Efficient Computation of the Singular Value Decomposition with Applications to Least Squares Problems

Ming Gu* James Demmel† Inderjit Dhillon‡

October 1, 1994

Abstract

We present a new algorithm for computing the singular value decomposition (SVD) of a matrix. The algorithm is based on using divide-and-conquer to compute the SVD of a bidiagonal matrix. Compared to the previous algorithm (based on QR-iteration) the new algorithm is at least 9 times faster on bidiagonal matrices of dimension $n = 400$, when running on a DEC Alpha with optimized BLAS. The speedup increases with dimension n. For the dense singular value decomposition, the speedup ranges from 2.2 to 3.9 for $n = 400$. When used to solve dense, square linear least squares problems, the operation count drops from $12n^3$ to $\frac{8}{3}n^3$, and the speedup ranges from 2.3 to 3.8 for $n = 400$. This means using the SVD for the least squares problem averages only 4.8 times slower than using simple QR decomposition, whereas it used to be over 15 times slower. We show how to modify the old least squares solver based on the SVD with QR-iteration to attain slightly better speedup, at the cost of $O(n^2)$ storage. This makes the SVD a much more economical tool than it was before.

*Department of Mathematics and Lawrence Berkeley Laboratory, University of California, Berkeley, CA 94720. The author was supported in part by the Applied Mathematical Sciences Subprogram of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

†Computer Science Division and Mathematics Department, University of California, Berkeley, CA 94720. The author was supported in part by NSF grant ASC-9005933, ARPA contact DAAL03-91-C-0047 via a subcontract from the University of Tennessee, ARPA grant DM28E04120 via a subcontract from Argonne National Laboratory, and DOE grant DE-FG03-94ER25206.

‡Computer Science Division, University of California, Berkeley, CA 94720. The author was supported by ARPA contact DAAL03-91-C-0047 via a subcontract from the University of Tennessee, and DOE grant DE-FG03-94ER25206.

1 Introduction

Given a matrix $A \in \mathbf{R}^{m\times n}$ with $m \geq n$, the *singular value decomposition* (SVD) of A is defined as

$$A = X \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} Y^T , \tag{1.1}$$

where $X \in \mathbf{R}^{m\times m}$ and $Y \in \mathbf{R}^{n\times n}$ are orthogonal matrices, and $\Sigma \in \mathbf{R}^{n\times n}$ is a non-negative diagonal matrix. The columns of X and Y are the *left singular vectors* and the *right singular vectors* of A, respectively, and the diagonal entries $\sigma_1, \cdots, \sigma_n$ of Σ are the *singular values* of A.

One of the many applications of the SVD is to the solution of the *linear least squares problem*:

$$\min_{x\in\mathbf{R}^n} \|Ax - b\|_2 ,$$

where $b \in \mathbf{R}^m$ is a given vector. It is well-known that in the case where A has full column rank (i.e., Σ is invertible), the solution to the above problem is

$$x_{LS} = Y \begin{pmatrix} \Sigma^{-1} & 0 \end{pmatrix} X^T b . \tag{1.2}$$

When A is rank deficient, the above problem is usually solved with additional constraints, which may be easily expressed by replacing Σ^{-1} by its pseudo-inverse or other approximate inverse. The SVD is also a very useful tool for other constrained least squares problems [13].

Of the various technique for solving rank deficient and constrained least squares problems, the SVD is considered the most reliable. Unfortunately, it is also the most expensive. When $m = n$, the SVD based on QR-iteration takes $12n^3$ floating point operations (flops) on average, whereas QR decomposition takes only $\frac{4}{3}n^3$ flops, 9 times fewer [13, p. 248]. In fact, on current computer architectures with steep memory hierarchies, the using the SVD may take over 15 times longer than QR decomposition. This is because the QR decomposition algorithm can be reorganized to exploit the memory hierarchy [3], but the conventional SVD algorithm is much less amenable to this reorganization.

The SVD is usually computed in two phases:

Phase I: Use orthogonal transformations to reduce A to an upper bidiagonal matrix:

$$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} B \\ 0 \end{pmatrix} V^T, \tag{1.3}$$

where $\begin{pmatrix} U_1 & U_2 \end{pmatrix} \in \mathbf{R}^{m\times m}$ and $V \in \mathbf{R}^{n\times n}$ are orthogonal matrices, with $U_1 \in \mathbf{R}^{m\times n}$ and $U_2 \in \mathbf{R}^{m\times(m-n)}$, and

$$B = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & \alpha_n \end{pmatrix}$$

is an upper bidiagonal matrix.

Phase II: Compute the SVD of B:

$$B = Q\Sigma W^T \tag{1.4}$$

where Q and W are orthogonal matrices.

The SVD of A is then computed as

$$A = (U_1 Q \quad U_2) \begin{pmatrix} \Sigma \\ 0 \end{pmatrix} (VW)^T$$

Phase II has previously been implemented using QR-iteration [11, 12, 8] or qd iteration [22, 24]. This has been the bottleneck of the overall algorithm, taking up to 80% of the total time.

We have made three contributions toward overcoming this bottleneck. First, we have implemented a variation of the bidiagonal divide-and-conquer algorithm (BDC) of Gu and Eisenstat [17], which is based on previous work by Arbenz and Golub [2], Cuppen [6], Golub [14], Gu and Eisenstat [15], and Jessup and Sorensen [19], for computing the SVD of B. Our numerical experiments indicate that our implementation of BDC averages 9-10 times faster for $n = 400$ than the LAPACK implementation `SBDSQR` [1] of the traditional QR based algorithm. This implementation, combined with LAPACK routines for Phase I and the rest of Phase II, is from 2.3 to 3.9 times faster than the corresponding LAPACK implementation [1] for computing the full SVD of A when $n = 400$.

Our second contribution is a "factored form" version of the BDC, which allows us to compute the SVD of B in $O(n^2)$ flops by representing Q and W as products of $O(\log_2 n)$ certain structured orthogonal matrices. Once A has been reduced to upper bidiagonal form (Phase I), this new version of BDC allows us to finish the rest of the computation in (1.2) in $O(mn)$ flops. Since the cost of Phase I is about $4mn^2 - 4n^3/3$ flops, about twice the cost of computing a QR factorization on A, our result means that the flop count of the SVD based least squares solver is only about twice that of the QR based solver. The "factored form" version is also useful for the case where the least squares solution is subject to some simple constraints [13].

Third, we implement another technique for representing the SVD in factored form, originally suggested in [5], and also known to Rutishauser in the context of Jacobi's method [23]. The idea is to store all the Givens rotation produced during the bidiagonal QR iteration and apply them directly to the solution vector, rather than accumulating them. This simple change to the current LAPACK routine for solving the least squares problem with the SVD, also reduces the flop count to just twice that of QR decomposition, but at the cost of $O(n^2)$ storage.

Based purely on operation counts, we expect either of our two least squares algorithms to take only about twice as long as the fastest method (QR decomposition).

This is in fact nearly the case on the DEC Alpha when we use Fortran implementations of the Basic Linear Algebra Subroutines, or BLAS [10, 9], but not when we use the BLAS optimized especially for the DEC Alpha. This is because the QR decomposition can be reorganized to do almost all its floating point operations by calls to Level 3 BLAS, whereas Phase I of the SVD does half its flops in the Level 3 BLAS and half in Level 2 BLAS. In Fortran, the Level 2 and 3 BLAS are comparable in speed; the optimized Level 3 BLAS are up to 4 times faster than the Level 2 BLAS. Thus, using optimized BLAS, the new SVD based least squares solvers are about 4.4 times slower than QR decomposition for $n = 400$, not twice as slow.

It may be possible to break the "BLAS 2" barrier in reduction to bidiagonal form by exploiting successive band reduction techniques proposed for the symmetric eigenproblem [4], but we have not yet pursued this.

The rest of the paper is organized as follows: Section 2 describes the basic idea of BDC and its "factored form" version, and shows how to use them to compute the full SVD and the least squares solution. Section 3 presents our numerical results. And Section 4 summarizes our conclusions.

2 Solving the full SVD and the least squares problem using bidiagonal divide-and-conquer (BDC)

In section 2.1 we outline the bidiagonal divide-and-conquer algorithm. In section 2.2 we show how to use it to solve the least squares problem quickly. And in Section 2.3 we show how to solve the least squares problem quickly by modifying the conventional SVD solution based on QR-iteration.

2.1 BDC and its "factored form" version

BDC recursively divides B into two subproblems as follows[1]:

$$B = \begin{pmatrix} B_1 & 0 \\ \alpha_k e_k & \beta_k e_1 \\ 0 & B_2 \end{pmatrix} , \tag{2.5}$$

where $B_1 \in \mathbf{R}^{(k-1)\times k}$ and $B_2 \in \mathbf{R}^{(n-k)\times(n-k)}$ are upper bidiagonal matrices, and e_j is the j-th unit vector of appropriate dimension. We take $k = \lfloor n/2 \rfloor$.

Assume that we are given the SVDs of B_1 and B_2:

$$B_1 = Q_1 (D_1 \quad 0) W_1^T \quad \text{and} \quad B_2 = Q_2 D_2 W_2^T ,$$

[1]This is actually the dividing strategy used in [2]; BDC in [17] takes out a column (instead of a row) of B at a time.

where Q_i and W_i are orthogonal matrices of appropriate dimensions, and the D_i are non-negative diagonal matrices. Let $(l_1^T \quad \lambda_1)$ be the last row of W_1, and let f_2^T be the first row of W_2. Plugging these into (2.5), we get

$$B = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} D_1 & 0 & 0 \\ \alpha_k l_1^T & \alpha_k \lambda_1 & \beta_k f_2^T \\ 0 & 0 & D_2 \end{pmatrix} \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix}^T . \tag{2.6}$$

Note that the middle matrix is quite simple in that its entries can be non-zero only on the diagonal and in the k-th row. We will discuss the computation of its SVD later in this section. Let $S\Sigma G^T$ be the SVD of the middle matrix. Plugging it into (2.6), we get the SVD of B as (see (1.4))

$$B = Q\Sigma W^T$$

with

$$Q = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} S \quad \text{and} \quad W = \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix} G .$$

To compute the SVDs of B_1 and B_2, this process can be recursively applied until the sizes of the subproblems are sufficiently small[2]. These small subproblems are then solved using a QR type algorithm (`SBDSQR` in LAPACK). There can be at most $O(\log_2 n)$ levels of recursion.

BDC also has a recursion for computing just the singular values. Let f_1^T be the first row of W_1; let l_2^T be the last row of W_2; and let f^T and l^T be the first and last rows of W, respectively. Suppose that D_i, f_i, l_i, and λ_1 are given for $i = 1, 2$. Then we can compute Σ, f, and l by computing the SVD of the middle matrix in (2.6) as $S\Sigma G^T$, and computing

$$f^T = (f_1^T \quad 0)\, G \quad \text{and} \quad l^T = (0 \quad l_2^T)\, G \ .$$

The "factored form" version of BDC is based on the singular value recursion. We store S and G for each subproblem in the recursion, and never explicitly form any Q and W at any level, except the bottom level where we use a QR type algorithm.

In order to compute the SVD of the middle matrix in (2.6), we note that, by permuting the k-th row and column to the first row and column, this matrix can be written as

$$M = \begin{pmatrix} z_1 & z_2 & \cdots & z_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix} , \tag{2.7}$$

[2]Strictly speaking, this process is not quite recursive since, unlike B, B_1 is not a square matrix. This is true for the following singular value recursion also. See [17] for the complete recursions.

where d_i's are the diagonal elements of D_1 and D_2; and z_i's are entries of the k-th row of the middle matrix, with z_1 being the (k,k) entry. We permute the matrix M so we can write $D = \mathrm{diag}(d_1, d_2, \ldots, d_n)$ with[3] $0 \equiv d_1 \leq d_2 \leq \ldots \leq d_n$, and $z = (z_1, z_2, \ldots, z_n)^T$. We further assume that

$$d_{j+1} - d_j \geq \tau \|M\|_2 \quad \text{and} \quad |z_j| \geq \tau \|M\|_2 \;, \tag{2.8}$$

where τ is a small multiple of ϵ specified in [17]. Any matrix of the form (2.7) can be reduced to one that satisfies these conditions by the *deflation* procedure described in [17].

The following lemma characterizes the singular values and singular vectors of M.

Lemma 1 (Jessup and Sorensen [18]) *Let $S\Sigma G^T$ be the* SVD *of M with*

$$S = (s_1, \ldots, s_n) \quad , \quad \Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n) \quad \textit{and} \quad G = (g_1, \ldots, g_n) \;,$$

where $0 < \sigma_1 < \ldots < \sigma_n$. Then the singular values $\{\sigma_i\}_{i=1}^n$ satisfy the interlacing property

$$0 = d_1 < \sigma_1 < d_2 < \ldots < d_n < \sigma_n < d_n + ||z||_2 \quad ,$$

and the secular equation

$$f(\sigma) = 1 + \sum_{k=1}^{n} \frac{z_k^2}{d_k^2 - \sigma^2} = 0 \;.$$

The singular vectors satisfy

$$s_i = \left(-1, \frac{d_2 z_2}{d_2^2 - \sigma_i^2}, \ldots, \frac{d_n z_n}{d_n^2 - \sigma_i^2} \right)^T \Bigg/ \sqrt{1 + \sum_{k=2}^{n} \frac{(d_k z_k)^2}{(d_k^2 - \sigma_i^2)^2}} \quad , \tag{2.9}$$

$$g_i = \left(\frac{z_1}{d_1^2 - \sigma_i^2}, \ldots, \frac{z_n}{d_n^2 - \sigma_i^2} \right)^T \Bigg/ \sqrt{\sum_{k=1}^{n} \frac{z_k^2}{(d_k^2 - \sigma_i^2)^2}} \quad . \tag{2.10}$$

On the other hand, given D and all the singular values, we can construct a matrix with the same structure as (2.7).

Lemma 2 (Gu and Eisenstat [17]) *Given a diagonal matrix $D = \mathrm{diag}(d_1, d_2, \ldots, d_n)$ and a set of numbers $\{\hat{\sigma}_i\}_{i=1}^n$ satisfying the interlacing property*

$$0 \equiv d_1 < \hat{\sigma}_1 < d_2 < \ldots < d_n < \hat{\sigma}_n \;, \tag{2.11}$$

there exists a matrix

$$\hat{M} = \begin{pmatrix} \hat{z}_1 & \hat{z}_2 & \cdots & \hat{z}_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}$$

[3] d_1 is introduced to simplify the presentation.

whose singular values are $\{\hat{\sigma}_i\}_{i=1}^n$. *The vector* $\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_n)^T$ *is determined by*

$$|\hat{z}_i| = \sqrt{(\hat{\sigma}_n^2 - d_i^2) \prod_{k=1}^{i-1} \frac{(\hat{\sigma}_k^2 - d_i^2)}{(d_k^2 - d_i^2)} \prod_{k=i}^{n-1} \frac{(\hat{\sigma}_k^2 - d_i^2)}{\left(d_{k+1}^2 - d_i^2\right)}}\,, \tag{2.12}$$

where the sign of $\hat{z}_i$ *can be chosen arbitrarily.*

We use the root-finder provided by R.-C. Li [21] to find approximate singular values $\{\hat{\sigma}_k\}_{k=1}^n$. Following [17], we then compute $\{\hat{z}_k\}_{k=1}^n$ by using (2.12) and compute the left and right singular vectors of M using (2.9) and (2.10), except we replace z_k by $\hat{z}_k$ using the sign of z_k. It has been shown [17] that this procedure is numerically stable, provided that one computes the differences $d_i - d_j$ to high relative accuracy, for $1 \le i \le j \le n$. This assumption is automatically satisfied on most modern computers except some earlier Cray machines (Cray XMP, YMP, C90 and 2) which do not have a guard digit. We overcome this difficulty by using the following technique provided by Kahan [20]. Before the singular values are computed, we first compute

$$d_i := (d_i + d_i) - d_i \quad \text{for} \quad i = 1, \ldots, n\ .$$

On machines with a guard digit, this does not change d_i at all (barring overflow), but it chops off the last bit of d_i on the the above mentioned Cray machines. After doing so, the differences $d_i - d_j$ can be computed to high relative accuracy even on these machines. To the best of our knowledge, our code should work on any commercially significant modern North America computers.

Since S and G are generally dense matrices, storing them explicitly will take $O(n^2)$ storage for the whole recursion. However, we note that they can be reconstructed from $\{\hat{z}_k\}_{k=1}^n$, $\{\hat{\sigma}_k\}_{k=1}^n$, and $\{d_k\}k = 1^n$ whenever they are needed[4]. Hence in our implementation, we store these data rather than S and G themselves. This increases the cost of BDC by $O(n^2)$ overall, but reduces the memory requirement from $O(n^2)$ to $O(n \log_2 n)$, since there are $O(\log_2 n)$ levels of recursion.

The subroutine for the bidiagonal SVD is called SBDSDC, and the dense SVD routine that calls it is called SGESBD-DC; these names will be used in section 3.

2.2 Solving the full SVD and the least squares problem

In the current version of LAPACK, in Phase I of the SVD computation (see Section 1), the matrices $(U_1 \quad U_2)$ and V are generated as products of Householder transformations, and in Phase II, the matrices Q and W are generated as products of Givens rotations. When the full SVD of A is desired, $(U_1 \quad U_2)$ and V are explicitly computed and the Givens rotations in Q and W are applied to U_1 and V as soon as they are generated.

[4]The actual implementation is slightly more complicated for efficiency and stability reasons.

When the least squares solution (1.2) is desired, it is then computed as

$$x_1 \equiv U_1^T b \quad , \quad x_2 \equiv Q^T x_1 \quad , \quad x_3 \equiv \Sigma^{-1} x_2 \quad \text{and} \quad x_{LS} = (VW) x_3 \; , \tag{2.13}$$

where x_1 is computed by applying the Householder transformations directly to b, x_2 is computed by applying the Givens rotations directly to x_1, and x_{LS} is computed by explicitly forming the matrix VW, as is done in the dense SVD case, and then applying VW to x_3. We note that computing VW takes $O(n^3)$ flops in general.

In contrast, for the full SVD computation, we first compute the matrices Q and W explicitly, by organizing the computation to use level 3 BLAS as much as possible, and then compute $U_1 Q$ and VW by applying the sequence of Householder transformations to Q and W, respectively. This approach is similar to that used for computing the full eigendecomposition of a dense symmetric matrix by using Cuppen's divide-and-conquer algorithm [25]. The current implementation requires $3n^2 + O(n)$ workspace in order to use the level-3 BLAS. We have developed some techniques to reduce this workspace requirement to $O(n\text{NB})$, where NB is the *block size*, a machine dependent parameter that balances the efficiency of level-3 BLAS and workspace requirement. We plan to use these techniques in a future version of our implementation.

To compute the least squares solution (1.2), we use the "factored form" version of BDC. After Phase I, A is reduced to the upper bidiagonal matrix B, with the orthogonal transformations $(U_1 \quad U_2)$ and V returned as products of Householder transformations. We then compute x_1 as in (2.13). This can be done in $O(mn)$ flops [13]. To compute x_2, we note that Q is represented as a product of $O(\log_2 n)$ orthogonal matrices, the i-th of which is block diagonal with the diagonal blocks being 1's and left singular vector matrices on the i-th level in the recursion. Since there are 2^{i-1} submatrices on the i-th level with each submatrix having size $O(n/2^{i-1})$, the cost of applying the transposes of these matrices to a vector is

$$O\left(\left(n/2^{i-1}\right)^2\right) \times 2^{i-1} = O\left(n^2/2^{i-1}\right) \; ,$$

summing all these costs up, the cost for computing $x_2 = Q^T x_1$ is

$$\sum_{i=1}^{O(\log_2 n)} O\left(n^2/2^{i-1}\right) = O(n^2)$$

flops. Computing x_3 takes $O(n)$ flops. To compute x_{LS} from x_3, we do not explicitly form VW. Instead, we compute

$$x_4 \equiv W x_3 \quad \text{and} \quad x_{LS} = V x_4 \; . \tag{2.14}$$

By the same argument as above, x_4 can be computed in $O(n^2)$ flops. Finally, it is again well known that computing x_{LS} as $V x_4$ takes $O(n^2)$ flops [13]. Overall, computing x_{LS} after Phase I takes $O(mn)$ flops.

The routine for solving the least squares problem using divide-and-conquer is called SGELSD; this name will be used in section 3.

2.3 A Fast SVD Least Squares Solver Based on QR Iteration

It turns out that explicit computation of VW can be avoided even with the QR based SVD algorithms, as originally noted in [5]. Instead of computing x_{LS} as $(VW)x_3$, we can again compute x_{LS} as in (2.14). x_4 can be computed by saving all $O(n^2)$ Givens rotations performed in computing the SVD of B, and applying them to x_3 in reverse order; x_{LS} can then be computed as Vx_4 as above. Let t be the total number of such Givens rotations. Then the cost of computing x_{LS} after Phase I is $O(mn + t)$ flops. Since we usually expect $t = O(n^2)$, this cost is again $O(mn)$ flops. One drawback with this approach, however, is that it requires $O(t)$ storage, and we cannot bound t exactly beforehand.

The routine implementing this idea is called SGELSS-QRf (for "QR iteration, factored"); this name will be used in section 3. The existing LAPACK routine will be called SGELSS-QR.

3 Numerical Experiments

We ran the following experiments on a Dec Alpha 3000/500X with a 200Mhz clock, 8 KByte first level cache and 512 KByte second level cache. The optimized BLAS were those in DEC's mathematical software library dxml. We compiled using f77 with the -O optimization option. All experiments were run in single precision, i.e. 32-bit, IEEE floating point arithmetic. We let $\epsilon = 2^{-23}$ denote the machine precision.

Table 1 lists the names of the subroutines we test and what they do. The reader may want to refer to this table to interpret the following performance tables.

3.1 Performance of the BLAS and basic LAPACK decompositions

Table 2 reports on the speed in Megaflops of the BLAS, SGEMV (matrix-vector multiplication) and SGEMM (matrix-matrix multiplication). It also reports the speeds of LU decomposition (SGETRF), QR decomposition (SGEQRF) and bidiagonal reduction (SGEBRD). It does this both for Fortran BLAS and optimized BLAS. All matrices are dimensioned (LDA,N), where LDA = 513. The block size NB in the blocked algorithms for SGETRF, SGEQRF and SGEBRD was 32. It is interesting to see that the performance of SGEMV is a strongly *nonmonotonic* function of matrix dimension. We believe this is because for N $<$ 256, the matrix fits in second level cache without conflicts, whereas for N $\geq$ 256, cache conflicts and cache misses occur. The Level 3 BLAS routines like SGEMM can more easily compensate for this than Level 2 BLAS like SGEMV.

Table 1: Names and descriptions of routines tested

Name	Description	Status
SGEMV	Matrix-vector multiply	Level 2 BLAS
SGEMM	Matrix-matrix multiply	Level 3 BLAS
SGETRF	LU decomposition	in LAPACK
SGEQRF	QR decomposition	in LAPACK
SGEBRD	Reduction to bidiagonal form	in LAPACK
SBDSQR	Compute complete SVD of a bidiagonal matrix using QR iteration	in LAPACK
SBDSDC	Compute complete SVD of a bidiagonal matrix using divide-and-conquer	new routine
SGESVD-QR	Compute complete SVD of a dense matrix using QR iteration	in LAPACK as SGESVD
SGESVD-DC	Compute complete SVD of a dense matrix using divide-and-conquer	new routine
SGELS	Solve the least squares problem using QR decomposition	in LAPACK
SGELSX	Solve the least squares problem using QR decomposition with pivoting	in LAPACK
SGELSS-QR	Solve the least squares problem using the SVD based on QR-iteration	in LAPACK as SGELSS
SGELSS-QRf	Solve the least squares problem using the SVD based on QR-iteration but where the left singular vectors are left factored	new routine
SGELSD	Solve the least squares problem using the SVD based on divide-and-conquer	new routine

Table 2: Speed of BLAS and LAPACK Routines (NB = 32, LDA = 513)

Speed in megaflops using optimized BLAS							
		Dimension					
Routine	Description	50	100	200	300	400	500
SGEMV	matrix-vector multiply	64.7	60.6	64.2	44.7	39.3	35.7
SGEMM	matrix-matrix multiply	128.1	146.4	134.4	136.3	136.8	140.3
SGETRF	LU decomposition	42.0	56.5	78.9	88.9	93.8	94.0
SGEQRF	QR decomposition	44.0	51.4	77.5	90.9	97.6	102.9
SGEBRD	Bidiagonal reduction	38.5	49.1	52.1	50.5	50.6	51.3
Speed in megaflops using Fortran BLAS							
		Dimension					
Routine	Description	50	100	200	300	400	500
SGEMV	matrix-vector multiply	51.7	50.2	51.0	41.5	36.8	32.6
SGEMM	matrix-matrix multiply	42.7	48.8	51.4	41.4	36.3	32.5
SGETRF	LU decomposition	28.0	30.8	39.7	42.9	44.2	45.3
SGEQRF	QR decomposition	35.2	38.5	36.2	35.8	35.5	35.1
SGEBRD	Bidiagonal reduction	31.5	41.1	33.0	31.9	32.3	32.6

3.2 Performance of the Bidiagonal SVD

We report on the speed of the bidiagonal SVD (computing all singular values and left and right singular vectors). We used four types of test matrices, all generated by LAPACK test matrix generator `SLATMS`:

Type 1. These bidiagonal matrices were randomly generated with singular values distributed arithmetically from ϵ up to 1.

Type 2. These bidiagonal matrices were randomly generated with singular values distributed geometrically from ϵ up to 1.

Type 3. These bidiagonal matrices have 1 singular value at 1 and the other $n-1$ clustered at ϵ.

Type 4. These bidiagonal matrices were generated by taking a dense matrix with independent random entries uniformly distributed in $(-1,1)$, and reducing it to bidiagonal form.

Table 3 shows the speedup of SBDSDC, the bidiagonal SVD based on divide-and-conquer, with respect to SBDSQR, the bidiagonal SVD based on QR-iteration (all singular values and left and right singular vectors are computed). As can be seen, the speedup is a growing function of matrix dimension. Indeed, the running

Table 3: Speedup of SBDSDC over SBDSQR

Speedup using optimized BLAS					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	1.48	2.71	5.53	7.94	9.47
type 2	1.33	2.35	5.79	8.67	11.33
type 3	18.00	55.10	68.97	97.59	121.43
type 4	1.59	2.93	6.00	8.57	10.56
Speedup using Fortran BLAS					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	1.58	2.27	3.67	4.47	5.00
type 2	1.33	2.41	4.78	6.67	7.67
type 3	42.86	55.10	62.07	82.81	116.67
type 4	1.62	2.27	4.00	4.94	5.76

time for SBDSDC appears to grow like $n^{2.3}$ rather than n^3, as for SBDSQR. Also, the speedup is better when using the optimized BLAS rather than Fortran BLAS, because SBDSDC spends much of its time in SGEMM, whereas SBDSQR cannot even use Level 2 BLAS.

3.3 Performance of the Dense SVD

We report on the speed of the dense SVD (computing all singular values and left and right singular vectors). We used the same four test matrix types as before, but now all are dense.

Table 4 shows how much the dense SVD speeds up when using divide-and-conquer instead of QR, both for optimized BLAS and Fortran BLAS. When $n = 400$, the speedups range from 2.19 to 3.86 for the optimized BLAS, and much less for Fortran BLAS. This is because using optimized BLAS helps SBDSDC much more than SBDSQR, as seen in Table 3.

Table 5 shows what fraction of time the dense SVD spends doing the bidiagonal SVD. The results are shown only for optimized BLAS; the Fortran BLAS fractions are comparable but slightly lower. The most significant result is that the bidiagonal fraction goes from being 60% to 80% of the total time for SGESVD-QR to at most 25% for SGESVD-DC, for large matrices. This means that the bidiagonal SVD has gone from being the bottleneck in the dense SVD to a small fraction of time.

Thus, any significant further improvements in the speed of the dense SVD must come from speeding up the non-bidiagonal part of the computation. One way to

Table 4: Speedup of SGESVD-DC over SGESVD-QR

Speedup using optimized BLAS					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	1.07	1.65	2.55	2.85	3.07
type 2	0.97	1.29	1.79	2.03	2.19
type 3	2.22	3.08	3.42	3.85	3.86
type 4	1.16	1.70	2.55	2.91	3.29
Speedup using Fortran BLAS					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	0.92	1.16	1.62	1.74	1.69
type 2	0.76	0.92	1.35	1.35	1.38
type 3	1.69	1.86	1.93	1.80	1.92
type 4	0.97	1.19	1.62	1.76	1.75

do this is to abandon computing the singular vectors explicitly, leaving them in the factored form provided by the algorithm. We exploit this possibility in the next section.

3.4 Performance of Solvers for the Linear Least Squares Problem

We consider solving N-by-N least squares problems with single right hand sides. We use the same four test matrices as before. The algorithms we consider are

- SGELS - QR decomposition (currently in LAPACK)
- SGELSX - QR decomposition with pivoting (currently in LAPACK)
- SGELSS-QR - SVD (currently in LAPACK)
- SGELSS-QRf - SVD but maintaining the left singular vectors of the bidiagonal matrix as a list of $O(N^2)$ Givens rotations
- SGELSD - SVD based on divide-and-conquer, factored form

We present square problems only, since M-by-N problems with $M \gg N$ are generally reduced to an N-by-N problem by an initial QR decomposition, and this dominates all later computations.

Table 5: Fraction of time Dense SVD spends in Bidiagonal SVD (optimized BLAS)

Fraction of SGESVD-DC spent in SBDSDC					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	0.50	0.42	0.35	0.29	0.25
type 2	0.43	0.30	0.20	0.16	0.13
type 3	0.09	0.04	0.04	0.03	0.02
type 4	0.51	0.41	0.32	0.27	0.25
Fraction of SGESVD-QR spent in SBDSQR					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	0.69	0.70	0.75	0.79	0.78
type 2	0.59	0.55	0.65	0.66	0.67
type 3	0.71	0.73	0.74	0.81	0.77
type 4	0.70	0.71	0.75	0.79	0.79

In addition to measuring the speedup of SGELSD and SGELSS-QRf over SGELSS-QR, we measure times relative to SGELS, the fastest, and least reliable, of all the methods. This quantifies the tradeoff between speed and reliability inherent in this problem. Results shown in tables are for optimized BLAS only.

Table 6 shows that both new least squares solvers, SGELSS-QRf and SGELSD, are significantly faster than the older SGELSS. Table 7 shows that a fully reliable SVD-based solution to the linear least square problem now costs no more than 4.35 times as much as the fastest solver (SGELS), whereas it used to cost as much as 15 times more. Furthermore, it is only about twice as expensive as the more reliable QR with pivoting scheme used in SGELSX. "Completely reliable" rank-revealing QR schemes have been designed [16], and these are likely to be intermediate in speed between SGELSX and the SVD based schemes. The table entry describing the speed of SGEBRD (bidiagonal reduction), which is performed by all SVD based schemes we consider, shows that at best we can expect to run 3.37 times faster than SGELS, and we are close to this lower bound.

When using Fortran BLAS, the time for SGELSD decreases to about 2.5 times as much as SGELS, and SGEBRD takes about twice as long as SGELS, as predicted by the operation counts.

3.5 Accuracy Assessment

We use two measures of accuracy of the computed SVD $A = X\Sigma Y^T$: the residual $\max_i \|Ay_i - \sigma_i x_i\|/(\epsilon\sigma_1)$ and the orthogonality of the singular vectors $\max(\|YY^T -$

Table 6: Speedups of New SVD-based Least Squares Solvers

Speedup of SGELSD over SGELSS-QR					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	0.87	1.33	2.36	2.95	3.18
type 2	0.86	1.27	2.07	2.18	2.32
type 3	2.77	3.38	3.78	4.00	3.82
type 4	0.89	1.43	2.54	3.21	3.18

Speedup of SGELSS-QRf over SGELSS-QR					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	1.50	2.00	2.93	3.47	3.59
type 2	1.33	1.69	2.35	2.47	2.38
type 3	1.57	2.00	2.98	3.29	3.33
type 4	1.48	2.00	3.10	3.59	3.50

$I\|/\epsilon, \|XX^T - I\|/\epsilon)$, where ϵ is machine precision. Ideally these two measure should be $O(1)$ for any dimension, but we would not be unhappy to get numbers growing with N, perhaps as $O(N)$, although we cannot prove so tight a bound. In fact, the QR based SVD routines exhibit measures as large as $2N$ for $N = 400$, though they are usually much smaller, whereas the ratios for divide-and-conquer routines were never larger than 13. In other words, the divide-and-conquer based SVD is not only faster but more accurate than the QR based approach.

The above results are for dense matrices. It turns out one can prove tighter relative error bounds for singular values and singular vectors for the QR-based bidiagonal SVD [8, 7]. We currently cannot guarantee this high relative accuracy with divide-and-conquer, just the absolute accuracy described in the last paragraph.

4 Conclusions

We have described a new implementation of the singular value decomposition which is both faster and more accurate than its predecessor. It achieves this by using a divide-and-conquer bidiagonal SVD algorithm instead of QR iteration. The speedup on bidiagonal matrices grows with dimension, so that for 400-by-400 matrices, the bidiagonal SVD is taking just 25% of the total time for the dense SVD, whereas the older bidiagonal SVD took up to 80% of the total time. This means the bidiagonal SVD is no longer the bottleneck in the computation.

We have also shown how to achieve large speedups for solving the linear least

Table 7: Timings of Least Squares Solvers relative to SGELS

Time(SGELSX) / Time(SGELS)					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	1.49	1.29	1.60	1.88	2.39
type 2	1.49	1.45	1.87	2.19	2.72
type 3	1.32	1.09	1.53	1.84	2.39
type 4	1.15	1.16	1.53	1.84	2.39

Time(SGEBRD) / Time(SGELS)					
	Dimension				
Test Matrix	50	100	200	300	400
all types	1.32	1.68	2.73	2.79	3.37

Time(SGELSS-QRf) / Time(SGELS)					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	3.73	3.23	3.87	3.95	4.24
type 2	3.05	2.68	3.40	3.49	4.02
type 3	3.90	3.44	3.80	3.95	4.24
type 4	3.38	3.23	3.87	3.95	4.35

Time(SGELSD) / Time(SGELS)					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	6.44	4.84	4.80	4.65	4.78
type 2	4.75	3.55	3.87	3.95	4.13
type 3	2.20	2.03	3.00	3.26	3.70
type 4	5.59	4.52	4.73	4.42	4.78

Time(SGELSS-QR) / Time(SGELS)					
	Dimension				
Test Matrix	50	100	200	300	400
type 1	5.59	6.45	11.33	13.72	15.22
type 2	4.07	4.52	8.00	8.60	9.57
type 3	6.10	6.88	11.33	13.02	14.13
type 4	5.00	6.45	12.00	14.19	15.22

squares problem using the SVD. While the old SVD based least squares solver required $12n^3$ flops and took up to 15 times longer than the fastest backward stable solution, the new SVD based routines require just $\frac{8}{3}n^3$ flops and take 4.4 times as long. These results make the SVD a more practical tool in a variety of computations.

Our improved routines will be part of a future LAPACK release.

Acknowledgements

We wish to thank Gene Golub for pointing out the Rutishauser reference on Jacobi, and William Kahan for help during the writing of this paper.

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, Release 1.0.* SIAM, Philadelphia, 1992. 235 pages.

[2] P. Arbenz and G. Golub. On the spectral decomposition of Hermitian matrices modified by row rank perturbations with applications. *SIAM J. Matrix Anal. Appl.*, 9(1):40–58, January 1988.

[3] C. Bischof. Adaptive blocking in the QR factorization. *J. Supercomputing*, 3(3):193–208, 1989.

[4] C. Bischof, X. Sun, and M. Marques. Parallel bandreduction and tridiagonalization. In R. Sincovec et al, editor, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume 1, pages 383–390. SIAM, 1993.

[5] J. J. M. Cuppen. The singular value decomposition in product form. *SIAM J. Sci. Stat. Comput.*, 4:216–221, 1983.

[6] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[7] P. Deift, J. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular values decomposition and Hamiltonian mechanics. *SIAM J. Num. Anal.*, 28(5):1463–1516, October 1991. (LAPACK Working Note #11).

[8] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990.

[9] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[11] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Num. Anal. (Series B)*, 2(2):205–224, 1965.

[12] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Num. Math.*, 14:403–420, 1970.

[13] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[14] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15:318–334, 1973.

[15] M. Gu and S. Eisenstat. A stable algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, September 1992.

[16] M. Gu and S. Eisenstat. An efficient algorithm for computing a rank-revealing QR decomposition. Computer Science Dept. Report YALEU/DCS/RR-967, Yale University, June 1993.

[17] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. Research Report YALEU/DCS/RR-933, Department of Computer Science, Yale University, December 1992. To appear in SIAM J. Matrix Anal. Appl., 1995.

[18] E. Jessup and D. Sorensen. A parallel algorithm for computing the singular value decomposition of a matrix. Mathematics and Computer Science Division Report ANL/MCS-TM-102, Argonne National Laboratory, Argonne, IL, December 1987.

[19] E. Jessup and D. Sorensen. A divide and conquer algorithm for computing the singular value decomposition of a matrix. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66, Philadelphia, PA, 1989. SIAM.

[20] W. Kahan, 1993. private communication.

[21] R.-C. Li, 1994. private communication.

[22] B. Parlett and V. Fernando. Accurate singular values and differential qd algorithms. Math Department PAM-554, University of California, Berkeley, CA, July 1992. to appear in Num. Math.

[23] H. Rutishauser. On Jacobi rotation patterns. In N. Metropolis, A. Taub, J. Todd, and C. Tompkins, editors, *Proceedings of Symposia in Applied Mathematics, Vol. XV: Experimental Arithmetic, High Speed Computing and Mathematics*. American Mathematical Society, 1963.

[24] H. Rutishauser. *Lectures on Numerical Mathematics*. Biskhäuser, 1990.

[25] J. Rutter. A serical implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Mathematics Dept. Master's Thesis available by anonymous ftp to tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-94-799, file all.ps, University of California, 1994.