inverse iteration,
- the QZ algorithm for the generalized nonsymmetric eigenvalue problem.

*eca@cray.com
†mrfahey@ms.uky.edu

Many of these modifications are algorithmic improvements that would be beneficial on architectures other than those considered in this report. Other more Cray-specific enhancements, such as selective inlining of BLAS, will be mentioned briefly for completeness. An interesting but otherwise irrelevant empirical result on the optimal blocksize for block factorization algorithms is included in an appendix.

The metric for much of this optimization work was the output generated by the timing programs distributed with the LAPACK test package [4]. These programs show the performance of standard LAPACK routines against that of the equivalent EISPACK routines, inviting comparison. We observed initially that some of the EISPACK routines were faster than the LAPACK routines that were supposed to replace them. We also found instances in which the Cray Scientific Library version of EISPACK was faster than the public domain version of EISPACK, and worked to identify those optimizations and transfer them to LAPACK.

In § 2, we describe the computing environment and identify the different libraries compared in this report. In § 3 and § 4 we review all the local modifications to LAPACK library routines. Changes to the algorithm or loop structure are called out in separate subsections; changes that involved only inlining are collected at the end. We conclude with followup remarks in § 5.

2 Preliminaries

2.1 Compute environment

Our compute environment consisted of a CRAY T94, a parallel-vector processing (PVP) computing system with four processors, a shared memory space of 128 MWords, and IEEE floating-point arithmetic. The cycle time is 2.222 nsec (450 MHz), and the vector floating-point units can retire four operations per clock, giving a peak rate of approximately 1800 Mflops per processor. Although the CRAY T94 is a parallel processor, most of the algorithms we evaluated are serial, so most of our results are for a single processor.

2.2 Libraries

We will compare the performance of our enhanced version of LAPACK to the public domain version, as well as to LINPACK[7] and EISPACK[16], which are earlier collections of Fortran 77 subroutines for solving linear systems, eigenvalue problems, and singular value problems. The following conventions will be used in referring to the different libraries:

libsci is the name of the Cray Scientific Library where the modifications described here have been implemented

LAPACK in this report refers to the public domain source code for LAPACK 2.0, available from netlib at `http://www.netlib.org`.

LINPACK refers to the public domain source code for LINPACK.

EISPACK refers to the modified source code for EISPACK supplied with the LAPACK timing routines. Further details on the modifications to EISPACK are given below.

libsci EISPACK is a unique hybrid of an early version of EISPACK with Cray library enhancements and the algorithmic changes from the 1983 update to EISPACK. Although this software is being phased out of libsci, we include it for comparison in a few cases where it is still the fastest alternative.

The naming convention for LAPACK routines in the Cray Scientific Library uses a leading 'S' in the routine name to indicate a subroutine operating on 64-bit real data and a leading 'C' for 64-bit complex data. On many other systems, 64-bit data is double precision, and the equivalent precision library routines would have a leading 'D' or 'Z' in their name. For example, the subroutine to compute an LU factorization of a 64-bit REAL array is named SGETRF in the Cray Scientific Library but DGETRF in SGI's CompLib. All comparisons in this report are for subroutines operating on full precision (64-bit real or complex) data.

2.3 Modifications to EISPACK

One difficulty in comparing LAPACK to EISPACK is that there are many test problems in the LAPACK timing suite for which EISPACK does not converge or converges with less accuracy than LAPACK. To remedy this, the LAPACK designers provided a modified version of EISPACK with the LAPACK timing suite in which the convergence criteria were changed to be more like LAPACK. Specific changes include

- The maximum number of iterations allowed for convergence was increased from 30 to 40 in IMTQL1 and IMTQL2.

- The test for determining if offdiagonal elements are small enough for the matrix to be split was relaxed in BISECT and TRIDIB. In EISPACK, the criterion was to split the matrix if
$$|E(i)| \le \varepsilon\left(|D(i)| + |D(i-1)|\right).$$
This condition implies that $|E(i)| \le 2\varepsilon \max\left(|D(i)|, |D(i-1)|\right)$, so if both $D(i)$ and $D(i-1)$ are less than $\mathtt{UNFL}/2\varepsilon$, where UNFL is the underflow threshhold, then $E(i)$ must be zero to split. In LAPACK, the criterion is to split the matrix if
$$|E(i)|^2 < \varepsilon^2 \, |D(i)| \, |D(i-1)| + \mathtt{SAFMIN},$$
where SAFMIN is the smallest representable number in the floating point model (the "safe minimum"). Thus $|E(i)|$ may be greater than $\varepsilon\,|D(i)|$ if it is sufficiently smaller than $\varepsilon\,|D(i-1)|$, and the matrix will always split if $|E(i)| < \sqrt{\mathtt{UNFL}}$, regardless of the values of D.

- Global matrix information was added to the splitting criterion for determining if offdiagonal elements are small enough in COMQR, COMQR2, HQR, and HQR2. For instance, when looking for a single small subdiagonal element in HQR, the test was
$$|H(i, i-1)| \le \varepsilon S$$
where $S = |H(i-1, i-1)| + |H(i, i)|$, or $S = \|H\|$ if this value is zero. This test was changed to
$$|H(i, i-1)| \le \max(\varepsilon S, \mathrm{r})$$

where $r = \max(\texttt{SAFMIN}, \varepsilon \|H\|)$. This allows the matrix to split if $H(i, i-1)$ is small either in an absolute sense or relative to the norm of the matrix H, even if it is not small relative to its immediate diagonal neighbors. Previously, global matrix information was incorporated only if $H(i-1, i-1)$ and $H(i, i)$ were zero.

- The absolute test for small offdiagonal elements in TQLRAT was replaced with a relative test. Instead of checking for values of $E2(i) < \varepsilon^2$, where $E2(i) = E(i)^2$, the new test checks for $\sqrt{E2(i)} < \varepsilon\,(|D(i)| + |D(i+1)|)$.

It is debatable whether or not EISPACK with all these changes is really EISPACK any more, but this is the methodology that has been established for comparing LAPACK to EISPACK.

2.4 Test Matrix Types

Some of the eigenvalue routines studied in this report have different performance characteristics for different types of matrices. This is because their rate of convergence depends on the separation of adjacent eigenvalues and on whether or not there are repeated eigenvalues in the solution. In the timing results of § 4, we report the matrix type along with the timing data. The enumeration of matrix types follows that used in the LAPACK timing program, which is described in further detail in the *Installation Guide for LAPACK* [4]. If no type is indicated, matrices of type 1 were used.

3 Modifications to LAPACK, I: Linear System Solving

The modifications we have made to subroutines in the LAPACK library are extensive and so we have divided them into two groups. The first group, described in this section, consists of improvements to the software for solving linear systems of equations and least squares problems. The second group, described in § 4, consists of improvements to the software for solving eigenvalue and singular value problems. Because of the modular design of LAPACK, there is some overlap between the two groups.

3.1 Linear System Solve Routines (xxxTRS)

Solving a linear system with multiple right-hand sides is a naturally parallel operation because each right-hand side can be solved independently. However, since vectorized code can run about 10 times faster than scalar code on a single Cray processor, it is often better to try to vectorize first. In keeping with their BLAS-first strategy, the LAPACK solve routines always vectorize across right-hand sides, leaving any opportunities for parallelism to the underlying BLAS. This approach may be acceptable if the number of right-hand sides is large relative to the number of processors, but it is inefficient for the small numbers of right-hand sides that are often found in applications.

Our strategy for redesigning the solve routines was to move the parallelism to the outermost loop. The standard solve routine xxxTRS was renamed xxxTS2@, and special case code for one right-hand side was added to xxxTS2@. Then a new routine xxxTRS was written to call xxxTS2@ in a parallel loop. The stride for the parallel loop, called `NB` by analogy with the block factorization routines where `NB` is the block size, is determined by a call to a new auxiliary routine ILATRS@, which returns `NB` = 1 if the number of right-hand sides is too small to vectorize, and NB > 1 to have each processor solve for NB right-hand sides at a time.

The structure of SSYTRS with the new design is as follows:

```
      IF( NRHS.EQ.1 ) THEN
         NB = 1
      ELSE
         NB = MAX( 1, ILATRS@( 1, 'SSYTRS', UPLO, N, NRHS, -1, -1 ) )
      END IF
*
      IF( NB.GE.NRHS ) THEN
         CALL SSYTS2@( IUPLO, N, NRHS, A, LDA, IPIV, B, LDB )
      ELSE
CMIC$ CNCALL
         DO 10 J = 1, NRHS, NB
            JB = MIN( NRHS-J+1, NB )
            CALL SSYTS2@( IUPLO, N, JB, A, LDA, IPIV, B( 1, J ), LDB )
   10    CONTINUE
      END IF
```

Note that we avoid the call to ILATRS@ when NRHS = 1 to minimize overhead in this common case. Also, the first character argument has been converted to an integer in SSYTS2@, for historical reasons not relevant to this report.

If only one processor is available, then ILATRS@ returns NB = 1 if it is more efficient to solve for each right-hand side separately, and NRHS if it is more efficient to solve for them all at once, vectorizing across right-hand sides. This is yet another tunable parameter for LAPACK implementors, but it is straightforward to determine. All that is needed is to create two tables, one where NB is set to 1 and one where NB is set to NRHS, and observe the crossover point on a single processor at which the vectorizing code wins out over solving for each right-hand side individually. For Cray PVP systems, the crossover point is typically around 8 right-hand sides, although finer tuning was used in the library.

If multiple processors are available, then an environment-specific decision must be made about the setting of NB. Cray PVP systems utilize a dynamic, demand-driven scheduling mechanism for assigning processors to processes. In a typical batch environment, users may specify a maximum number of processors for their jobs via the environment variable NCPUS, but the actual number of processors assigned may vary from 1 to NCPUS during execution, depending on the system load and the number of processors kept busy by the user's job. To make the best use of the available resources under these circumstances, a dynamic load-balancing algorithm such as Guided Self-Scheduling [15] is often employed. However, if the system load is light, such as during dedicated time, the user may be able to expect all NCPUS processors to be available. Then static load balancing can be used to keep all the processors busy and minimize the execution time. The Cray Scientific Library reads the environment variable MP_DEDICATED to choose between its batch (MP_DEDICATED = 0) and dedicated (MP_DEDICATED = 1) scheduling strategies.

3.1.1 Non-dedicated strategy

In a multi-processing batch environment, the goal is to use multiple processors without degrading the single-processor efficiency too much. Then, if the worst case occurs and only one processor is attached to the user's job at run time, the execution time will be only slightly worse than if the user had just set NCPUS = 1. These are fuzzy guidelines, so the scheduling strategy used for the LAPACK solve routines is heuristic. Using the observation that the single-processor solve routines have nearly reached their asymptotic speed when $\texttt{NRHS} = \texttt{VLEN}$ (the length of the vector registers), we divide the number of right-hand sides into $m = \lceil \texttt{NRHS}/\texttt{VLEN} \rceil$ pieces, each of size $\texttt{NB} = \lceil \texttt{NRHS}/m \rceil$. For example, if $\texttt{NRHS} = 200$ and $\texttt{VLEN} = 128$, we divide the number of right-hand sides into $m = \lceil 200/128 \rceil = 2$ pieces, each having $\texttt{NB} = 100$ right-hand sides, regardless of the number of available processors. If more than two processors are attached to the job at run time, we count on the system to reclaim the processors left idle.

For instance, the solve routine SSYTRS with $N = 512$ runs at 1256 Mflops when $\texttt{NRHS} = 200$, at 1226 Mflops when $\texttt{NRHS} = 100$, and at 953 Mflops when $\texttt{NRHS} = 50$. If the system had 4 processors, we would still use only 2, and each would solve 100 right-hand sides at a rate of of 1226 Mflops, for a possible speedup of $2 \cdot (1226/1256) = 1.96$. If the system were so busy that we only ever got one processor, we would still run at $1 \cdot (1226/1256) = 0.98$ times the single-processor rate. Note that we could have tried to divide the right-hand sides into four pieces for a potential speedup of $4 \cdot (953/1256) = 3.04$, but in the worst case we would run at only $1 \cdot (953/1256) = 0.76$ times the single-processor rate. Since we are unlikely to get the total number of processors on a busy system, the conservative cutting strategy is preferred.

3.1.2 Dedicated strategy

In a multi-processing dedicated environment, the goal is to minimize the wall-clock time of the one running process by using all available resources without regard to single-processor efficiency. The simple-minded strategy employed in this case is to give each of the p processors $\texttt{NRHS}/p$ right-hand sides to solve. If NRHS is large relative to p, this splitting gives a good load balance. However, the speedup will be less than p because a single processor can solve for NRHS right-hand sides at a higher rate of speed than it can solve for $\texttt{NRHS}/p$ right-hand sides.

For example, the solve routine SSYTRS with $N = 512$ and $\texttt{NRHS} = 10$ runs at about 354 Mflops on a single processor of the CRAY T94 using code that vectorizes across right-hand sides. When $\texttt{NRHS} = 5$, the solve runs at about 258 Mflops using code that does not vectorize across right-hand sides. If two processors equally share the work, the speedup (ignoring any multiprocessing overhead) would be $(10/354)/(5/258) = 1.46$. The cumulative CPU time will go up using the parallel method, but we assume that, in a dedicated environment, we only care about wall-clock time.

3.1.3 Sample performance improvements

The dedicated and non-dedicated cutting strategies are the same if $\texttt{NCPUS} = 1$, so we will illustrate the improvements in the single-processor case. Table 1 compares the performance of SSYTRS from LAPACK with the new design in libsci. The factors of 3–8 times improvement for one right-hand side were the obvious motivation for this work. Particular problem sizes may still benefit from some finer tuning; for example, it appears from this table that $N = 750$ should use non-vectorizing code for $\texttt{NRHS} = 8$, while the current code does not increase the cutoff from 7 to 8 right-hand sides until $N = 768$.

		Values of N					
Version	`NRHS`	50	100	250	500	750	1000
LAPACK SSYTRS	1	14	23	36	42	45	47
	2	27	45	68	81	87	90
	4	53	84	126	149	157	164
	8	101	160	232	276	294	304
	32	284	443	636	724	766	771
	100	467	727	1008	1128	1185	1138
libsci SSYTRS	1	35	64	142	247	336	409
	2	34	64	140	245	334	406
	4	54	87	130	247	335	409
	8	102	163	242	286	306	410
	32	287	450	657	757	791	815
	100	494	773	1077	1207	1204	1297

Table 1: Speed in megaflops for SSYTRS, CRAY T94, 1 processor

3.2 Tridiagonal Solvers

The case `NRHS` = 1 is especially important in tridiagonal solvers, for which operations on the right-hand side constitute a major portion of the work. The LAPACK routines we studied were the factorization routines xGTTRF and xPTTRF, the solve routines xGTTRS and xPTTRS, and the driver routines xGTSV and xPTSV. Unlike other LAPACK driver routines, xGTSV contains special case code for `NRHS` = 1 that combines the factor and solve, similar to the LINPACK routines xGTSL and xPTSL. Using the LAPACK timing program to measure our progress, we set out to make the LAPACK tridiagonal solvers at least as fast as their LINPACK equivalents, starting from a point at which LAPACK was up to two times slower.

3.2.1 Tridiagonal factorizations

In SGTTRF, the LU factorization routine for a real general tridiagonal matrix, a row interchange is done at the i^{th} step if the subdiagonal element `DL`(i) is greater than the diagonal element `D`(i). The LAPACK implementation uses two tests to select the pivot:

```
IF( DL( I ).EQ.ZERO ) THEN
   ...
ELSE IF( ABS( D( I ) ).GE.ABS( DL( I ) ) ) THEN
   ...
ELSE
   ...
END IF
```

This order of tests favors a diagonal matrix first, then a matrix which does not require row interchange, and last a matrix which does require row interchanges. We reordered the tests as follows:

```
IF ( ABS( D( I ) ).GE.ABS( DL( I ) ) ) THEN
   IF( D( I ).NE.ZERO ) THEN
      ...
   END IF
```

```
      ELSE
         ...
      END IF
```

The new arrangement requires more comparisons for a diagonal matrix, but fewer comparisons in the more typical case when an interchange must be done.

Several other optimizations provided further improvement to SGTTRF:

- The fill-in vector `DU2` was initialized to zero before entering the main loop.
- The last loop iteration ($i = N$) was moved outside the main loop.
- The setting of `INFO` in the case of a zero diagonal in `U` was postponed until after the main loop. This is possible because the LU factorization continues past a zero pivot.

Table 2 shows the total effect of these changes on one processor of a CRAY T94.

The Cholesky factorization SPTTRF was more difficult to improve upon because it is so simple. The inner loop in the LAPACK implementation contained only 5 instructions, one of them an IF test which checks for a zero diagonal. Unlike in the general factorization, the presence of a zero diagonal element in the Cholesky factorization is a fatal error condition, so this test could not be moved outside the loop. Unrolling the inner loop by four provided some benefit, as shown in Table 2.

	Values of N				
Version	25	50	100	200	400
LAPACK SGTTRF	13	23	45	88	176
libsci SGTTRF	10	18	33	63	124
LAPACK SPTTRF	8	14	27	52	102
libsci SPTTRF	7	12	22	42	82

Table 2: Time in microseconds for tridiagonal factorizations, CRAY T94, 1 processor

3.2.2 Tridiagonal solves

The solve routines SGTTRS and SPTTRS were redesigned as described in section 3.1, and additional optimizations were directed at the case `NRHS` = 1. Within the auxiliary routine SGTTS2@, the code to solve $Lx = b$ was simplified for the special case `NRHS` = 1 from

```
      DO 10 I = 1, N - 1
         IF( IPIV( I ).EQ.I ) THEN
            B( I+1, J ) = B( I+1, J ) - DL( I )*B( I, J )
         ELSE
            TEMP = B( I, J )
            B( I, J ) = B( I+1, J )
            B( I+1, J ) = TEMP - DL( I )*B( I, J )
         END IF
   10 CONTINUE
```

to

```
      DO 10 I = 1, N - 1
         IP = IPIV( I )
         TEMP = B( I+1-IP+I, J ) - DL( I )*B( IP, J )
         B( I, J ) = B( IP, J )
         B( I+1, J ) = TEMP
   10 CONTINUE
```

A similar trick was used when solving $L^T x = b$ for `NRHS` $= 1$. Within the auxiliary routine SPTTS2@, the solve with the bidiagonal matrix L was replaced by a call to the libsci routine FOLR (first order linear recurrence), as had been done in the libsci version of LINPACK's SPTSL.

3.2.3 Tridiagonal driver routines

The LAPACK implementation of the simple driver routines SGTSV and CGTSV solve an augmented system instead of just calling the factor and solve routines separately. We extended this idea to SPTSV and CPTSV, and added further optimizations to the `NRHS` $= 1$ case similar to those already described for SGTTRF/SGTTRS and SPTTRF/SPTTRS. Table 3 compares the times in microseconds on one processor of a CRAY T94 for solving a tridiagonal system with one right-hand side using the libsci, LAPACK, and corresponding libsci LINPACK subroutines. The libsci LAPACK routines are now faster than LINPACK in all cases except SPTSL, which outperforms SPTSV by not checking for zeroes on the diagonal during the factorization.

	Values of n				
Version	25	50	100	200	400
libsci SGTSV	16	28	52	102	201
LAPACK SGTSV	24	48	91	181	361
libsci SGTSL	18	36	67	131	264
libsci SPTSV	10	17	30	56	99
LAPACK SPTSV	14	23	42	79	156
libsci SPTSL	8	14	26	48	85
libsci CGTSV	33	64	126	249	499
LAPACK CGTSV	41	84	162	335	676
libsci CGTSL	42	80	164	321	642
libsci CPTSV	17	30	56	108	212
LAPACK CPTSV	23	39	71	135	265
libsci CPTSL	23	44	88	175	348

Table 3: Times in microseconds for tridiagonal solvers, CRAY T94, 1 processor

3.3 Sum of Squares (xLASSQ)

An important but often overlooked contribution of LAPACK is its extensive collection of auxiliary routines, some of which are general enough to be candidates for BLAS extensions. A noteworthy example is SLASSQ, which computes a scaled sum of squares, returning two constants `SCL` and `SUMSQ` such that

$$(\texttt{SCL})^2\texttt{SUMSQ} = {x_1}^2 + {x_2}^2 + \ldots + {x_n}^2 + s^2 q,$$

where s is the initial value of `SCL` and q is the initial value of `SUMSQ`. The values s and q allow SLASSQ to be used to compute a single sum of squares for a series of vectors, as is required to compute the Frobenius norm of a matrix.

SLASSQ could be used to implement the Level 1 BLAS routine SNRM2 by means of the following Fortran code:

```
SCL   = 0.0
SUMSQ = 0.0
CALL SLASSQ( N, X, INCX, SCL, SUMSQ )
SNRM2 = SCL*SQRT( SUMSQ )
```

The scaling factor SCL is the key to the safe implementation of SLASSQ; without it, the sum of squares would overflow if the magnitude of any element of x were greater than the square root of overflow, or it would underflow to zero if the magnitude of each element of x were less than the square root of underflow.

The public domain version of SLASSQ computes the scaled sum of squares by rescaling every time it finds a value in the vector whose absolute value is greater than the current value of SCL. If x is an increasing vector, it rescales with every x_i. The following Fortran fragment is equivalent to the public domain algorithm when INCX = 1:

```
DO I = 1, N
   IF( X( I ).NE.0. ) THEN
      ABSX = ABS( X( I ) )
      IF( SCL.LT.ABSX ) THEN
         SUMSQ = 1.0 + SUMSQ*( SCL/ABSX )**2
         SCL = ABSX
      ELSE
         SUMSQ = SUMSQ + ( ABSX/SCL )**2
      END IF
   END IF
END DO
```

This algorithm prevents underflow or overflow in SUMSQ by guaranteeing that it is never less than one or greater than N. However, it is a textbook example of inefficient code! The IF tests inhibit vectorization, and the divides are slow on RISC processors, guaranteeing poor performance on almost any architecture. [1]

Our implementation of SLASSQ is a two-pass algorithm which expands the permissible range of SUMSQ in order to avoid scaling in most cases. In the first pass, we compute the maximum absolute value in the vector, SMAX. If SMAX is less than $1/N$ times the square root of overflow, but greater than the square root of underflow, scaling is not necessary, and the second pass consists of an unscaled sum of squares, returning SCL $= 1.0$ and SUMSQ $= {x_1}^2 + {x_2}^2 + \ldots + {x_n}^2$. Otherwise SCL is reset to SMAX and the sum of squares is computed as SUMSQ $= (x_1/\mathtt{SCL})^2 + (x_2/\mathtt{SCL})^2 + \ldots + (x_n/\mathtt{SCL})^2$. Because it avoids scaling unless it needs to, our scaled sum of squares does not produce the same values of SCL and SUMSQ as SLASSQ, so the subroutine has been renamed SLASSQ@ in the Cray Scientific Library. An abbreviated listing of the libsci implementation of SLASSQ@ is shown in Figure 1. The thresholds for scaling have been set for 64-bit IEEE arithmetic in this version.

Table 4 compares the performance of LAPACK's SLASSQ and libsci's SLASSQ@ on a random vector, an increasing vector, and a zero vector. SLASSQ rescales the sum of squares of the random vector many times and the sum of squares of the increasing vector N times, but does not need to rescale the zero vector. SLASSQ@ does not scale any of the three sums, and in fact does not even do the sum of the zero vector because the maximum value is zero. At larger sizes, the libsci routine is 50 times faster than LAPACK on the CRAY T94.

[1] The same observation applies to the current netlib version of SNRM2.

```
      SUBROUTINE SLASSQ@( N, X, INCX, SCL, SUMSQ )
      INTEGER            INCX, N
      REAL               SCL, SUMSQ
      REAL               X( * )
      INTEGER            I, IX, IX2
      REAL               CUTHI, CUTLO, HITEST, SMAX, SQMAX
      INTRINSIC          ABS, MAX, REAL
      DATA               CUTLO / 1.00104154759155046E-146 /
      DATA               CUTHI / 9.48075190810917589E+153 /

      IF( N.LE.0 ) RETURN
      HITEST = CUTHI / REAL( N+1 )
      IF( SUMSQ.EQ.0.0 ) SCL = 1.0

      IF( INCX.EQ.1 ) THEN
*
*        Pass through once to find the maximum value in X.
*
         SMAX = ABS( X( 1 ) )
         DO 10 I = 2, N
            SMAX = MAX( SMAX, ABS( X( I ) ) )
   10    CONTINUE
         SQMAX = MAX( SUMSQ, SMAX )
*
         IF( SCL.EQ.1.0 .AND. SQMAX.GT.CUTLO .AND. SQMAX.LT.HITEST )
     $      THEN
*
*           If SCL = 1.0 and max(SUMSQ,abs(X(i))) is greater than
*           CUTLO and less than HITEST, no scaling should be needed.
*
            DO 20 I = 1, N
               SUMSQ = SUMSQ + X( I )**2
   20       CONTINUE
         ELSE IF( SMAX.GT.0.0 ) THEN
*
*           Scale by SMAX if SCL = 1.0, otherwise scale by
*           MAX( SMAX, SCL ).
*
            IF( SCL.EQ.1.0 .OR. SCL.LT.SMAX ) THEN
               SUMSQ = ( SUMSQ*( SCL / SMAX ) )*( SCL / SMAX )
               SCL = SMAX
            END IF
*
*           Add the sum of squares of values of X scaled by SCL.
*
            DO 30 I = 1, N
               SUMSQ = SUMSQ + ( X( I ) / SCL )**2
   30       CONTINUE
         END IF
      ELSE

         ... {general case of INCX is similar}

      END IF
      RETURN
      END
```

Figure 1: Two-pass implementation of SLASSQ

	Random Vector		Vector $[1,...,N]$		Zero Vector	
N	LAPACK	libsci	LAPACK	libsci	LAPACK	libsci
128	41	6	44	6	17	5
256	78	7	85	7	31	5
512	153	8	166	8	60	6
1024	302	10	329	10	116	8
2048	601	14	657	14	228	11
4096	1197	23	1308	24	452	16

Table 4: Time in microseconds for SLASSQ, CRAY T94, 1 processor

3.4 Generating Givens and Householder Transformations

Perhaps the most fundamental of the LAPACK auxiliary routines are the subroutines to compute and apply elementary Givens and Householder transformations. These subroutines are used in the factorization and reduction routines SGEQRF, SGEHRD, SSYTRD, and SGEBRD, the orthogonal transformation routines SORGQR, SORMQR, and others, the eigenvalue routines SSTEQR, SHSEQR, SBDSQR, and in many other places as well. Our evaluation of these kernels turned up several places in which the design could be simplified and the performance improved. While the performance improvements may seem small, they are significant because inlining these routines is an effective technique for optimizing other parts of LAPACK.

3.4.1 Generating a Givens rotation (SLARTG)

A Givens rotation is a rank-2 correction to the identity of the form [2]

$$G(i,k,\theta) = \begin{bmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & c & & & & s & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & -s & & & & c & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{bmatrix}$$

where $c = cos(\theta)$ and $s = sin(\theta)$ for some angle θ. Premultiplication of a vector x by $G(i,k,\theta)$ amounts to a clockwise rotation of θ radians in the (i,k) coordinate plane. If $y = G(i,k,\theta)x$, the vector y can be described by

$$y(j) = \begin{cases} cx(i) + sx(k), & j = i \\ -sx(i) + cx(k), & j = k \\ x(j), & j <> i,k \end{cases}$$

[2]This discussion borrows from Golub and Van Loan [11], but corrects the notation to match that of Wilkinson [17] and of LAPACK.

We can force $y(k)$ to be zero by choosing θ to be the angle described by the vector $[x(i), x(k)]^T$ in the (i, k) plane, which leads to the formulae

$$c = \frac{x(i)}{\sqrt{x(i)^2 + x(k)^2}}, \qquad s = \frac{x(k)}{\sqrt{x(i)^2 + x(k)^2}}$$

This is the particular form of plane rotation computed by the BLAS routine SROTG and the LAPACK auxiliary routine SLARTG.

Since a Givens rotation only modifies two elements of a vector, its action can be described by the 2-by-2 linear transformation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The algorithm used to compute c and s, given a and b, can be described as follows:

```
if b = 0
      c = 1;
      s = 0
else
      if |a| > |b|
            τ = b/a;
            c = 1/√(1 + τ²);
            s = τ * c
      else
            τ = a/b;
            s = 1/√(1 + τ²);
            c = τ * s
      end
end
```

This is approximately the algorithm used in SLARTG, except that $a = 0$ is treated as a special case, and r is computed in addition to c and s.

The LAPACK 2.0 version of SLARTG takes the additional precaution of testing the magnitudes of a and b before dividing, and rescaling them if necessary to avoid dividing by a denormalized number. Since Cray arithmetic, including Cray IEEE arithmetic on CRAY T90 and CRAY T3D/T3E systems, does not support denormalized numbers, we discarded these additional tests. Figure 2 does a side-by-side comparison of the LAPACK and libsci versions of SLARTG. Besides looking more like the mathematical algorithm, the libsci version is about 15% faster. We have found that additional speed can be gained by writing this totally scalar algorithm in C. However, the standalone performance of SLARTG is less important than having a straightforward design that lends itself to inlining.

3.4.2 Generating a Householder reflection (SLARFG)

A Householder reflection is a matrix of the form

$$H = I - \tau v v^T$$

where v is a vector and $\tau = 2/(v^T v)$ is a scalar[3]. The LAPACK auxiliary routine SLARFG generates a real elementary reflector H that reduces a real scalar α and a real vector x of length $n-1$ to a real scalar β:

$$H \begin{pmatrix} \alpha \\ x \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \end{pmatrix}$$

In order that the n-element Householder vector v may be stored in the $(n-1)$-element vector x, the first element of v is constrained to be 1. Also, the sign of the vector is chosen carefully to avoid cancellation error that would affect the orthogonality of the computed H [11]. The algorithm for computing τ, $v(2:n)$ (overwriting x), and β can be described as follows:

$$
\begin{array}{l}
\text{if } \|x\| = 0 \\
\qquad \tau = 0; \\
\qquad \beta = \alpha \\
\text{else} \\
\qquad \gamma = sign(\alpha) * \sqrt{\alpha^2 + \|x\|^2}; \\
\qquad \beta = \alpha + \gamma; \\
\qquad \tau = \beta/\gamma; \\
\qquad x = x/\beta; \\
\qquad \beta = -\gamma \\
\text{end}
\end{array}
$$

As always, the 2-norms of x and of $[\alpha, x]^T$ must be computed carefully to avoid underflow and overflow. In LAPACK, $\|x\|$ is computed by SNRM2 and $\sqrt{\alpha^2 + \|x\|^2}$ is computed by the LAPACK auxiliary routine SLAPY2, which is careful about scaling intermediate results. But then, apparently concerned that SNRM2 might be implemented without scaling, the LAPACK version adds a test to see if $|\gamma|$ is at least a factor of ϵ away from underflow. If $|\gamma|$ is less than this threshold, x is rescaled away from underflow and the 2-norms are recomputed. [4]

In libsci, SLASSQ@ is used to compute $\|x\|$ instead of SNRM2. The use of a scaled sum of squares guarantees the accuracy of $\|x\|$, so there is no need ever to rescale x. The libsci version of SLARFG is shown alongside the LAPACK version in Figure 3. The two versions are approximately the same speed except in the case where $\|x\|$ is small enough to trigger rescaling; then the libsci version is about two times faster.

3.4.3 Vectors of Givens rotations

It is worth mentioning that LAPACK also contains subroutines to generate and apply vectors of Givens rotations (xLARGV and xLARTV). These are used in the reduction of a symmetric band matrix to condensed form, because if the band is narrow enough, a single Givens rotation does not affect the entire matrix, and it may be more efficient to apply many at once. For consistency, the same algorithm used in xLARTG should also be found in xLARGV. This is true of the libsci implementation, but it was not true of LAPACK 2.0.

---

[3] Lehoucq [14] describes the algorithm used in the complex case. Our comments and modifications to SLARFG also apply to CLARFG.

[4] There is no test in LAPACK's SLARFG to see if $|\gamma|$ is within ϵ of *overflow*, because if $|\gamma|$ were not computed with scaling, it would have blown up already!

LAPACK SLARTG

```
      SUBROUTINE SLARTG( F, G, CS, SN, R )
      REAL       CS, F, G, R, SN
      LOGICAL    FIRST
      INTEGER    COUNT, I
      REAL       EPS, F1, G1, SAFMIN, SAFMN2,
     $           SAFMX2, SCALE
      REAL       SLAMCH
      EXTERNAL   SLAMCH
      SAVE       FIRST, SAFMX2, SAFMIN, SAFMN2
      DATA       FIRST / .TRUE. /

      IF( FIRST ) THEN
        FIRST  = .FALSE.
        SAFMIN = SLAMCH('S')
        EPS    = SLAMCH('E')
        SAFMN2 = SLAMCH('B')**INT(LOG(SAFMIN/
     $           EPS)/LOG(SLAMCH('B'))/2.0)
        SAFMX2 = 1.0 / SAFMN2
      END IF
      IF( G.EQ.0.0 ) THEN
        CS = 1.0
        SN = 0.0
        R = F
      ELSE IF( F.EQ.0.0 ) THEN
        CS = 0.0
        SN = 1.0
        R = G
      ELSE
        F1 = F
        G1 = G
        SCALE = MAX( ABS(F1), ABS(G1) )
        IF( SCALE.GE.SAFMX2 ) THEN
          COUNT = 0
   10     CONTINUE
          COUNT = COUNT + 1
          F1 = F1*SAFMN2
          G1 = G1*SAFMN2
          SCALE = MAX( ABS(F1), ABS(G1) )
          IF( SCALE.GE.SAFMX2 )
     $       GO TO 10
          R = SQRT( F1**2+G1**2 )
          CS = F1 / R
          SN = G1 / R
          DO 20 I = 1, COUNT
            R = R*SAFMX2
   20     CONTINUE
        ELSE IF( SCALE.LE.SAFMN2 ) THEN
          COUNT = 0
   30     CONTINUE
          COUNT = COUNT + 1
          F1 = F1*SAFMX2
          G1 = G1*SAFMX2
          SCALE = MAX( ABS(F1), ABS(G1) )
          IF( SCALE.LE.SAFMN2 )
     $       GO TO 30
```

libsci SLARTG

```
      SUBROUTINE SLARTG( F, G, CS, SN, R )
      REAL       CS, F, G, R, SN
      REAL       T, TT

      IF( G.EQ.0.0 ) THEN
        CS = 1.0
        SN = 0.0
        R = F
      ELSE IF( F.EQ.0.0 ) THEN
        CS = 0.0
        SN = 1.0
        R = G
      ELSE IF( ABS(F).GT.ABS(G) ) THEN
        T = G / F
        TT = SQRT( 1.0+T*T )
        CS = 1.0 / TT
        SN = T*CS
        R = F*TT
      ELSE
        T = F / G
        TT = SQRT( 1.0+T*T )
        SN = 1.0 / TT
        CS = T*SN
        R = G*TT
      END IF

      RETURN
      END
```

LAPACK SLARTG, cont.

```
          R = SQRT( F1**2+G1**2 )
          CS = F1 / R
          SN = G1 / R
          DO 40 I = 1, COUNT
            R = R*SAFMN2
   40     CONTINUE
        ELSE
          R = SQRT( F1**2+G1**2 )
          CS = F1 / R
          SN = G1 / R
        END IF
        IF( ABS(F).GT.ABS(G) .AND.
     $      CS.LT.0.0 ) THEN
          CS = -CS
          SN = -SN
          R = -R
        END IF
      END IF

      RETURN
      END
```

Figure 2: Comparison of LAPACK and libsci implementations of SLARTG

LAPACK SLARFG

```
      SUBROUTINE SLARFG (N,ALPHA,X,INCX,TAU)
      INTEGER    INCX, N
      REAL       ALPHA, TAU
      REAL       X( * )

      INTEGER    J, KNT
      REAL       BETA, RSAFMN, SAFMIN, XNORM
      REAL       SLAMCH, SLAPY2, SNRM2
      EXTERNAL   SLAMCH, SLAPY2, SNRM2
      EXTERNAL   SSCAL

      IF( N.LE.1 ) THEN
        TAU = 0.0
        RETURN
      END IF
      XNORM = SNRM2( N-1, X, INCX )
      IF( XNORM.EQ.0.0 ) THEN
        TAU = 0.0
      ELSE
        BETA = -SIGN( SLAPY2(ALPHA,XNORM),
     $                ALPHA)
        SAFMIN = SLAMCH('S') / SLAMCH('E')
        IF( ABS( BETA ).LT.SAFMIN ) THEN
*
*         XNORM, BETA may be inaccurate;
*         scale X and recompute them
*
          RSAFMN = 1.0 / SAFMIN
          KNT = 0
   10     CONTINUE
          KNT = KNT + 1
          CALL SSCAL( N-1, RSAFMN, X, INCX )
          BETA = BETA*RSAFMN
          ALPHA = ALPHA*RSAFMN
          IF( ABS(BETA).LT.SAFMIN ) GO TO 10
*
*         Now SAFMIN <= BETA <= 1
*
          XNORM = SNRM2( N-1, X, INCX )
          BETA = -SIGN( SLAPY2(ALPHA,XNORM),
     $                  ALPHA)
          TAU = ( BETA-ALPHA ) / BETA
          CALL SSCAL(N-1, 1.0/(ALPHA-BETA),
     $               X, INCX)
*
*         If ALPHA is subnormal, it may lose
*         relative accuracy
*
          ALPHA = BETA
          DO 20 J = 1, KNT
             ALPHA = ALPHA*SAFMIN
   20     CONTINUE
```

libsci SLARFG

```
      SUBROUTINE SLARFG (N,ALPHA,X,INCX,TAU)
      INTEGER    INCX, N
      REAL       ALPHA, TAU
      REAL       X( * )

      REAL       SCL, SUMSQ, XA, XB, XN
      REAL       SLAPY2
      EXTERNAL   SLAPY2
      EXTERNAL   SLASSQ@, SSCAL
*
*     Quick return
*
      TAU = 0.0
      IF( N.LE.1 ) RETURN
*
*     Compute the 2-norm of x
*
      SCL = 1.0
      SUMSQ = 0.0
      CALL SLASSQ@(N-1, X, INCX, SCL, SUMSQ)
      XN = SCL*SQRT( SUMSQ )
*
*     Compute the reflection if || x || > 0.
*
      IF( XN.GT.0.0 ) THEN
        XA = SIGN( SLAPY2(ALPHA,XN), ALPHA )
        XB = ALPHA + XA
        TAU = XB / XA
        CALL SSCAL( N-1, 1.0 / XB, X, INCX )
        ALPHA = -XA
      END IF

      RETURN
      END
```

LAPACK SLARFG, cont.

```
        ELSE
          TAU = ( BETA-ALPHA ) / BETA
          CALL SSCAL(N-1, 1.0/(ALPHA-BETA),
     $               X, INCX)
          ALPHA = BETA
        END IF
      END IF
      RETURN
      END
```

Figure 3: Comparison of LAPACK and libsci implementations of SLARFG

4 Modifications to LAPACK, II: Eigensystem Solving

4.1 Balancing and Back Transformation (xGEBAL and xGEBAK)

In the nonsymmetric eigenvalue problem, balancing (row or column scaling) is sometimes used to narrow the spectrum and improve convergence. The effect of this scaling on an eigenvalue λ and eigenvector x of a nonsymmetric matrix A is that the equation

$$Ax = \lambda x$$

becomes

$$DAD^{-1}Dx = \lambda Dx$$

for a nonsingular scaling matrix D. The scaled matrix $B = DAD^{-1}$ has the same eigenvalues as A, and an eigenvector y of B is related to an eigenvector x of A by the equation $Dx = y$. The subroutine to compute the scaled matrix B is called BALANC in EISPACK and SGEBAL in LAPACK, and the subroutine to do the back-transformation, that is, solve for x in the equation $Dx = y$, is called BALBAK in EISPACK or SGEBAK in LAPACK.

Balancing in the style of EISPACK proceeds by computing the row sum r and the column sum c (excluding the diagonal element) of each row/column pair in turn. When r is less than c (similarly, c is less than r) by more than a scaling constant s, then $D(i, i)$ is initialized to s. Multiplying the row by s and the column by $1/s$ changes r to $r * s$ and c to c/s, narrowing the gap between r and c by a factor of s^2. Since $r * s < c$ and $c/s > r$, scaling always brings the sums closer together. This process is repeated until r is within a factor s of c. In EISPACK, $s = 2$, guaranteeing full accuracy in the scaled matrix, while LAPACK uses $s = 10$. Scaling by a factor that is not a power of the base introduces a small relative error, but if balancing is used as a preprocessing step for another algorithm, the error should not be significant. The larger scaling factor in LAPACK brings r and c into agreement faster than EISPACK if they are many orders of magnitude apart.

The libsci version of SGEBAL, while equivalent to EISPACK except for the size of the scaling factor, has been modified into a more structured programming style. We compare the balancing portion of this subroutine to the LAPACK 2.0 version in Figure 4. The `DO WHILE` loops in the libsci version compute the scaling constant for a particular row and column using two multiplies for each factor of `SCL` – one to update the scaling factor and one to reduce the larger of the row or column sum. By contrast, the LAPACK version uses a combination of six multiplies and divides to keep track of the 1-norm of the row, the max norm of the row, the 1-norm of the column, the max norm of the column, and both the cumulative row scaling factor and the cumulative column scaling factor. We are mystified by this redundant work and could not construct an example for which it is needed.

Table 5 compares the performance of the balancing routines from EISPACK, LAPACK, and libsci for three different matrix types:

1. Random matrix (does not require scaling)
2. Matrix with row sums greater than column sums (superdiagonal is set to $1. \times 10^{100}$)
3. Matrix with column sums greater than row sums (subdiagonal is set to $1. \times 10^{100}$)

Matrices of types 2 and 3 are pathological cases designed to exercise the scaling code, and the lower operation count of the libsci version is evident here. However, libsci's SGEBAL is also three times faster than LAPACK for larger sizes when no scaling is done.

LAPACK SGEBAL

```
140 CONTINUE
    NOCONV = .FALSE.
    DO 200 I = K, L
      C = 0.0
      R = 0.0
      DO 150 J = K, L
         IF( J.EQ.I ) GO TO 150
         C = C + ABS( A( J, I ) )
         R = R + ABS( A( I, J ) )
150   CONTINUE
      ICA = ISAMAX( L, A(1,I), 1 )
      CA = ABS( A(ICA,I) )
      IRA = ISAMAX( N-K+1, A(I,K), LDA )
      RA = ABS( A(I,IRA+K-1) )
      IF(C.EQ.0.0 .OR. R.EQ.0.0) GO TO 200
      G = R / SCLFAC
      F = ONE
      S = C + R
160   IF(C.GE.G .OR. MAX(F,C,CA).GE.SFMAX2
   $   .OR. MIN(R,G,RA).LE.SFMIN2) GO TO 170
      F = F*SCLFAC
      C = C*SCLFAC
      CA = CA*SCLFAC
      R = R / SCLFAC
      G = G / SCLFAC
      RA = RA / SCLFAC
      GO TO 160
170   CONTINUE
      G = C / SCLFAC
180   IF(G.LT.R .OR. MAX(R,RA).GE.SFMAX2.OR.
   $    MIN(F,C,G,CA).LE.SFMIN2 ) GO TO 190
      F = F / SCLFAC
      C = C / SCLFAC
      G = G / SCLFAC
      CA = CA / SCLFAC
      R = R*SCLFAC
      RA = RA*SCLFAC
      GO TO 180
190   CONTINUE
      IF( (C+R).GE.0.95*S ) GO TO 200
      IF( F.LT.ONE.AND.SCALE(I).LT.ONE )
   $    THEN IF( F*SCALE(I).LE.SFMIN1 )
   $      GO TO 200
      END IF
      IF( F.GT.ONE.AND.SCALE(I).GT.ONE )
   $    THEN IF( SCALE(I).GE.SFMAX1/F )
   $      GO TO 200
      END IF
      G = ONE / F
      SCALE( I ) = SCALE( I )*F
      NOCONV = .TRUE.
      CALL SSCAL( N-K+1, G, A(I,K),LDA )
      CALL SSCAL( L, F, A(1,I), 1 )
200 CONTINUE
    IF( NOCONV ) GO TO 140
```

libsci SGEBAL

```
 110 CONTINUE
     NOCONV = .FALSE.
     DO 120 I = ILO, IHI
       F = ABS( A(I,I) )
       C = SASUM(IHI-ILO+1, A(ILO,I), 1)-F
       R = SASUM(IHI-ILO+1, A(I,ILO), LDA)-F
*
*      No need to scale if |A(I,I)|
*      dominates the row or column.
*
       IF(C.EQ.0.0 .OR. R.EQ.0.0) GO TO 120

       IF( C.LE.R ) THEN
*
*        If C <= R, compute a scaling
*        constant G for the row.
*
         F = R*SCL
         G = ONE
         DO WHILE( C.LT.F )
           F = F*SCL2
           G = G*SCL
         END DO
         F = ONE / G
       ELSE
*
*        If C > R, compute a scaling
*        constant F for the column.
*
         G = C*SCL
         F = ONE
         DO WHILE( R.LT.G )
           G = G*SCL2
           F = F*SCL
         END DO
         G = ONE / F
       END IF
*
*      Balance if C+R is reduced by 5%.
*
       IF( (C*F+R*G).LT.0.95*(C+R) ) THEN
         SCALE( I ) = SCALE( I )*F
         NOCONV = .TRUE.
         CALL SSCAL(IHI, F, A(1,I), 1)
         CALL SSCAL(N-ILO+1, G, A(I,ILO), LDA)
       END IF
 120 CONTINUE
*
*    Compute the scaling factors again
*    if any were changed.
*
     IF( NOCONV ) GO TO 110
```

Figure 4: Balancing portion of LAPACK and libsci versions of SGEBAL

N	matrix type	libsci BALANC	LAPACK SGEBAL	libsci SGEBAL
128	1	0.59	1.29	0.82
128	2	24.57	18.63	8.52
128	3	28.91	21.06	8.62
256	1	1.42	3.35	1.89
256	2	55.34	49.02	20.70
256	3	70.44	55.33	20.83
384	1	2.55	6.19	3.21
384	2	104.03	91.62	37.07
384	3	127.04	104.49	37.82
512	1	3.69	10.10	4.51
512	2	142.21	146.25	52.64
512	3	179.58	165.64	53.20
640	1	4.74	14.61	5.94
640	2	185.91	212.43	68.84
640	3	237.36	242.02	69.62
768	1	5.93	19.90	7.40
768	2	247.19	292.47	87.99
768	3	303.75	335.71	88.65
896	1	7.42	26.20	9.22
896	2	294.20	392.24	112.03
896	3	365.91	439.11	106.88
1024	1	8.65	33.32	10.59
1024	2	377.36	492.85	129.68
1024	3	435.20	529.03	127.80

Table 5: Time in milliseconds for balancing routines, CRAY T94, 1 processor

Modifications were also made to SGEBAK to inline SSWAP and SSCAL. These simple changes made the libsci version of SGEBAK about two times faster than LAPACK and about 20% faster than EISPACK's BALBAK.

4.2 Eigenvalues of a symmetric matrix (SSTEQR)

SSTEQR computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit shift QL or QR method. The closest routines in EISPACK are IMTQL1 (eigenvalues only) and IMTQL2 (eigenvalues and eigenvectors), but EISPACK utilizes only the QL variant. The symmetric QL/QR algorithm is an iterative technique that diagonalizes a tridiagonal matrix by repeated application of orthogonal transformations. In the inner loop of a QL or QR iteration step, a Givens rotation is generated and applied to the tridiagonal matrix. This rotation creates a bulge – a fill element outside the tridiagonal structure – which must be eliminated by applying several more rotations to chase the bulge up or down the matrix.

The LAPACK 2.0 version of SSTEQR generates the Givens rotations by calling the LAPACK auxiliary routine SLARTG in its two innermost loops. These calls are obvious candidates for inlining because the granularity of SLARTG is quite small. Our previous work to simplify the design of this kernel pays benefits here. We inlined SLARTG, and further optimized the inlined code by reordering the tests so that the nonzero cases of `F` and `G` are evaluated sooner, as follows:

```
IF( ABS(F).GT.ABS(G) .AND. G.NE.0.0 ) THEN
   ...
ELSE IF( ABS(G).GE.ABS(F) .AND. F.NE.0.0 ) THEN
   ...
ELSE IF( G.EQ.0.0 ) THEN
   ...
ELSE IF( F.EQ.0.0 ) THEN
   ...
END IF
```

This order of tests is preferred for the tridiagonal matrices that arise in the symmetric QL/QR algorithm, because if one of the offdiagonal elements were zero we would have detected it already.

When eigenvectors are also requested of SSTEQR, the bulge-chasing code of the QL/QR iteration step is followed by a call to SLASR to apply the sequence of rotations to the orthogonal matrix of eigenvectors. Inlining calls to SLASR further improved the performance in this case. We also split the code into two parts, one that computes eigenvalues only and one that computes both eigenvalues and eigenvectors, thereby removing the test to see if each rotation needed to be saved.

The sum of these changes improved the performance of SSTEQR so that it is comparable to that of libsci's IMTQL1 in the eigenvalue only case. When eigenvalues and eigenvectors are requested, libsci's SSTEQR outperforms libsci's IMTQL2 for $N > 100$, and is faster than the LAPACK 2.0 version for all matrix types and problem sizes. Results are shown in Table 6. We note that the unmodified IMTQL1/IMTQL2 routines from libsci did not converge for all matrix types.

	Matrix	Eigenvalues only					Eigenvalues and Eigenvectors				
Routine	Type	50	100	200	300	400	50	100	200	300	400
libsci SSTEQR	1	2.2	8.3	30	64	111	4.0	18.1	95	269	577
	2	1.7	6.3	23	46	86	3.1	13.7	74	199	457
	3	1.9	6.2	24	55	88	3.5	13.2	74	227	456
	4	1.1	3.1	11	21	38	1.9	6.5	34	86	197
LAPACK SSTEQR	1	3.0	11.1	39	84	145	7.9	30.6	130	331	683
	2	2.3	8.7	32	64	119	5.8	22.6	102	245	536
	3	2.6	8.4	32	73	118	6.6	22.0	101	281	536
	4	1.5	4.2	15	29	52	3.9	11.5	48	110	234
EISPACK IMTQL1/2	1	5.5	21.3	81	170	301	7.2	31.0	156	410	871
	2	3.1	11.1	44	94	156	4.3	17.4	98	266	566
	3	4.5	14.8	58	136	221	6.0	22.3	117	338	660
	4	2.3	7.3	26	51	92	3.0	10.4	50	124	276
libsci IMTQL1/2	1	2.0	7.7	29	61	109	3.8	17.7	105	304	682
	2	1.5	5.4	NA	NA	NA	2.8	12.1	NA	NA	NA
	3	1.6	5.6	22	53	87	3.1	12.8	82	255	526
	4	1.0	3.0	NA	NA	NA	1.8	6.3	NA	NA	NA

Table 6: Time in milliseconds for SSTEQR equivalents, CRAY T94, 1 processor

4.3 SSTERF

SSTERF computes all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. The LAPACK 2.0 version of this subroutine was 10–20% slower than its EISPACK equivalent, TQLRAT, on the CRAY T94. We achieved a slight performance improvement by peeling off the first loop iteration to avoid an IF test in the inner loop. Faced with a construct like this:

```
      DO I = M - 1, L, -1
         ...
         IF( I.NE.M-1 )
     $      E( I+1 ) = S*R
         ...
      END DO
```

we replaced it with special case code for the $I = M - 1$ loop iteration, followed by a loop from $M - 2$ down to L. Although the improvement in absolute terms was small, these changes eliminated much of the performance difference between LAPACK and EISPACK, as can be seen in Table 7.

	Matrix	Matrix size N				
Routine	Type	50	100	200	300	400
libsci SSTERF	1	1.5	5.4	19.0	40.0	69.3
	2	0.8	2.8	10.3	21.0	39.2
	3	1.2	3.8	14.2	32.5	52.5
	4	0.7	1.9	6.3	12.2	22.0
LAPACK SSTERF	1	1.6	5.7	20.0	42.2	72.7
	2	0.9	3.1	11.3	23.0	42.8
	3	1.4	4.2	15.5	35.5	57.4
	4	0.8	2.2	7.0	13.5	24.1
EISPACK TQLRAT	1	1.4	4.9	17.7	36.2	63.7
	2	0.8	2.9	10.5	21.6	38.2
	3	1.2	3.7	13.5	32.6	53.2
	4	0.7	2.2	6.9	13.5	24.3

Table 7: Time in milliseconds for PWK algorithms, Cray T94, 1 processor

4.4 SSTEIN

SSTEIN computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration. The basic outline of inverse iteration is

1. Choose a starting vector y with $\|y\|_2 = 1$.
2. Solve the tridiagonal system $(T - \lambda_j)z = y$.
3. If the reorthogonalization criterion is satisfied, orthogonalize the iterate z with respect to those previously computed eigenvectors corresponding to computed eigenvalues close to λ_j.
4. If the stopping criterion is not satisfied, set $y = x$ and repeat from Step 2.

5. Accept $z/\|z\|_2$ as the computed eigenvector.

Each of the first four steps was modified in LAPACK from its EISPACK equivalent, TINVIT, generally with good reason. However, we found the LAPACK 2.0 version of SSTEIN to be as much as six times slower than TINVIT, the biggest performance difference of any LAPACK routine.

The motivation for many of the algorithmic changes in SSTEIN compared to TINVIT was the work of Jessup [12]. Two of these, changing the stopping criterion and performing a fixed number of iterations for all eigenvectors, instead of a fixed number of iterations for each eigenvector, have already been discussed in § 2.3, and our timing comparisons have been adjusted for them (by modifying TINVIT). Other enhancements suggested by Jessup to improve the accuracy of inverse iteration were:

- Use a random starting vector for each eigenvalue, instead of a scaled vector of 1's.
- Perform an extra iteration after convergence, specifically to improve the accuracy of computed eigenvectors that satisfy the convergence criterion after only one iteration.

Additionally, SSTEIN includes yet another extra iteration after convergence and implicit row scaling in the solution of $(T - \lambda_j)z = y$. These last two features are computationally expensive, so we focused most of our attention on them.

We quickly abandoned the second of the two extra iterations after convergence. The extra iterations account for most of the performance degradation between TINVIT and SSTEIN in the case of well-separated eigenvalues, and while the case had been made for one extra iteration, there was scant evidence to justify a second. In the cases we examined, the only effect appeared to be to change the sign of the already computed eigenvector. The LAPACK functionality tests all passed with only one extra iteration.

SSTEIN calls two auxiliary routines to solve the shifted tridiagonal system: SLAGTF to compute an LU factorization of $(T - \lambda_j)$ and SLAGTS to solve the factored system with one right-hand side. Our first observation was that, before calling SLAGTF, SSTEIN makes three calls to SCOPY to initialize data for the call. These copies could be performed more efficiently inside the auxiliary routine, so we replaced SLAGTF with a new interface having separate input and output vectors. By reusing the original data for the tridiagonal matrix and making use of its symmetry, we eliminated the equivalent of two vector copies. We also removed the computation of a tolerance, which was not used, and the test for a zero subdiagonal element, which was unnecessary, and placed the case K = N-1 outside the main loop to cut the number of IF statements. The much-streamlined result retains all the functionality of the original.

LAPACK's SSTEIN calls SLAGTS with an argument specifying that if overflow would otherwise occur, the diagonal elements of U are to be perturbed. This is more rigorous than EISPACK, which only perturbs zero diagonal elements of U. We separated the L-solve and the U-solve and optimized the L-solve by unrolling, as had already been done for the tridiagonal solvers. For the U-solve, we noted that the straightforward U-solve as in TINVIT was much simpler than the code with perturbations of small diagonal elements and could be used for part of the solve, until the first perturbation were required. Also, the solves are part of an iterative method in which the number of iterations is at least two. Recalling previous work with scaled triangular solvers in the context of iterative refinement [2], we computed a growth factor for the U-solve to find the largest trailing submatrix of U that does not require any perturbations. This allowed us to replace a single call to

SLAGTS with an unperturbed solve using part of U, a series of updates, and a robust solve with only the portion of U that may require perturbations. We also simplified the test for small diagonal elements in the perturbed U-solve, from

```
      AK = A( K )
      PERT = SIGN( TOL, AK )
   40 CONTINUE
      ABSAK = ABS( AK )
      IF( ABSAK.LT.ONE ) THEN
         IF( ABSAK.LT.SFMIN ) THEN
            IF( ABSAK.EQ.ZERO .OR. ABS( TEMP )*SFMIN.GT.ABSAK ) THEN
               AK = AK + PERT
               PERT = 2*PERT
               GO TO 40
            ELSE
               TEMP = TEMP*BIGNUM
               AK = AK*BIGNUM
            END IF
         ELSE IF( ABS( TEMP ).GT.ABSAK*BIGNUM ) THEN
            AK = AK + PERT
            PERT = 2*PERT
            GO TO 40
         END IF
      END IF
      Y( K ) = TEMP / AK
```

to

```
      AK = A( K )
      IF( MAX( ABS( TEMP )*SFMIN,SFMIN ).GT.ABS( AK ) ) THEN
         PERT = SIGN( TOL, AK )
   50    CONTINUE
         AK = AK + PERT
         PERT = 2*PERT
         IF( ABS( TEMP )*SFMIN.GT.ABS( AK ) )
     $      GO TO 50
      END IF
      Y( K ) = TEMP / AK
```

Finally, we noted that in the Gram-Schmidt reorthogonalization step, it is possible to replace a loop of calls to SDOT and SAXPY with two calls to SGEMV. This is a computationally-intensive portion of SSTEIN when there are repeated eigenvalues, and the introduction of Level 2 BLAS dramatically improved the performance of this case. The original code was

```
         IF( ABS( XJ-XJM ).GT.ORTOL )
     $      GPIND = J
         IF( GPIND.NE.J ) THEN
            DO 80 I = GPIND, J - 1
               CTR = -SDOT( BLKSIZ, WORK( INDRV1+1 ), 1, Z( B1, I ), 1 )
               CALL SAXPY( BLKSIZ, CTR, Z( B1, I ), 1, WORK( INDRV1+1 ), 1 )
   80       CONTINUE
         END IF
```

and this was replaced with

```
         IF( J.GT.J1 ) THEN
            IF( ABS( XJ-XJM ).GT.ORTOL ) THEN
```

```
            GPIND = J
         ELSE
            CALL SGEMV( 'Transpose', BLKSIZ, J-GPIND, ONE, Z( B1, GPIND ),
     $                  LDZ, WORK( IX ), 1, ZERO, Z( B1, J ), 1 )
            CALL SGEMV( 'No transpose', BLKSIZ, J-GPIND, -ONE, Z( B1,
     $                  GPIND ), LDZ, Z( B1, J ), 1, ONE, WORK( IX ), 1 )
            END IF
      END IF
```

With these changes, libsci's SSTEIN is now faster than all previous versions in the clustered eigenvalue case (test matrix type 3), but is still up to 2.5 times slower than the original TINVIT in the case of well-separated eigenvalues (test matrix type 1) due to the algorithmic changes to improve accuracy. The improvements over the LAPACK 2.0 version of SSTEIN can be seen from Table 8. Note that the amount of work performed varies widely between matrix types because of the iterative nature of this algorithm.

	Matrix	Values of N				
Routine	type	50	100	200	300	400
libsci SSTEIN	1	4.5	15.9	59.3	129.	226.
	2	5.6	20.9	85.5	206.	393.
	3	6.2	22.7	94.5	230.	443.
	4	5.8	20.7	86.8	204.	386.
LAPACK SSTEIN	1	9.7	35.3	133.	293.	518.
	2	16.1	65.4	272.	654.	1220.
	3	18.7	77.4	331.	788.	1450.
	4	17.3	63.5	282.	657.	1170.
EISPACK TINVIT	1	5.1	20.1	78.8	175.	309.
	2	5.8	23.9	99.9	234.	440.
	3	8.8	43.9	217.	522.	1050.
	4	6.1	23.6	101.	235.	433.
libsci TINVIT	1	1.8	6.4	24.5	54.	94.
	2	3.7	15.6	67.2	169.	313.
	3	9.5	50.9	234.	597.	1120.
	4	4.1	15.2	67.2	168.	303.

Table 8: Time in milliseconds for symmetric inverse iteration, CRAY T94, 1 processor

4.5 SHSEQR

SHSEQR computes the Schur factorization of a Hessenberg matrix by a multiple-shift QR algorithm. The multishift scheme uses the k eigenvalues of the k-by-k trailing submatrix, which are computed using a double-shift QR algorithm as in EISPACK's HQR. The k-by-k shift creates a bulge of size k along the diagonal of the Hessenberg matrix. A series of Householder reflections are then generated and applied to chase the bulge down the matrix.

The key computational components of the multishift algorithm are forming the block Householder reflections, applying the block Householder reflections, and solving for the k eigenvalues of the k-by-k shift matrix. Each of these components was optimized to bring the performance of SHSEQR closer to that of HQR, particularly for small problem sizes where HQR was three times faster.

Much of the disparity between the performance of SHSEQR and HQR is due to the modularity of LAPACK, which adds overhead for small problem sizes. We removed some of this overhead by inlining SCOPY and SLARFG both in SHSEQR and in its auxiliary routine SLAHQR where they determine a reflection matrix. Although we would have liked to remove it, we retained the auxiliary routine SLARFX from the LAPACK distribution, which is just like SLARF but includes hand-unrolled cases for $m \leq 10$ and $n \leq 10$ to avoid two calls to Level 2 BLAS with one dimension small. We also removed the test to see if the Z matrix should be updated in the double-shift QR loop of SLAHQR and provided two separate DO loops for the eigenvalue-only and eigenvalue/eigenvector cases.

The results in Table 9 show that, while the LAPACK version of SHSEQR is always slower than HQR in the eigenvalue-only case, the libsci version is more competitive, ranging from about 15% slower to 15% faster for most matrices. The advantages of higher-level BLAS in LAPACK are more evident when the Schur form is also computed.

	Matrix	Eigenvalues only					Eigenvalues and Schur Form				
Routine	Type	50	100	200	300	400	50	100	200	300	400
libsci SHSEQR	1	9.2	49	169	409	785	11.7	67	268	719	1480
	3	11.6	42	181	404	784	14.9	58	289	711	1490
	4	10.2	40	177	401	790	13.0	55	281	709	1490
	6	9.1	38	156	361	719	11.6	51	246	630	1350
LAPACK SHSEQR	1	26.3	80	250	558	1040	30.5	98	354	873	1740
	3	25.4	74	254	573	1040	29.3	91	359	893	1740
	4	28.1	74	254	547	1050	32.0	90	458	856	1760
	6	25.3	68	227	512	913	28.9	83	322	797	1530
EISPACK HQR/2	1	8.2	40	162	420	903	14.1	73	370	1050	2320
	3	8.8	36	173	462	882	15.1	68	390	1120	2280
	4	8.6	34	171	465	887	14.8	64	383	1120	2290
	6	5.3	24	108	299	590	10.5	51	272	797	1620

Table 9: Time in milliseconds for multishift QR, CRAY T94, 1 processor

4.6 SHSEIN

SHSEIN applies inverse iteration to compute the eigenvectors of a nonsymmetric matrix that has been reduced to Hessenberg form. Most of the work of SHSEIN is contained in the auxiliary routine SLAEIN, which finds a single right or left eigenvector corresponding to a particular eigenvalues of the real Hessenberg matrix H.

One of the reasons SHSEIN is slower than its EISPACK equivalent INVIT is that SLAEIN computes the 1-norm of the offdiagonal elements and checks for possible overflow in the next step if this norm is too large. INVIT does not do this test, which was always false in our test cases. However, we were able to hide most of this extra work by computing the 1-norms "on the fly" during the LU decomposition of the Hessenberg matrix H. Also in SLAEIN, we moved the test `IF( RIGHTV ) THEN` out of the loop that solves $Ux = sv$ and provided separate code for the right and left eigenvector cases.

Table 10 shows that libsci's SHSEIN is faster than LAPACK's SHSEIN for all sizes and EISPACK's INVIT for $N > 50$, but does not beat the performance of libsci's INVIT until $N > 300$.

Routine	Values of N				
	50	100	200	300	400
libsci SHSEIN	8.2	31.9	161.	390.	751.
LAPACK SHSEIN	8.5	45.9	237.	599.	1160.
EISPACK INVIT	6.8	52.0	273.	672.	1560.
libsci INVIT	5.5	26.8	141.	345.	822.

Table 10: Time in milliseconds for nonsymmetric inverse iteration, CRAY T94, 1 processor

4.7 STGEVC

STGEVC computes some or all of the right and/or left eigenvectors of a pair of real matrices (S, P), where S is a quasi-triangular matrix and P is upper triangular. Matrix pairs of this type are produced by the generalized Schur factorization of a matrix pair (A, B):

$$A = QSZ^T, \quad B = QPZ^T$$

as computed by SGGHRD + SHGEQZ. The right eigenvector x and the left eigenvector y of (S, P) corresponding to an eigenvalue λ are defined by

$$Sx = \lambda Px, \quad y^T S = \lambda y^T P.$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks of S and P. If $s = S_{ii}$ is a 1-by-1 diagonal block of S and $p = P_{ii}$, then $\lambda = s/p$ is a generalized eigenvalue of the matrix pair (S, P). In the case of a 2-by-2 diagonal block of S, s is one of a complex conjugate pair of eigenvalues.

It is convenient to leave the eigenvalue in its quotient form and to express the generalized eigenvalue problem as

$$aSx = bPx \quad \text{or} \quad ay^T S = by^T P,$$

where a is the eigenvalue p of P, possibly rescaled, and b is the eigenvalue s of S after scaling. In solving for the right generalized eigenvector x or the left generalized eigenvector y, we must differentiate between the cases where b is real and b is complex. This is implemented in STGEVC by testing the flag `ILCPLX`, which is false for a real eigenvalue and true for a complex eigenvalue, or by use of an outer loop from 1 to `NW`, where `NW` is either 1 or 2. We found that testing the size of `NW` at an outer level, interchanging the loop from 1 to `NW` with the next innermost loop, and explicitly unrolling for the separate cases of `NW` = 1 and `NW` = 2 improved the performance of both the left and right eigenvector cases.

For example, the following code from the left eigenvector case

```
      DO 120 JW = 1, NW
        DO 110 JA = 1, NA
          SUMA(JA,JW) = ZERO
          SUMB(JA,JW) = ZERO
          DO 100 JR = JE, J - 1
            SUMA(JA,JW) = SUMA(JA,JW) + A(JR,J+JA-1)*WORK((JW+1)*N+JR)
            SUMB(JA,JW) = SUMB(JA,JW) + B(JR,J+JA-1)*WORK((JW+1)*N+JR)
  100     CONTINUE
  110   CONTINUE
  120 CONTINUE
      DO 130 JA = 1, NA
        IF( ILCPLX ) THEN
```

```
          SUM(JA,1) = -ACOEF*SUMA(JA,1) + BCOEFR*SUMB(JA,1) -
     $                BCOEFI*SUMB(JA,2)
          SUM(JA,2) = -ACOEF*SUMA(JA,2) + BCOEFR*SUMB(JA,2) +
     $                BCOEFI*SUMB(JA,1)
        ELSE
          SUM(JA,1) = -ACOEF*SUMA(JA,1) + BCOEFR*SUMB(JA,1)
        END IF
  130 CONTINUE
```

was replaced with

```
      IF( ILCPLX ) THEN
        DO 110 JA = 1, NA
          SUMA(JA,1) = ZERO
          SUMA(JA,2) = ZERO
          SUMB(JA,1) = ZERO
          SUMB(JA,2) = ZERO
          DO 100 JR = JE, J - 1
            SUMA(JA,1) = SUMA(JA,1) + A(JR,J+JA-1)*WORK(2*N+JR)
            SUMA(JA,2) = SUMA(JA,2) + A(JR,J+JA-1)*WORK(3*N+JR)
            SUMB(JA,1) = SUMB(JA,1) + B(JR,J+JA-1)*WORK(2*N+JR)
            SUMB(JA,2) = SUMB(JA,2) + B(JR,J+JA-1)*WORK(3*N+JR)
  100     CONTINUE
          SUM(JA,1) = -ACOEF*SUMA(JA,1) + BCOEFR*SUMB(JA,1) -
     $                BCOEFI*SUMB(JA,2)
          SUM(JA,2) = -ACOEF*SUMA(JA,2) + BCOEFR*SUMB(JA,2) +
     $                BCOEFI*SUMB(JA,1)
  110   CONTINUE
      ELSE
        DO 130 JA = 1, NA
          SUMA(JA,1) = ZERO
          SUMB(JA,1) = ZERO
          DO 120 JR = JE, J - 1
            SUMA(JA,1) = SUMA(JA,1) + A(JR,J+JA-1)*WORK(2*N+JR)
            SUMB(JA,1) = SUMB(JA,1) + B(JR,J+JA-1)*WORK(2*N+JR)
  120     CONTINUE
          SUM(JA,1) = -ACOEF*SUMA(JA,1) + BCOEFR*SUMB(JA,1)
  130   CONTINUE
      END IF
```

The effect of this change was dramatic in the left eigenvector case, improving the performance by a factor of three, as shown in Table 11.

A further optimization in the back transformation step of the right eigenvector case involved replacing two loop nests with a call to SGEMV from the Level 2 BLAS. The original code was

```
      DO 410 JW = 0, NW - 1
        DO 380 JR = 1, N
          WORK((JW+4)*N+JR) = WORK((JW+2)*N+1)*VR(JR,1)
  380   CONTINUE
        DO 400 JC = 2, JE
          DO 390 JR = 1, N
            WORK((JW+4)*N+JR) = WORK((JW+4)*N+JR) +
     $                          WORK((JW+2)*N+JC)*VR(JR,JC)
  390     CONTINUE
  400   CONTINUE
  410 CONTINUE
      DO 430 JW = 0, NW - 1
```

```
      DO 420 JR = 1, N
        VR(JR,IEIG+JW) = WORK((JW+4)*N+JR)
420   CONTINUE
430 CONTINUE
```

and this was replaced with

```
    IF( ILCPLX ) THEN
       CALL SGEMV( 'N', N, JE, ONE, VR(1,1), LDVR,
   $               WORK(2*N+1), 1, ZERO, WORK(4*N+1), 1 )
       CALL SGEMV( 'N', N, JE, ONE, VR(1,1), LDVR,
   $               WORK(3*N+1), 1, ZERO, WORK(5*N+1), 1 )
       DO 420 JR = 1, N
          VR(JR,IEIG) = WORK(4*N+JR)
          VR(JR,IEIG+1) = WORK(5*N+JR)
420    CONTINUE
    ELSE
       CALL SGEMV( 'N', N, JE, ONE, VR(1,1), LDVR,
   $               WORK(2*N+1), 1, ZERO, WORK(4*N+1), 1 )
       DO 430 JR = 1, N
          VR(JR,IEIG) = WORK(4*N+JR)
430    CONTINUE
    END IF
```

Lastly, SLALN2 was inlined for the real case but not for the complex case.

Table 11 shows the performance of the libsci and LAPACK versions of STGEVC and the EISPACK equivalent QZVEC when all eigenvectors are computed and back-transformed. The factor of three improvement of libsci over LAPACK in the left eigenvector case has already been noted. In the right eigenvector case, which is the only case computed by EISPACK, the libsci version of STGEVC is comparable to QZVEC for small problems and faster for larger values of N.

	Left eigenvectors					Right eigenvectors				
Routine	50	100	200	300	400	50	100	200	300	400
libsci STGEVC	5.6	23	94	223	420	4.0	16	67	160	306
LAPACK STGEVC	8.8	42	216	613	1310	5.1	20	81	193	363
EISPACK QZVEC						3.5	16	79	200	382

Table 11: Time in milliseconds for STGEVC vs. QZVEC, CRAY T94, 1 processor

4.8 Miscellaneous Inlining

Many other LAPACK computational routines saw performance improvements on the CRAY T94 from selective inlining of BLAS or LAPACK auxiliary routines. These include

- SGEHD2: Inlined SLARF
- SSYTD2: Inlined SLARFG
- SGEBD2: Inlined SLARF, SLARFG
- SGGHRD: Inlined SROT
- STREVC: Inlined SAXPY, SDOT

- SBDSQR: Inlined SLASR
- SHGEQZ: Inlined SLARTG, SROT, SLARFG

Performance improvements ranged from 2% for larger sizes of SGEHRD to 40% for SGGHRD and SHGEQZ.

5 Conclusion

Several of the performance improvements to LAPACK described in this report take advantage of Cray architectural and software features:

- Cray multitasking software was used to reimplement the linear system solve routines with higher-level parallelism and better single processor performance. The LAPACK designers had avoided the use of explicit parallelism because there was no portable way to express it.
- The absence of denormalized numbers in Cray IEEE arithmetic (or, for that matter, in traditional Cray arithmetic) led to simpler designs for key kernel routines such as SLARTG and SLARFG.
- Cray's advanced compiler technology – and relatively high subroutine overhead on nearly 2 Gflop processors – combined to give a significant advantage to inlining.

Other optimizations in this report are algorithmic and would benefit architectures other than Cray's:

- Our two-pass algorithm for the sum of squares requires fewer operations and admits better compiler optimization than rescaling at every step as in LAPACK.
- Using two multiplies per scaling step of the balancing routine, as in EISPACK's BALANC or libsci's SGEBAL, is naturally faster than the six operations in each step of the LAPACK 2.0 version.
- Limiting the number of extra iterations in SSTEIN to one and avoiding unnecessary scaling save time by reducing the amount of work to be done.
- Introducing Level 2 BLAS calls in SSTEIN and STGEVC takes advantage of more efficient library routines, correcting apparent oversights in the LAPACK design.
- Changing the order of tests in the tridiagonal routines, combining the factor and solve in SPTSV and CPTSV for `NRHS` = 1, moving IF tests out of inner loops, and unrolling loops that only go from 1 to 2 are good programming practices even if their benefit on other architectures is not as great.

Many of these suggestions have been communicated to the LAPACK team and may already be on their way to being implemented as LAPACK continues to evolve.

A Setting Block Sizes

The LAPACK developers left the tuning of block algorithms to the implementors via an auxiliary routine ILAENV. Many hours of dedicated Cray time were consumed in fine-tuning the block size of the LAPACK routines, although in retrospect this was not a very effective use of resources. However, we did discover a simple formula for the optimal block size on one processor.

It was empirically observed that the vector length `VLEN` is often a good choice of block size for sufficiently large problems on Cray vector machines. The vector length also figures prominently in the cutting strategy used in the highly optimized Cray BLAS. In the BLAS, vector operations that are too long for the vector registers are subdivided into a minimal number of approximately equal parts. Equipartitioning is preferred because the vector processors have multiple vector units which can operate in parallel, so load balancing is an issue even on a single processor.

Carrying this rationale to a higher level, we can assist the BLAS in creating regular partitions by choosing a block size that is tuned to the number of vector segments in one of the matrix dimensions. The formula is

$$\mathrm{NB} = \lceil n / \lceil n/\mathtt{VLEN} \rceil \rceil .$$

In Fortran, this is coded as

```
NTMP = ( N+VLEN-1 ) / VLEN
NB = ( N+NTMP-1 ) / NTMP
```

For example, if $N = 300$ and $\mathtt{VLEN} = 128$, the problem must be divided into at least 3 parts, so $\mathtt{NB} = 300/3 = 100$. The formula increases the blocksize until at $N = 384$ we have $\mathtt{NB} = 128$. At $N = 385$ the blocksize resets to $\mathtt{NB} = 97$ and begins increasing again until it reaches 128 at $N = 512$. Table 12 illustrates the minor variations in speed of the libsci routine SGETRF near the optimal blocksize of 100 at $N = 300$ and $N = 400$.

NB	N = 300	N = 400
64	1176.5	1291.7
92	1177.2	1293.1
100	1179.7	1294.7
108	1179.4	1292.2
128	1178.6	1291.7

Table 12: Speed of SGETRF in megaflops

Unfortunately, the choice of block size has resisted our attempts to fit into a formula except in the single-processor case. However, empirical data suggests that deciding whether or not to use blocking at all is the key point, and the performance varies only slightly among a range of block sizes.

References

[1] 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*. SIAM, Philadelphia, 1992.

[2] Edward Anderson. Robust triangular solves for use in condition estimation. LAPACK Working Note 36, Technical Report CS-91-142, University of Tennessee, Aug. 1991.

[3] Edward Anderson and Jack Dongarra. Evaluating block algorithm variants in LAPACK. In Jack Dongarra et al., editors, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 3–8. SIAM, Philadelphia, 1990. (also LAPACK Working Note 19).

[4] Edward Anderson, Jack Dongarra, and Susan Ostrouchov. Installation guide for LAPACK. LAPACK Working Note 41, Technical Report CS-91-138, University of Tennessee, Feb. 1992.

[5] Cray Research, Eagan, Minnesota. *Scientific Libraries Reference Manual (SR-2081)*, 1997.

[6] Michel J. Daydé and Iain S. Duff. Use of level 3 BLAS in LU factorization on the CRAY-2, the ETA-10P, and the IBM 3090-200/VF. Technical Report CSS 229, Harwell Laboratory, Oct. 1988.

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

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[10] K. A. Gallivan and A. H. Sameh. Matrix computation on shared-memory multiprocessors. CSRD Report 760, Center for Supercomputing Research and Development, University of Illinois, April 1988.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, second edition, 1989.

[12] I. C. F. Ipsen and E. R. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comput., Vol. 11, No. 2*, pages 203–229, 1990.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[14] R. B. Lehoucq. The computation of elementary unitary matrices. LAPACK Working Note 72, Technical Report CS-94-233, University of Tennessee, April 1994.

[15] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[16] B. T. Smith et al. *Matrix Eigensystem Routines – EISPACK Guide*. Lecture Notes in Computer Science 6. Springer-Verlag, New York, second edition, 1976.

[17] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, England, 1965.