

$UB_{k+1}V$ Block Sparse Householder Decomposition *

G.W. Howell
North Carolina State University
Raleigh, North Carolina

July 8, 2009

Abstract

This paper describes Householder reduction of a rectangular sparse matrix to small band upper triangular form. Using block Householder transformations gives good orthogonality, is computationally efficient, and has good potential for parallelization. The algorithm is similar to the standard dense Householder reduction used as part of the usual dense SVD computation. For the sparse algorithm, the original sparse matrix is accessed only for sparse matrix dense matrix (SMDM) multiplications. For a triangular bandwidth of $k + 1$, the dense matrices are the k rows or columns of a block Householder transformation.

Using an initial random block Householder transformation allows reliable computation of a collection of largest singular values. Some other potential applications are in finding low rank matrix approximations and in solving least squares problems.

1 Introduction

In 1964, Golub and Kahan proposed two bidiagonalization algorithms. For dense matrices, they proposed Householder bidiagonalization, for sparse matrices Lanczos bidiagonalization.

This paper evolved through several realizations. A first realization was that the usual Householder bidiagonalization can be applied to a sparse matrix, filling in alternating rows and columns only as they are eliminated. [?].

*Supported by NIH Molecular Libraries Roadmap for Medical Research, Grant 1 P20 Hg003900-01.

A second realization was that, as in the dense case, a reduction to banded upper triangular form allows the reduction to banded form to be BLAS-3 as opposed to a mixture of BLAS-2 and BLAS-3. BLAS-3 operations are much faster than BLAS-2 and are more amenable to efficient parallelization. Similarly, multiplications AX , $A'Y$, A sparse, perform a higher rate of computation when the number of columns in X and Y is increased.

Finally, a problem with eliminating alternate rows and columns is that if these are already zero, then eliminating them converts them to dense storage, but delivers no new information. Applying initial random Householder block transformations provides dense column and block eliminations and also ensures that multiplications of the original sparse matrix are by dense blocks of vectors.

Section 2 gives some examples to show why Lanczos bidiagonalization is unstable, justifying why Householder bidiagonalization may be appropriate.

Section 3 gives some numeric results justifying the assertion that AX , $A'Y$ (Sparse matrix dense matrix or SPMD) operations are likely to be faster than Ax , $A'y$.

Section 4 gives a bit of theory.

Section 5 shows numeric experiments with the Davis collection of sparse matrices. Using a couple of GBytes of RAM, most matrices to size 100K allow computation of at least some singular values.

Section 6 proposes related work.

Householder transformations are scalably stable. Dense QR decompositions using blocked Householder transformations are predominantly BLAS-3, hence efficient on a variety of computer architectures, e.g., cache-based multi-core architectures, GPU processors, and distributed memory parallel architectures.

It's natural to extend use of blocked Householder decomposition to the sparse case. Applications include solving $Ax = b$, finding x to minimize $\|Ax - b\|_2$, and finding some singular values of a sparse matrix A .

- Householder reductions can be applied to sparse matrices by forgoing updates of blocks of the original matrix till the step on which a column or row block is to be eliminated. For reduction to Hessenberg form or upper triangular form, the idea of deferring updates has been repeatedly used. Kaufman implemented deferral of updates in sparse Householder QR factorization [19]. For other applications of this idea, see for example ARPACK [24], Sosonkina, Allison, and Watson [32], and Dubrulle [8]. For Householder reduction to bidiagonal form, the idea of deferring updates is implicit in the LAPACK reduction `_GEBRD` (for the dense case) [4], with the extension to the sparse case explicitly outlined in Howell, Demmel, Fulton, Hammarling, and Marmol [15]. The work here is es-

essentially an extension of the dense Grösser and Lang algorithm (Grösser and Lang [12] and Lang [20]) to apply in the sparse case.

- Block Householder transformations are BLAS-3 and hence computationally efficient.

On current computer architectures, whether cache-based single or multicore, GPU or other hardware accelerators connected to general processors, or for distributed and massively parallel computing, dense BLAS-3 matrix matrix multiplies are significantly faster than BLAS-2 matrix vector multiplications. In all these cases, BLAS-3 operations run at near peak predicted performance. So for example, the top 500 computers are ranked by their performance in dense LU decomposition, accomplished almost entirely by BLAS-3 operations.

Arguably, sparse matrices are more common than dense ones. Sparse BLAS-2 operations consisting of sparse matrix dense vector operations are ubiquitous, for example, in iterative solution of linear equations. The BLAS-3 operation consisting of multiplying a sparse matrix by a dense matrix (SMDM) is faster. We make the case explicitly here for a cache based architecture, but SMDM operations are also likely to be faster for GPUs and massively parallel distributed computations ([6],[7]). SMDM operations and other BLAS-3 operations comprise almost all the operations in the sparse Householder reduction to banded form.

- This paper concentrates on reduction to banded triangular form. We indicate some applications including finding some singular values and finding low rank approximations. We also indicate how to extend the LSQR algorithm ([25]) for solving least squares problems to use the banded reduction as a BLAS-3 sparse solver.

Sparse Householder bidiagonalization was proposed in [15], which used the BLAS 2.5 idea of combining the multiplications $y \leftarrow A^T x$ and $w \leftarrow Ay$ into one operation. When A is too large to fit in cache memory, the combined operation reduces the required data movement. The combined computation can often be accomplished in about the same time as the slower of $A^T x$ or Ay . The combined operation can be used either in Householder bidiagonalization or in replacing the Lanczos procedures used in LSQR [25] or SVDPACK, or in a Lanczos scheme with partial reorthogonalization such as PROPACK [23],[22]. Here we replace the multiplications Ax and $A^T y$ by AX and $A^T Y$ (multiple columns in X and Y). Typically the BLAS-3 operations AX and $A^T Y$, X, Y with k columns, execute at a higher rate than the BLAS 2.5 operation, motivating reduction to bandwidth $k+1$ as compared to $k+1 = 2$ for bidiagonalization.

1.1 Comparison to Other Sparse Methods

Both the sparse and dense $UB_{K+1}V$ Householder based decompositions are BLAS-3 algorithms with U and V scalably orthogonal.

For comparison consider the classic approach to finding a few singular values of a sparse matrix, so-called Lanczos bidiagonalization¹, which Golub and Kahan proposed as a sparse alternative to Householder bidiagonalization [10]. Lanczos bidiagonalization can proceed without storing multipliers, relying on a three term recursion, so that only the last few left and right multipliers are needed. Storage requirements are minimal. In exact arithmetic the Lanczos bases would be orthogonal. In rounding arithmetic, there is a rapid loss of orthogonality, and even of linear independence, as illustrated in Figure (1.1). A matrix of size a few hundred was randomly generated in octave, the left and right multipliers were saved, and the numeric rank of the multipliers was calculated from a computed singular value decomposition.

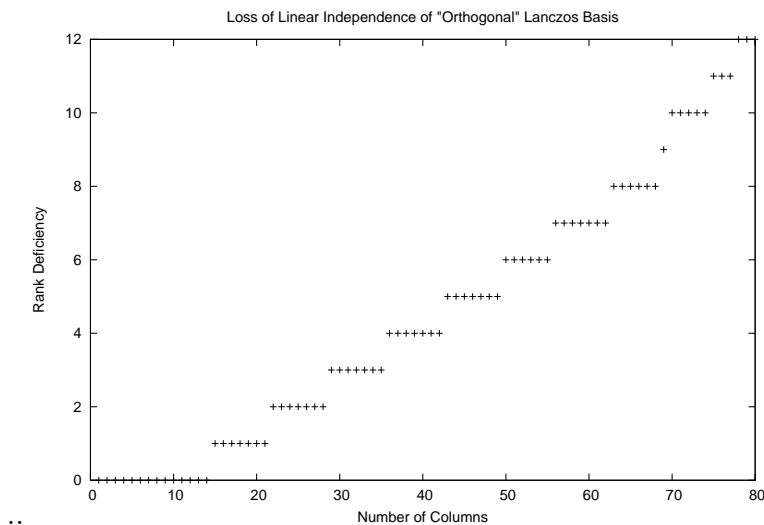


Figure 1: Lanczos bases suffer loss of numeric rank.

As memory per available node grows, using the memory for purposes of better stability becomes feasible. In order to preserve orthogonality, the multiplier vectors are frequently stored, and in a Lanczos algorithm, reorthogonalized. Table 1 compares condition numbers of various methods for constructing bases for the columns of Hilbert matrices, illustrating the orthogonality of Householder transformations.

¹Most usually Golub-Kahan bidiagonalization refers to dense Householder bidiagonalization

Sizes	10	20	30	40	50
Householder QR	1.0	1.0	1.0	1.0	1.0
L in LU	9.70	18.7	46.5	53.0	72.2
MGS QR	1.001	98.5	170.	122.5	3631

Table 1: L_2 Condition Numbers of Bases. Here we factored Hilbert matrices of various sizes and compared condition numbers of Q for QR factorization or L for LU decomposition. By working with blocks X we can use BLAS-3 block Householder as opposed to x and BLAS-2 Householder (e.g. ARPACK, or the GMRES algorithm).

Basis	Lanczos UB_2V	PROPACK UB_2V	ARPACK GMRES UHU^T	$UB_{K+1}V$
Vecs	$O(1)$	$2N$	$2N$	$4N$
Basis	Loss of Rank	Loss of Orthog	Keeps Orthog	Keeps Orthog
BLAS	BLAS-1	BLAS-1	BLAS-2	BLAS-3
flops	$4Nn_z$ $+O(N)$	$4Nn_z$ $+4N^2n$	$4Nn_z$ $+4N^2n$	$4Nn_z$ $+4(n+m)N^2$ Scalable

Table 2: Summary Chart Comparing Sparse Decompositions.

If modified Gram-Schmidt as opposed to Householder orthogonalization is used, the number of flops can be halved. Since modified Gram-Schmidt is BLAS-1, use of BLAS-3 block Householder transformations is likely to be faster, as well as more nearly orthogonal. Alternatively, Jalby and Philippe [17], Vanderstraten [35], Stewart [33], Giraud and Langou [9] and others have designed block Gram-Schmidt algorithms to be of comparable stability to modified Gram-Schmidt, which gives a well-conditioned but not orthogonal basis for an ill-conditioned set of vectors such as those obtained a Lanczos method. If used in combination with block Lanczos methods, such as that proposed by Golub, Lusk, and Overton [11], it is possible that a predominantly BLAS-3 algorithm with good stability can be obtained.

More usually, Lanczos bidiagonalization with reorthogonalization is used in SVDPACK [3] and PROPACK [22], with theoretical development in Larsen's thesis [23] and work by Simon and Zha [31]. Each of these methods is appropriate for finding a few singular values. Sparse MATLAB instead uses ARPACK [24].

Table 2 compares some sparse decompositions. The storage requirements of $UB_{K+1}V$ are seen to be high, but if the speedup $\gg 2$ of Figure 2 is rep-

representative, $UB_{K+1}V$ should be competitive in speed with methods entailing orthogonalization.

1.2 Shared Memory Parallelization

For shared memory (or multi-core processors), we get good speedups with a few OpenMP calls in the AX and $Y^T A$ routines – and with a couple of OpenMP routines to use enable multi-core “long skinny” BLAS-3 operations.

2 Sparse Matrix Full Matrix Products

For many current computer architectures, BLAS-3 computations (matrix matrix multiplies) are much faster than BLAS-2 (e.g. matrix vector multiplications) and BLAS-1 (vector operations). When a relatively small set of data can be held in fast access memory, BLAS-3 matrix matrix multiplications allow a high ratio of floating point operations to data transfer from the slower memory. Dense LU and QR matrix decompositions can be blocked so that almost all operations are BLAS-3. Thus LU and QR decompositions of dense matrices can be performed at near the peak rate of operation on most current computer processors. Table 3 compares some dense and sparse BLAS-3 and BLAS-2 operations on a typical modern processor.

On this machine, multiplication of A and B each of size $2K$ square runs at about 70 per cent of the peak speed of 6472 Mflops (2 flops on each clock cycle on each of 2 cores). Other operations are significantly slower. When the number of columns in a “tall skinny” matrix A is less than about 100, computation rate is roughly proportional to the number of columns.

Classic iterative schemes for solving systems of sparse linear equations rely on multiplications Ax , $y^T A$, and BLAS-1 (vector vector) operations. In Table 3, sparse Ax , $y^T A$ operations are spectacularly slow, as is AB for A dense with 4 columns. These operations run at less than one per cent as fast as the $2K \times 2K$ BLAS-3, motivating the algorithm presented in this paper.

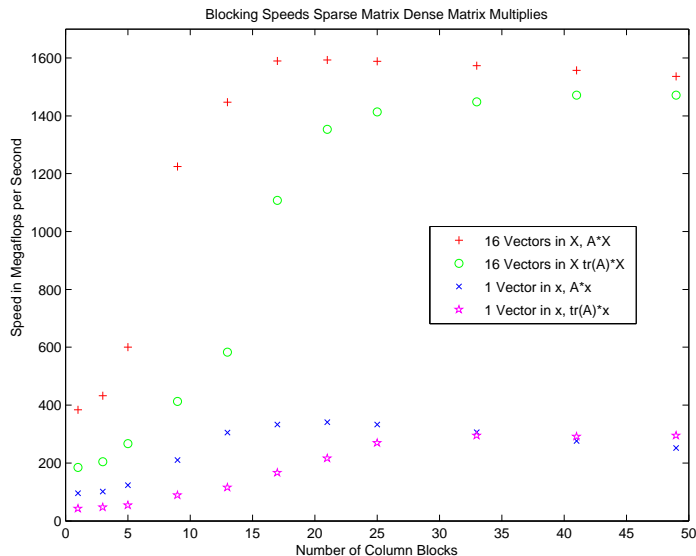
Accessing A to perform multiplications AX is seen to give significantly better performance, motivating the algorithm developed here. For the partial $UB_{K+1}V^T$ decomposition, access to A is only for the multiplications AX , $Y^T A$. Essentially all the rest of the operations are BLAS-3 with the block size the same as the number of columns in X .

Figure 2 indicates the relative effects of block size vs. matrix storage in speeding sparse matrix multiplications. Column blocking is effective in improving performance when nonzero entries are uniformly distributed. For other sparse matrices, different matrix storages can improve performance of the AX kernel. See for example, Toledo [34], Angeli, et. al [1] and Im’s Phd dissertation [16] offer some guidance in arranging storage of A to speed

Operation		Mflops/second	Fraction of Peak
Dense BLAS-3	AB		
Fedora-8 default	$2K \times 2K \times 2K$	1054	.141
ATLAS	$2K \times 2K \times 2K$	5246	.70
ATLAS	$100K \times 96 \times 96$	2021	.27
ATLAS	$100K \times 64 \times 64$	1445	.19
ATLAS	$100K \times 32 \times 32$	661	.089
ATLAS	$100K \times 16 \times 16$	299	.040
ATLAS	$100K \times 4 \times 4$	36.3	.0049
Dense BLAS-2	$A \ 4K \times 4K$		
ATLAS DGEMV	Ax	1011	.136
A random sparse	x dense		
$100K \times 100K$			
A nonblocked	Ax	57	.0077
A nonblocked	$x^T A$ (1 core)	30	.0040
A blocked	Ax	115	.0154
A blocked	$x^T A$	188	.025
A random sparse	X dense		
$100K \times 100K$	with 16 columns		
A nonblocked	AX	390	.052
A nonblocked	$A^T X$ (1 core)	195	.026
A blocked	AX	1570	.21
A blocked	$A^T X$	1450	.19
A random sparse	X dense		
$100K \times 100K$	with 32 columns		
A nonblocked	AX	568	.076
A nonblocked	$A^T X$ (1 core)	241	.032
A blocked	AX	1770	.24
A blocked	$A^T X$	1721	.23

Table 3: The table compares speeds of sparse and dense Ax , AX . SMDM (sparse matrix dense matrix) multiplication is as much thirty times faster than sparse matrix dense vector multiplication. The SMDM flop rates are comparable to BLAS-3 flop rates for “tall skinny” matrices (which correspond to the BLAS-3 computations in a $UB_{K+1}V$ partial decomposition). These timings are from an Intel Core2 Duo with a clock speed of 1862 MHz, a peak flop rate of 7448 Mflops/sec, 2 MBytes L2 cache, Fedora-8 32 bit, gfortran version 4.1.2. Unless otherwise indicated, calculations used both cores. The sparse matrices have 1 of 200 entries nonzero, randomly distributed.

(Sparse A)*(Dense X)–20X Seedup



..

Figure 2: Speeding Multiplication of a Randomly Generated Sparse Matrix. Blocking the Matrix and Multiplying by Multiple Vectors Reduce Cache Misses. The matrix here is 100K by 100K with 500 randomly distributed nonzeros entries per row. The computation was with a 64 bit 2.4 GHz two Pentium processor with 512 MByte L2 cache compiled with an Intel Fortran compiler. Parallelization is provided with one openMP parallel loop for AX , Ax and one also for $X^T A, x^T A$.

the computation Ax . The OSKI package (Vuduc, Demmel, and Yelick [36]) automates the process of choosing storage of A . Nishtala, Vuduc, Demmel, and Yelick [28] offer some guidance as to when OSKI is likely to be effective.

3 The sparse $UB_{k+1}V$ partial decomposition

We adapt a dense BLAS-3 algorithm for reduction to bandwidth $k + 1$ using Householder reductions of block size k . Dense implementations were by Grösser and Lang, [12], [20]. Using deferred updates to convert a dense to a sparse algorithm for $k = 1$ (the bidiagonal case) is discussed in Howell, Demmel, Fulton, and Marmol [15].

3.1 Pseudo-code for $UB_{K+1}V$

As given here, the code runs to “completion”, returning an upper triangular banded matrix, with bandwidth $k + 1$ (except see below). In exact arithmetic, the returned matrix would have the same singular values as the original matrix and is related to the original matrix by

$$A_{return} = (I - U_l)L_l(I - U_l^T)(A_{orig} - UZ - WV) \quad (3.1)$$

or by

$$A_{return} = \prod_{i=1}^l (I - U_i L_i U_i^T) A_{orig} \prod_{i=1}^{l-1} (I - V_i^T T_i V_i) \quad (3.2)$$

where $(I - U_i L_i U_i^T)$ and $(I - V_i^T T_i V_i)$ are block Householder transformations.

On return the blocks U_i are stored in the i th block of k columns in U (lower triangular) and the blocks V_i are stored in the i th block of k rows in V (upper triangular). Similarly, W is lower triangular and Z upper triangular.

Required inputs are

- m number of rows
- n number of columns
- K block size

The algorithm proceeds by alternately eliminating blocks of K columns and K rows. When no more blocks of size K can be made, then the rest of the columns are eliminated as one block. Hence the last block of L can be up to twice as large as the others.

For $A = UB_4V$, $m = 10$, $n = 8$, $K = 3$, the returned B_4 has the following form

$$B_4 = \begin{pmatrix} x & x & x & x & 0 & 0 & 0 & 0 \\ 0 & x & x & x & x & 0 & 0 & 0 \\ 0 & 0 & x & x & x & x & 0 & 0 \\ 0 & 0 & 0 & x & x & x & x & x \\ 0 & 0 & 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Capital letters are used below to indicate that variables are matrices (as opposed to vectors).

Pseudo-Code for $UB_{K+1}V$

```

function [B, U, W, V, Z, L, Ltemp, T] = band(m, n, K, Aold)
!
!   Assume  $m > n$ 
!   Input Variables
!       m – number of rows
!       n – number of columns
!       k – number of superdiagonals in returned matrix
!           (also the block size for multiplications by Aold)
!       Aold – input matrix in sparse storage
!
!   Output Variables
!   B is an m by n matrix with upper bandwidth k+1
!       U (m x n), W (m x n), V (n x n), Z (n x n)
!       L (m x K), T (K x n), Ltemp (2K x 2K)
!   where (compare to 3.1,3.2, the extra
!   term here is from eliminating all remaining columns
!   as a final block, for a large sparse problem, this final
!   block will not be eliminated).
!       
$$B = (I - U_{last}L_{temp}U_{last}^T) \prod_i (I - U_i L_i U_i^T) A_{old} \prod_i (I - V_i^T T_i V_i)$$

!       
$$= (I - U_{last}L_{temp}U_{last}^T) (A_{old} - UZ^T - WV^T)$$

!   where  $W = [W_1|W_2|\dots|W_l]$ ,  $U = [U_1|U_2|\dots|U_l]$ ,
!        $V = [V_1|V_2|\dots|V_l]$ ,  $Z = [Z_1|Z_2|\dots|Z_l]$ ,
!       where each of the blocks  $U_i, V_i, W_i, Z_i$  has  $K$  columns
!       and  $U_{last}$  may have up to  $2K$  columns.
!
!   Initializations
!
B ← 0m,n ; W ← 0m,n ; U ← 0m,n ;
  V ← 0n,n ; Z ← 0n,n ; L ← 0m,k ; T ← 0n,K ;
blks = floor((n-K)/K) ;
for i=1:blks,
  il(i) ← K (i-1) + 1 ; ih(i) ← ik ;
end
if ( K blks != n )
  il(blks+1) ← k blks + 1 ; ih(blks+1) ← n ;
end
mnow ← m ; nnow ← K ;
Atemp ← Aold( : ,1:K) ;           ! Extract first column block of Aold
ilow ← 1 ; ihi ← K ; ihp1 ← K+1 ;
C ← Aold( : ,1:K) ;           ! Extract first row block of Aold
!

```

```

! - qrl returns the QR factorization of the
! - of the first column block of  $A_{old}$  where
!        $R = (I - U_{temp} L_{temp} U_{temp}^T) C$ 
!
! [  $U_{temp}, R, L_{temp}$  ] = qrl(m, K, C) ;
!  $U(i_{low}:m, i_{low}:i_{hi}) \leftarrow U_{temp}$  ;  $L(i_{low}:i_{hi}, :) \leftarrow L_{temp}$  ;
!  $B(i_{low}:m, i_{low}:i_{hi}) \leftarrow R$  ;
!
! - C will be the update of the first row block of  $A_{old}$ 
!
!  $L_{ua} \leftarrow (L_{temp} U_{temp}^T) A_{old}(i_{low}:m, i_{hp1}:n)$ ;
!                                     ! SPMD multiplication with  $A_{old}$ 
!
!  $C \leftarrow B(i_{low}:i_{hi}, i_{hp1}:n)$  ;
!  $C \leftarrow C - U(1:K, 1:K) L_{ua}$  ;
!
! - qlt performs the QL factorization of C so that
!        $T_{temp} = C (I - V_{temp} L_r V_{temp}^T)$ 
!
! [  $V_{temp}, L_r, T_{temp}$  ] = qlt( K, n-K, C) ;
!  $B(1:K, K+1:n) \leftarrow L_r$ ;  $T(1:K, :) \leftarrow T_{temp}$  ;
!
! - Get the first blocks for U,V,Z,W
!
!  $T_{emp} \leftarrow T_{temp} V_{temp}$  ;
!  $Z(1:K, K+1:n) \leftarrow L_{ua} - (L_{ua} V_{temp}^T) T_{emp}$  ;
!  $T_{emp2} \leftarrow V_{temp}^T T_{temp}$  ;
!  $W(:, 1:K) \leftarrow A_{old}(:, K+1:n) T_{emp2}$  ;
!  $V(1:K, K+1:n) \leftarrow V_{temp}$  ;
!
! - Now loop through all but the end block
!
! for i=2 : blks ,
!        $i_{low} \leftarrow i_l(i)$  ;  $i_{hi} \leftarrow i_h(i)$  ;  $i_{hp1} \leftarrow i_h(i)+1$  ;
!
! To proceed with a reduction to banded form,
! we need to multiply the updated A
!  $A_{updated} = A_{old} - U Z - W V$ 
! by a block of vectors X. Since  $A_{updated}$  is presumed dense,
!  $A_{updated} X$  is accomplished as
!  $A_{old} X - U (Z X) - W (V X)$ 
!
! Update the current column block of A

```

```

!
C ← Aold(ilow:m, ilow:ihi) ;      ! Extract a column block of Aold
C ← C - U(ilow:m, 1:ilow-1) Z(1:ilow-1, ilow:ihi) ;
C ← C - W(ilow:m, 1:ilow-1) V(1:ilow-1, ilow:ihi) ;
!
! qrl performs the QR factorization of the current column block.
!
[ Utemp, R, Ltemp ] ← qrl(m-ilow+1, K, C) ;
U(ilow:m, ilow:ihi) ← Utemp ; L(ilow:ihi, :) ← Ltemp ;
!
! Multiply (Li UiT) Aupdate
!
B(ilow:m, ilow:ihi) ← R ;
Lup ← Ltemp UtempT ;
Lua ← Lup Aold(ilow:m, ihp1:n) ;      ! SMDM with Aold
Lua ← Lua - (Lup U(ilow:m, 1:ilow-1)) Z(1:ilow-1, ihp1:n) ;
Lua ← Lua - (Lup W(ilow:m, 1:ilow-1)) V(1:ilow-1, ihp1:n) ;
!
! Update the current row block of B
!
C ← Aold(ilow:ihi, ihp1:n) ;      ! Extract row block of Aold
C ← C - U(ilow:ihi, 1:ilow-1) Z(1:ilow-1, ihp1:n) ;
C ← C - W(ilow:ihi, 1:ilow-1) V(1:ilow-1, ihp1:n) ;
! Also need the update from the current column block
C ← C - U(ilow:ihi, ilow:ihi) Lua ;
!
! Having updated the current row block, get its
! QL factorization by calling qlt
!
[Vtemp, Lr, Ttemp] ← qlt( K, n-ihi, C ) ;
B(ilow:ihi, ihp1:n) ← Lr ; T(ilow:ihi, :) ← Ttemp ;
!
! Get the next blocks for the U, V, Z, W matrices
!
Temp ← Ttemp Vtemp ;
Z(ilow:ihi, ihp1:n) ← Lua - (Lua VtempT) Temp ;
Temp2 ← VtempT Ttemp ;
Temp3 ← Aold(ilow:m, ihp1:n) Temp2 ;
! SMDM with Aold
Temp3 ← Temp3 - U(ilow:m, 1:ilow-1) (Z(1:ilow-1, ihp1:n) Temp2) ;
Temp3 ← Temp3 - W(ilow:m, 1:ilow-1) (V(1:ilow-1, ihp1:n) Temp2) ;
W(ilow:m, ilow:ihi) ← Temp3 ;

```

```

    V(i_low:i_hi, i_hp1:n) ← V_temp ;
!
end;      ! End of loop on blocks
!
! We've eliminated all the "K" wide row blocks
!   Eliminate the rest of the columns as one block
!
i_low ← i_h(b_lks) + 1 ; i_hi ← n ;
!
! Update the current column block from A_old
!
C ← A_old(i_low:m, i_low:i_hi) ;      Extract block of A_old
C ← C - U(i_low:m, 1:i_low-1) Z(1:i_low-1, i_low:i_hi) ;
C ← C - W(i_low:m, 1:i_low-1) V(1:i_low-1, i_low:i_hi) ;
!
! Call qrl to do the QR factorization of the last column block.
!
[U_temp, R, L_temp] = qrl( m-i_low+1, n-i_low+1, C) ;
U(i_low:m, i_low:i_hi) ← U_temp ;
L2 = L_temp ;
B(i_low:m, i_low:i_hi) ← R ;
!
endfunction

```

3.2 Notes on the Algorithm

The detailed presentation of the algorithm² should allow the reader to verify the following points.

- Entries of A_{old} are not changed. The only accesses to A_{old} are for SMDM multiplications and extractions of matrix sub-blocks.
- Block Householder transformations can be used for the reduction. The implementations of qlt and qlr used but not specifically detailed use the algorithms due to Schreiber and Van Loan [30]. Alternately, the method proposed by Joffrain, Low, Quintana-Orti, Van de Geijn, and Van Zee [18] or Puglisi [27] could be used. Householder transformations are reliably orthogonal. Blocking the transformations enables use of BLAS-3.

²As presented, and given appropriate qlr and qlt functions, the algorithm closely follows an octave script implementation

- Operations updating column and row blocks and in forming the update matrices are BLAS-3. BLAS-2 operations are only in initializations and copies, and in the qlt and qlr formation of block Householder transformations. Excluding the qlt and qlr functions, the total number of BLAS-2 operations requiring $O(nm)$ flops in an elimination of all columns and $O(m+n)Kl$ flops for l eliminations of blocks of K rows and columns. If the qlt and qlr operations are BLAS-2, the total number of BLAS-2 flops is $O(mnK)$ for elimination of all columns, $O(m+n)K^2l$ for elimination of l K -sized blocks.

As presented, the algorithm “runs to completion”, useful in that the returned matrix B can be observed to have very nearly the same singular values as the input matrix, enabling a test for correct implementation.

In a more usual sparse case, l is chosen (either *a priori* as an input, or from a convergence criterion) so that the number of blocks l satisfies $l \ll n/K$, and the algorithm is ended at the “! End of loop on blocks”, the returned matrix B_{K+1} is then $lK \times lK$ with nonzero entries confined to the diagonal and K superdiagonals, B_{K+1} satisfying

$$\left(\begin{array}{c|cc} B_{K+1} & 0 & 0 \\ \hline & C_K & 0 \\ \hline 0 & & A_{updated} \end{array} \right) = \left[\prod_{i=1}^l (I - U_i)L_i(I - U_i^T) \right] (A_{old} + E) \left[\prod_{i=1}^l (I - V_i^T)T_i(I - V_i) \right] \quad (3.3)$$

In 3.3, C_K is a lower triangular $K \times K$ matrix. Denote ϵ as the largest number satisfying $1 = fl(1 + \epsilon)$. Due to the use of block Householder transformations, E satisfies

$$\|E\|/\|A\| = O(\epsilon). \quad (3.4)$$

If 3.4 holds, the $UB_{K+1}V$ decomposition is backward stable.

When the algorithm is not run to completion so that $A_{updated}$ in 3.3 exists, then we typically assume the the E term in (3.3) to be negligible compared to $\|A_{updated}\|$. Runtime estimates of $\|A_{updated}\|$ are discussed below.

3.3 Comparison to the Dense Algorithm

The following section summarizes how the algorithm has been adapted to the case of sparse matrices.

The dense algorithm would proceed by alternately eliminating column and row blocks. Consider a partitioning of the original sparse matrix where an initial column block corresponding to the first column block and an initial column block have been eliminated.

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \quad (3.5)$$

The elimination of a block of columns corresponding to A_{21} and A_{31} and an initial row corresponding to A_{12} and A_{13} has changed no entries of A . A_{11} corresponds to an upper triangular matrix B_{11} , A_{12} , A_{13} , A_{31} , and the upper triangular part of A_{12} would have been eliminated in the dense algorithm. Also the dense algorithm would have updated the trailing matrix

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix}$$

At this partition we will extract A_{22} , A_{32} as the column block to be eliminated, A_{23} as the row block to be eliminated. The multiplication by the trailing A_{33} is accomplished by multiplying AX where X has leading rows of zeros, $Y^T A$ where Y has leading columns of zeros.

3.3.1 $\hat{A}X$

In the dense algorithm we would be updating the trailing part of the matrix. But we can't actually form the updated $\hat{A} = A + UZ + WV$, as it would be dense and hence exhaust RAM. Instead compute

$$\hat{A}X = AX + U(ZX) + W(VX) \quad (3.6)$$

On the extracted blocks of A

- Perform the sequence of block Householder eliminations which had already been made to eliminate A_{21} , A_{11} , A_{12} . For example,

$$\hat{A}_{23} \leftarrow A_{23} - U(2, :) Z(:, 3) - W(2, :) V(:, 3)$$

These are all BLAS-3.

- Perform a QR (actually since this is the row block an LQ) factorization of \hat{A}_{23} .

Append new blocks to U , V , W , Z To get the new block of W , Z , multiply the currently produced blocks of dense vectors by the sparse A , e.g.

$$W(:, 3) \leftarrow V(3, :) A$$

with a similar multiplication by $U(:, 3)$.

Reducing an $m \times n$, $m \geq n$ matrix to upper bandwidth $K+1$ by Householder transformations requires $4mn^2 - 4/3n^3$ flops. For the i th elimination of row and column blocks of size K , the dense algorithm requires $4(m-iK)(n-iK)K$ flops for updates and $4(m-iK)(n-iK)K$ flops for multiplications of A by blocks of K row and column multipliers where the i th block requires

$$8K(m-iK)(n-iK) \quad (3.7)$$

flops for elimination.

The sparse algorithm for reduction differs in that instead of multiplications of the form $AU_i, V_i^T A$, A dense we instead perform multiplications

$$\begin{aligned} A_{update}U_i &= A_{sparse}U_i + U(iK+1: m, 1: iK)Z(1: iK, Ki+1: n)U_i \\ &\quad + W(iK+1: m, 1: iK)V(1: iK, Ki+1: n)U_i \end{aligned} \quad (3.8)$$

and

$$\begin{aligned} V_i^T A_{update} &= V_i^T A_{sparse}U_i + V_i^T U(iK+1: m, 1: iK)Z(1: iK, Ki+1: n) \\ &\quad + V_i^T W(iK+1: m, 1: iK)V(1: iK, Ki+1: n) \end{aligned} \quad (3.9)$$

Neglecting the sparse matrix dense matrix flops, the flop count for a completed reduction would be $6mn^2 - 2n^3$ with the incremental number of flops for the i th pair of row column blocks being

$$12K(iK)[m+n-2iK]. \quad (3.10)$$

requiring a total of

$$6(iK)^2(m+n) - 8(iK)^3 \quad (3.11)$$

flops to eliminate i row and column blocks of size K .

For the dense algorithm, required storage is independent of the number of row-column pairs eliminated. As seen in (3.3.1), initial row-column eliminations require more flops than later ones.

Conversely, for the sparse algorithm, the number of flops for the next block eliminated is proportional to i (when $iK \ll n+m$), so that the flop count is proportional to the square of the total number l of eliminated blocks. For $m=n$, the incremental flop counts for sparse and dense algorithm are equal for $iK = n/4$, so that at $n/4$ the difference in required flops is maximal. For $iK > n/4$, the dense algorithm becomes more competitive in terms of required flops.

In the dense serial algorithm, the size of matrix which can be reduced to small band form (on a single processor) depends on the available how large a

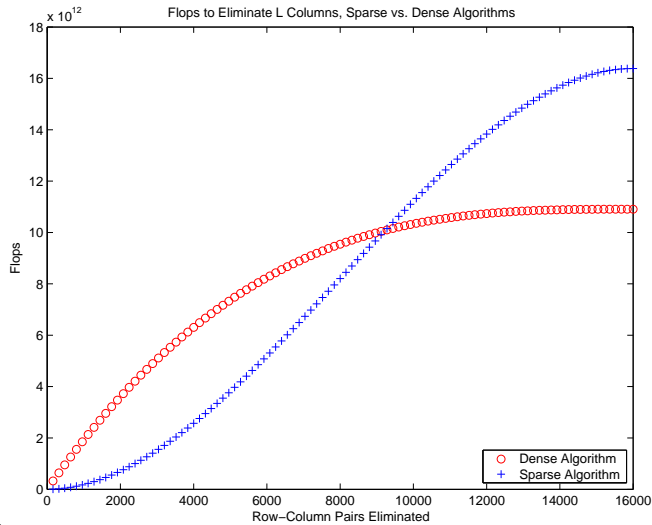


Figure 3: Sparse and Dense Flops vs. Columns Eliminated for a Matrix for Which the Dense Algorithm Requires 2 GBytes Storage

dense matrix will fit in available RAM. For a double precision “in-core” serial computation, the largest matrix we can expect to reduce with 2 GBytes of RAM is at most 16K square ³

In the sparse serial algorithm, available storage limits the number of blocks that can be eliminated. Neglecting the storage of A , eliminating l rows and columns requires $2(m+n)l$ double precision numbers stored in W , U , V and Z . When $lK = n/4$, the total storage for U , V , W , and Z is $nm/2 + n^2/2$ so is comparable. Given a 2 GByte RAM, l is inversely proportional to $m+n$. For $m+n = 100000$ (one hundred thousand), at most 1250 row-column pairs can be eliminated “in-core”; for $m+n = 1000000$ (one million), at most 125. to the mn storage required for the dense algorithm.

For large problems, larger shared memory or a distributed memory implementation may be needed to allow enough steps for the partial $UB_{K+1}V$ factorization to be useful.

4 Some Convergence Theory

If A can be permuted to diagonal blocks (is reducible), it would make more sense to compute a decomposition of each block independently. Generally, the $UB_{k+1}V$ partial decomposition will be more likely to reflect A if LU^T , V^TT ,

³K here means 2^{10} , $16K \cdot 16K \cdot 8 = 2\text{Gbytes}$, with 8 bytes per double precision number. This assumes that only U , V are stored, overwriting part of A .

are dense. Else the multiplications

$$(LU^T)A, A(V^T T) \tag{4.1}$$

(which are the only access to the trailing part of A) may not pick up all entries of A , dependent not on the size of entries of A , but only on the sparsity patterns of $V^T T$ and LU^T . Permutations of A to expose a diagonal block structure and to ensure density of U, V are desirable and will be the subject of future work.

Nevertheless, we can state some properties of the $UB_{K+1}V$ decomposition. Using block Householder transformations, we expect the overall condition number of transformations to be very near one, and expect that if we compute B_{K+1} to satisfy $A = UB_{K+1}V$, then the singular values of B_{K+1} and A will be closely matched. If only a partial decomposition is carried out i.e., if for A of m rows and n columns we compute only the first N rows and columns B_{K+1} , $N < n$, how the $N \times N$ truncated matrix B_{K+1}^K useful? The following subsections address interlacing of singular values as N increases and also approximation of A by $U_N B_{K+1} V_N$ in the Frobenius norm.

4.1 Interlacing of Singular Values

In the dense case, singular values are typically found by reducing an $m \times n$ A matrix to a condensed $m \times n$ matrix B (upper triangular, or with banded structure such as bidiagonal) with the same singular values, then finding the singular values of the condensed form by some iterative procedure.

Adapting the reduction to small band (or upper triangular) form to the sparse case, obtaining an $m \times n$ reduced matrix is impractical if the transformations are stored, unstable if they are not stored (the transformations must be stored to maintain orthogonality, linear independence, and more generally to ensure that rank of the transformations continues to increase as more transformational steps are taken).

It's natural then to try to use the singular values of an $N \times N$ reduced matrix B_K obtained after "eliminating" N columns as approximate singular values of the original matrix A . The singular values of B_K are sometimes called Ritz values of A . Cauchy's interlacing property relates the the Ritz values to the singular values of A .

Cauchy's Interlace Theorem [26]: Let C be a Hermitian matrix partitioned as

$$C = \begin{bmatrix} H & B^* \\ B & U \end{bmatrix}, \quad \begin{array}{l} C \text{ is } n \times n \\ H \text{ is } L \times L \end{array}$$

where C has eigenvalues:

$$\alpha_1 \leq \alpha_2 \leq \dots \alpha_n$$

and H has eigenvalues

$$\theta_1 \leq \theta_2 \leq \dots \theta_L$$

Then for $j = 1, \dots, N$,

$$\alpha_j \leq \theta_j \leq \alpha_{j+n-L} \tag{4.2}$$

and for $l = 1, 2, \dots, n$,

$$\theta_{l-n+L} \leq \alpha_l \leq \theta. \tag{4.3}$$

Suppose A has been transformed to A_L

$$A_L = \left[\begin{array}{c|c} R_L & T_L \\ \hline 0 & B_L \end{array} \right]$$

Then The Hermitian matrix $A_L^T A_L$

$$A_L^T A_L = \left[\begin{array}{c|c} R_N^T R_N & R_N^T T_N \\ \hline T_N^T R_N & T_N^T T_N + B_N^T B_N \end{array} \right]$$

has the same eigenvalues as $A^T A$. For the symmetric matrix $A_N^T A_N$, Cauchy's interlacing theorem implies that the eigenvalues of the symmetric matrix $R_N^T R_N$ interlace with those of $A^T A$. Since the singular values σ_i of A have the same ordering in size as the eigenvalues $\lambda_i = \sigma_i^2$ of $A^T A$, Cauchy's interlacing value theorem interlaces the singular values of A and R_L .

As an example of interlacing consider singular values of of the 4 by 4 upper triangular matrix

$$T = \begin{bmatrix} 4 & 3 & 2 & 0 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Let T_1, T_2, T_3 be the upper left 1×1 , 2×2 , 3×3 matrices respectively. The singular values of T_1, T_2, T_3, T are respectively

$$\begin{array}{cccc} 4 & 5.3890 & 6.0959 & 6.13336 \\ & 2.2267 & 2.5356 & 2.8331 \\ & & 1.5527 & 1.62131 \\ & & & .85183 \end{array}$$

Actually, we can do a bit more. When reducing a banded upper triangular form, we get A_L of the form

$$A_N = \left[\begin{array}{c|c|c} R_N & L_N & 0 \\ \hline 0 & B & C \\ \hline 0 & D & E \end{array} \right] \tag{4.4}$$

and we naturally wonder whether singular values of

$$\hat{R} = [R_L \mid L_L \mid 0]$$

are related to those of A . The following result of Kahan from P. 196 [26] can be applied.

The Residual Interlace Theorem. Let F be a Hermitian matrix of the form

$$F = \left[\begin{array}{c|c|c} H & C^* & 0^* \\ \hline C & V & Z^* \\ \hline 0 & Z & W \end{array} \right]$$

where H is $N \times N$, V is $j \times j$, F is $n \times n$.

Define

$$M(X) = \left[\begin{array}{c|c} H & C^* \\ \hline C & X \end{array} \right]$$

where $V - X$ is assumed to be invertible. Denote the eigenvalues of $M(X)$ as

$$\mu_1 \leq \mu_2 \leq \dots \leq \mu_{j+N}$$

Then each interval $[\mu_i, \mu_{i+N}]$, $i = 1, \dots, j$ contains a different eigenvalue α_i of F . Also, outside each open interval (μ_l, μ_{l+j}) , $l = 1, \dots, N$, there is a different eigenvalue α_N of F .

The residual interlace theorem applies to $A_N^T A_N$ as it is of the form (suppressing the N subscripts)

$$A_N^T A_N = \left[\begin{array}{c|cc} R^T R & R^T L & 0 \\ \hline L^T R & L^T L + B^T B + D^T D & B^T C + D^T E \\ \hline 0 & C^T B + E^T D & C^T C + E^T E \end{array} \right]$$

Taking $X = L^T L$ gives $X - V = B^T B + D^T D$. The theorem will apply if $X - V$ is nonsingular, which will be the case when either the columns of B or the columns of D are linearly independent.

We conclude that the $j + N$ singular values α_i of \hat{R} taken as the square roots the eigenvalues of $M(L^T L)$ are lower bounds for the top $j + N$ singular values of A . In particular if α_i , $i \leq N$ is the i th largest singular value of \hat{R} , then $\alpha_i < \sigma_i$, where σ_i is the i th largest singular value of A . Applying the interlacing theorem to R and \hat{R} the i th of L largest singular values of \hat{R} is larger than the i th singular value η_i of R .

Since $\eta_i \leq \alpha_i \leq \sigma_i$, the singular values α_i of \hat{R} are better estimates of singular values of A than are the singular values of η_i of R .

For example take

$$A = \begin{bmatrix} 4 & 3 & 2 & 0 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

Let R_1, R_2, R_3 be the upper left $2 \times 2, 2 \times 3, 2 \times 4$ matrices respectively. The first two singular values of R_1, R_2, R_3, A are respectively

$$\begin{array}{cccc} 5.3890 & 6.0220 & 6.0422 & 6.1452 \\ 2.2267 & 2.3949 & 2.5479 & 2.7981 \end{array}$$

4.2 Approximation of A by $J_{Kl} = U_{Kl} \hat{B}_{K+1} V_{Kl}^T$

Take $N = Kl$. In exact arithmetic, $U_{Kl} A V_{Kl}^T = A_{Kl}$ where U_{Kl} and V_{Kl} are orthogonal. Due to the orthogonality of U_{Kl} and V_{Kl}

$$\|A_{Kl}\|_F = \|A\|_F$$

Because of the use of block Householder transformations, U_{Kl} and V_{Kl} are very nearly orthogonal even in rounding arithmetic.

Recalling (3.3), simplify the partitioning as

$$A_{Kl} = \left[\begin{array}{c|c} B_{K+1} & C_K \\ \hline 0 & \hat{A}_{Kl} \end{array} \right] \quad (4.5)$$

In practice, \hat{A}_{Kl} is not computed as it would be dense and large and likely to overflow the RAM. Since

$$\begin{aligned} \|A\|_F^2 &= \|A_{Kl}\|_F^2 = \|B_{K+1}\|_F^2 + \|C_K\|_F^2 + \|\hat{A}_{Kl}\|_F^2, \\ \|\hat{A}_{Kl}\|_F^2 &= \|A\|_F^2 - \|B_{K+1}\|_F^2 - \|C_K\|_F^2. \end{aligned} \quad (4.6)$$

Take $\hat{B}_{K+1} = [R_K | C_K]$ as a $Kl \times K(l+1)$ matrix and $J_{Kl} = U_K \hat{B}_{Kl} V_K^T$ as a rank Kl approximation to A . The accuracy of the approximation can be gauged by (4.6) with the quantities on the right hand side easily computable.

5 Some Benchmark Problems

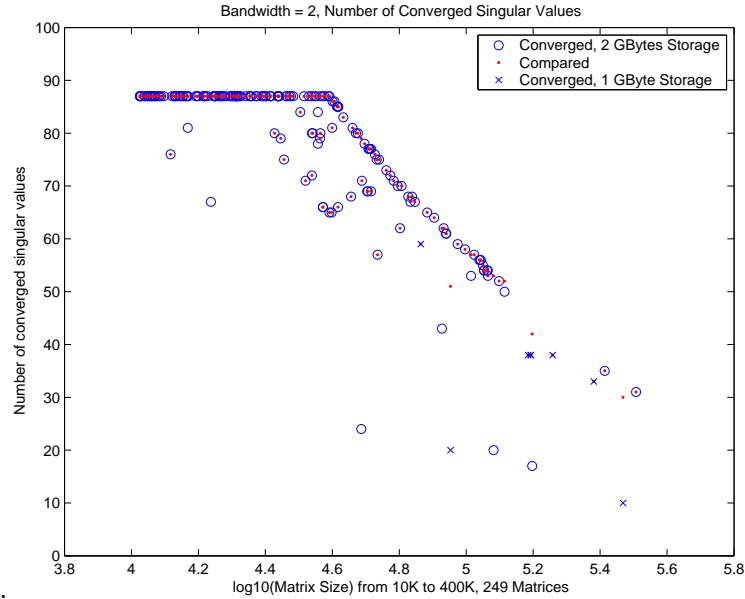
As with Householder implementations of GMRES and ARPACK, $UB_{K+1}V$ is a stable algorithm which in the sparse case is not (due to storage constraints or convergence) run to completion. Limiting the computation to use 2 GByte of dimensioned space, we can test a Fortran implementation of $UB_{K+1}V$ against LAPACK. Comparison can be made for matrices to size around 3000×3000

LAPACK dgesvd gives the same singular values for A and B_{K+1} to high accuracy.

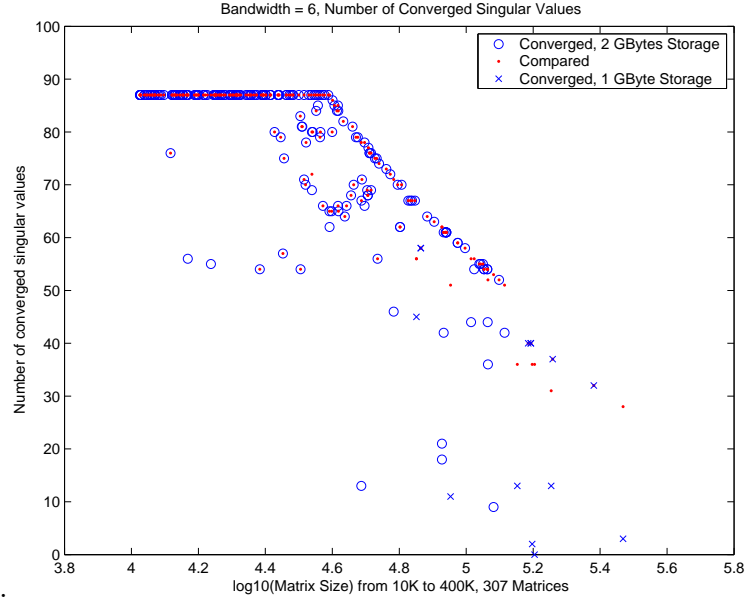
To get some feel for the reliability of the algorithm, we downloaded the largest matrices from the UF Sparse collection. Compiling Fortran code with g77, “relocation errors” and “segmentation faults” limited problem size to about $m + n < 800000$, (m rows, n columns), so that a few dozen matrices were too large for this compiler. The other limitation was that the $UB_{K+1}V$ implementation currently requires conversion of a sparse matrix to a blocked column form. Conversion from coordinate to blocked column form did not succeed when a row or column was full.

For bandwidths 2, 6, and 12 we computed singular values for all the unsymmetric or rectangular matrices with $2 \times 10^4 < m + n < 8 \times 10^5$. The results for bandwidth 6 and 12 include integer valued matrices. In each case, the number of steps N is calculated so that the $2N(m + n)$ double precision numbers allocated for W, U, V, Z plus the nz elements of the sparse matrix A require less than 2 GBytes of storage (taking 8 bytes of storage per double precision number). For each matrix the code is recompiled to reset parameters for matrix dimensioning. For some of the larger matrices, g77 compiled code suffers “relocation” errors at compile time, or run time “segmentation faults”. These instances were recompiled to use one GBytes for matrix storage, and rerun. For each bandwidth $K = 2, 6, 12$ around 300 matrices successfully ran. In each instance, the Frobenius norm of B_{K+1}^N was less than or equal to the Frobenius norm of A , with near equality in some cases.

Converged Singular Values for Bandwidth 2



Converged Singular Values for Bandwidth 6



In each computation, singular values computed of B_{K+1}^N were computed by three calls to dgesvd. One call was for the entire $N \times N$ matrix. One call was for the upper left matrix square submatrix B_1 of dimension $\min(N - K, N - 6)$. The last call is for the upper left $N/2 \times N/2$ matrix. The largest $L = 2\sqrt{(N)} + 10$ (N inversely proportional to $m + n$) singular values of were compared.

Let $\sigma_1 \geq \sigma_2 \geq \dots \sigma_L$ be the largest $\sqrt{(N)}$ singular values of B_{K+1}^N , and $\hat{\sigma}_1 \geq \hat{\sigma}_2 \geq \dots \hat{\sigma}_L$ the largest singular values of B_1 . σ_i was said to be converged if

$$\frac{|\sigma_j - \hat{\sigma}_j|}{|\sigma_j|} < 10^{-8}, \forall j, 1 \leq j \leq i$$

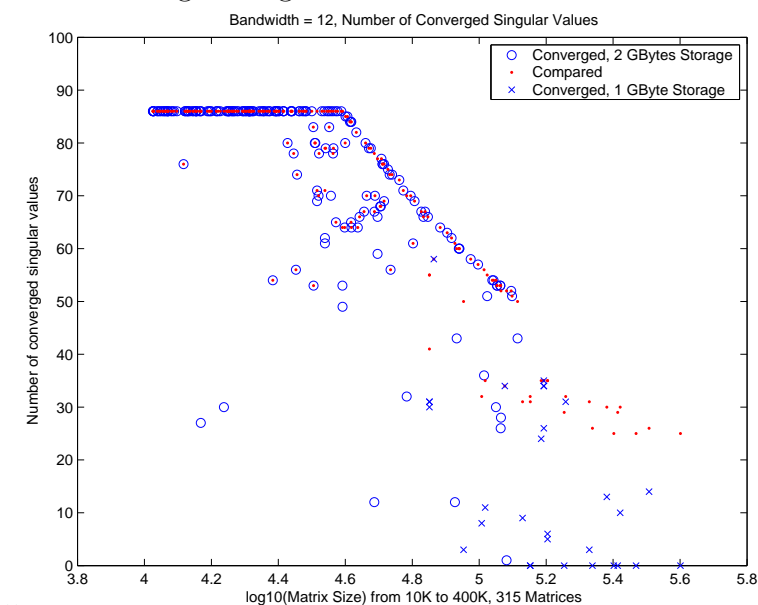
Converged singular values for different bandwidths agreed to high accuracy.

Figures 4 to 6 plot the number of singular values converged for matrices of sizes $(m + n)/2$ from 10000 to about 400000. The legend "." represents the number of singular values compared, "o" represents the number converged for 2 Gbytes of storage. "x" represents the number of converged singular values for 1 GBytes of storage. For the largest matrices represented only about 40 right and left basis vector could be computed. The maximal number of computed basis vectors was 1500 (representing the flat part of the plots).

The isolated "o"s and "x"s indicate instances for which few singular values converged than were compared.

For a high proportion of the test matrices, all the $L = 2\sqrt{N}i + 10$ singular values converged. For bandwidth 2, for 238 of 250 matrices, all the compared

Converged Singular Values for Bandwidth 12



singular values converged. The minimal number of converged singular values was 10 of 30 compared. For bandwidth 6, for 278 of 308 matrices, all compared singular values converged. In one instance there were no converged singular values of 36 compared. Two other instances of poor convergence were 2 of 36 and 3 of 28. All the worst cases were when only 1 GByte of storage was used. Also these cases tended to be the matrices of higher dimension (for which the size N of B_7 was relatively small) The next worst was 9 of 53. For a bandwidth of 12, singular values were computed for 315 matrices. 32 of these had suffered "relocation" errors or runtime segmentation faults for 2 GBytes of storage, so were rerun allowing 1 GBytes for storage. For 259 of 315 matrices, all the compared singular values converged. There were many instances of no converged singular values, especially for large matrices and in the case that only 1 GByte of storage could be used.

The plots seem to show that higher bandwidths K may slow convergence due to having multiplied by A only N/K in constructing a basis size N (lower degree Krylov subspace). Future work will explore the trade-off between

- high bandwidths allowing faster computations
- and low bandwidths which for a given number of computations and allocated memory may allow determination of more singular values.

6 Solving Least Squares Problems

It's natural to use a reduction to small band form to solve least squares problems. A classic and beautiful algorithm for this purpose is LSQR, due to Paige and Saunders [25]. LSQR uses the sparse Golub-Kahan Lanczos bidiagonalization procedure in a three term recurrence to find x to minimize $\|Ax - b\|_2$.

- LSQR requires minimal storage (3 left and 3 right vectors) and accesses A only for multiplications Ax , $y^T A$. But as seen in Figure 1.1, there is a significant loss of rank if the basis is not reorthogonalized, which can cause convergence to “stall”. Reorthogonalization entails saving the basis vectors.
- The sparse Ax and $A^T y$ operations are rather slow. The rest of the LSQR operations are BLAS-1.
- Computing Ax and $A^T y$ in a massively parallel manner may not be computationally efficient due to the frequent synchronization required.

The development given here corresponds to the development of the LSQR algorithm from Householder bidiagonalization. We accomplish a partial reduction to small-band form of the augmented matrix $[b|A]$, and after l steps have

$$U_{l+k}^T A V_l = H_l,$$

H_l an upper Hessenberg matrix with bandwidth k above the diagonal. and with $U_{l+d} \beta e_1 = b$. Given y_l , let

$$x_l = V_l y_l \tag{6.1}$$

$$r_l = b - A x_l$$

$$t_l = \beta e_1 - H_l y_l$$

Then,

$$\begin{aligned} U_{l+d} t_l &= U_{l+d} (\beta e_1 - H_l y_l) \\ &= b - A V_k y_k = b - A x_k = r_k \end{aligned}$$

Choose y_l to minimize $\|t_l\|_2$,

$$t_l = \beta e_1 - H_l y_l \tag{6.2}$$

We have $U_{l+d} t_l = r_l = b - A x_l$ where in exact arithmetic U_{l+d} is orthogonal so that $\|r_l\|_2 = \|t_l\|_2$. In practice, when U_{l+d} is determined from Householder transformations, it is orthogonal to working numeric precision.

Choosing x_l from (6.2) and (6.1) results in an algorithm with several valuable properties.

- The only accesses to A are for extractions of blocks and for SMDM multiplications. Accessing A only for matrix vector multiplications is a frequently cited virtue for LSQR and also for Krylov based methods, see for example Saad [29].
- The other operations to produce the $U[b|A]V^T$ reduction are almost entirely BLAS-3. In contrast, most Krylov matrix solvers such as GMRES, are BLAS-2, where typically BLAS-3 algorithms are obtained only by solving for several RHS simultaneously, as in Langou [21], Guennouni, Jbilou, and Sakok [13], or Baker, Dennis, and Jessup, [2]. BLAS-3 is helpful in serial speedup on cache based machines and also in reducing communication frequency so that the algorithm can efficiently use many processors, see for example Hernandez, Roman, and Tomas [14].
- If produced by block Householder transformations, the algorithm is scalably stable, independent of block size k and the particular choice of a matrix. In contrast, methods which attempt to simultaneously compute $Ax, A^2x, A^3x, \dots A^kx$ depend both on properties of A to ensure parallel computation and on limiting the size of k to avoid loss of rank in the Krylov basis, see for example Demmel, Hoemmen, Mohiyudding, and Yelick [6], [7].

A prototype implementation checks the accuracy of the above algorithm by “running to completion”, i.e., eliminating all n columns via an octave script. The computing solution x minimizing $\|Ax - b\|$ closely matches the octave QR solution. Current work is in implementing good convergence criteria for a practical sparse algorithm.

References

- [1] J. Angeli, O. Basset, C. Fulton, G. Howell, R. Hsuand A. Sawetprawhickal, M. Schuster, D. Richardson, H. Thompson, and S. Wilberscheid. Some issues in efficient implementation of a vector based module for document retrieval, June 2001. <http://www.ncsu.edu/itd/hpc>.
- [2] A. Baker, J. Dennis, and E. Jessup. On improving linear solver performance: A block variant of GMRES, 2004. Submitted to SIAM Journal on Scientific Computing. Also available from <http://www.osti.gov/energycitations/purl.cover.jsp?purl=/881896-hwkWn2/>.
- [3] M. Berry, T. Do, G. O’Brien, V. Krishna, and S. Varadhan. SVDPACKC: Version 1.0 user’s guide. Technical Report Tech. Report CS-93-194, University of Tennessee, Knoxville, TN, October 1993.

- [4] J. Choi, J. Dongarra, and D. Walker. The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form Cholesky factorization routines. *Num. Alg.*, 10:379–399, 1995. LAPACK Working Note # 92.
- [5] J. Demmel, L. Grigori, M. Hoemmem, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89,lawn204, University of California, <http://www.netlib.org/lapack/lawns/downloads/>, August 2008.
- [6] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in computing krylov subspaces. Technical Report Technical Report UCB/EECS-2007-123, University of California, October 2007.
- [7] J. W. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE, April 2008.
- [8] A. A. Dubrulle. On block Householder algorithms for the reduction of a matrix to Hessenberg form. *Supercomputing 88. Vol.II: Science and Applications. Proceedings, IEEE Explore*, 2:129–140, Nov. 1988.
- [9] L. Giraud and J. Langou. Robust selective Gram-Schmidt reorthogonalization. Technical Report TR/PA/02/52, CERFACS, Toulouse, FR, 2002.
- [10] G. Golub and W. Kahan. Calculating the singular values and psuedo-inverse of a matrix. *SIAM J. Num. Anal.*, 2:205–224, 1965.
- [11] G. Golub, F. Lusk, and M. Overton. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Trans. Math. Soft.*, 7:147–169, 1981.
- [12] B. Grösser and B. Lang. Efficient parallel reduction to bidiagonal form, 1998. Preprint BUGHW-SC 98/2 (Available from <http://www.math.uni-wuppertal/>).
- [13] A. El Guennouni, K. Jbilou, and H. Sadok. A block Lanczos method for linear systems with multiple right-hand sides. *Appl. Numer. Math.*, 51:243–256, 2004.
- [14] V. Hernandez, J. Roman, and A. Tomas. Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement. *Parallel Computing*, 33:521–540, 2007.

- [15] G. Howell, J. Demmel, C. Fulton, S. Hammarling, and K. Marmol. BLAS 2.5 Householder bidiagonalization. *ACM Transactions on Mathematical Software*, 34(3):13–46, May 2008. Preliminary version available as LAPACK Working Note 174.
- [16] E. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, University of California, Berkeley, 2000.
- [17] W. Jalby and B. Philippe. Stability analysis and improvement of the block Gram-Schmidt algorithm. *SIAM J. Sci. Stat. Comput.*, 12(5):1058–1073, 1991.
- [18] T. Joffrain, T. M. Low, E. S. Quintana-Orti, R. Van de Geijn, and F. G. Van Zee. Accumulating Householder transformations, revisited. *ACM Trans. on Math. Software*, 32(2):169–179, 2006.
- [19] L. Kaufman. Application of dense Householder transformation to a sparse matrix. *ACM Trans. on Math. Software*, 5(4):442–450, 1979.
- [20] B. Lang. Parallel reduction of banded matrices to bidiagonal form. *Parallel Comput.*, 22:1–18, 1996.
- [21] J. Langou. *Iterative methods for solving linear systems with multiple right-hand sides*. PhD thesis, INSA, Toulouse, 2003.
- [22] R. Larsen. PROPACK, software package for sparse SVD. Available from <http://soi.stanford.edu/~rmunk/PROPACK/>.
- [23] R.-M. Larsen. *Lanczos bidiagonalization with partial reorthogonalization*. PhD thesis, Dept. Computer Sci., University of Aarhus, 1998.
- [24] R. Lehoucq, D. Sorensen, and C. Yang. ARPACK users’ guide, solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods. SIAM, 1998.
- [25] C. Paige and M. Saunders. An algorithm for sparse linear equations and sparse least squares. *ACM Trans. on Math. Software*, 8(43–71), 1982.
- [26] B. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, 1980.
- [27] C. Puglisi. Modification of the householder method based on the compact wy representation. *SIAM J. Sci. Statist. Comput.*, 13(3):723–726, 1992.
- [28] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18:297–311, March 2007.

- [29] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd. Ed.* SIAM, 2003.
- [30] R. Schreiber and C. F. Van Loan. A storage-efficient WY^T representation for products of Householder transformations. *SIAM Scientific and Statistical Computing*, 10:53–57, 1989.
- [31] D. Simon and H. Zha. Low-rank matrix approximation using the Lanczos bidiagonalization process with applications. *SIAM J. Sci. Computing*, 21(6):2257–2275, 2000.
- [32] M. Sosonkina, D. C. S. Allison, and L. T. Watson. Scalable parallel implementations of the GMRES algorithm via Householder reflections. In *Proc. Intern. Conf. on Parallel Processing*, pages 396–404. IEEE Explore, 10-14 Aug. 1998.
- [33] G. W. Stewart. The Gram-Schmidt algorithm and its variations. Technical Report TR-4642, Department of Computer Science, University of Maryland, December 2004.
- [34] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6), 1997.
- [35] D. Vanderstraeten. A stable and efficient parallel block Gram-Schmidt algorithm. In *Euro-Par'99, Lecture Notes Computer Science, No. 1685*, pages 1128–1135. Springer-Verlag, 1999.
- [36] R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Proceedings of SCIDAC 2005, Journal of Physics: Conference Series*. SCIDAC, June 2005.