Cache Efficient Bidiagonalization Using BLAS 2.5 Operators

Howell, G. W. and Demmel, J. W. and Fulton, C. T. and Hammarling, S. and Marmol, K.

2006

MIMS EPrint: **2006.56**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Cache Efficient Bidiagonalization Using BLAS 2.5 Operators*

G. W. Howell
North Carolina State University
Raleigh, North Carolina 27695
J. W. Demmel
University of California, Berkeley
Berkeley, California 94720
C. T. Fulton
Florida Institute of Technology
Melbourne, Florida 32901
S. Hammarling
Numerical Algorithms Group
Oxford, United Kingdom
K. Marmol
Harris Corporation
Melbourne, Florida 32901

March 23, 2006

*This research was supported by National Science Foundation Grant EIA-0103642.

Abstract

On cache based computer architectures using current standard algorithms, Householder bidiagonalization requires a significant portion of the execution time for computing matrix singular values and vectors. In this paper we reorganize the sequence of operations for Householder bidiagonalization of a general $m \times n$ matrix, so that two (_GEMV) vector-matrix multiplications can be done with one pass of the unreduced trailing part of the matrix through cache. Two new BLAS 2.5 operations approximately cut in half the transfer of data from main memory to cache. We give detailed algorithm descriptions and compare timings with the current LAPACK bidiagonalization algorithm.

1 Introduction

A primary constraint on execution speed is the "von Neumann" bottleneck in delivering data to the CPU. Most current computer architectures store data hierarchically. Data in a small number (from a few dozen to several hundred) registers can be used for computations in the current or next clock cycle. Data in several levels of cache memory is available in at most a few clock cycles. Accessing data in main memory requires several dozen or hundred clock cycles [13, 15, 12]. Matrices larger than a few hundred square are typically too large to fit in cache memory and must be stored in main storage (RAM). For example, a comparatively large 8 Mbyte L3 cache would be filled by a double precision 1K by 1K matrix. Reads and writes of a number to RAM (even reads and writes organized so that the data bus is working at full bandwidth) are typically much slower than floating point operations. For example, a 533 MHz 128 bit bus can deliver 1.06 Billion double precision numbers per second from RAM. If the bus feeds two processors which can perform 12 billion flops/sec, then about 12 flops per number fetched are needed to achieve peak computational speed.

The von Neumann bottleneck motivates the algorithmic rearrangements of this paper. Since bidiagonalization algorithms are constrained by data transfer, reducing data transfer decreases computation time. Most of the algorithms given here have been implemented for inclusion in the LAPACK library, which is designed to efficiently perform dense matrix computations on cache based architectures.

The LAPACK library [1] uses the Basic Linear Algebra Subprograms (BLAS).[1] For matrices too large to fit in cache memory, LAPACK routines

[1] LAPACK and reference BLAS routines can be downloaded from Netlib. The reference BLAS libraries give correct results but do not run very fast. Vendor tuned BLAS libraries

using tuned BLAS usually execute much faster than the older LINPACK or EISPACK routines. For LU and QR decomposition, almost all computations are BLAS-3 matrix matrix multiplications (_GEMMs), for which many floating point operations can be performed for each number transferred from RAM to cache. When tuned BLAS are used, LAPACK LU and QR decompositions run at nearly the peak theoretical computational speed.

BLAS-2 matrix vector multiplications are more constrained by data bus bandwidth. For a matrix too large for cache, only one add and multiply is performed for each element fetched from cache. For LAPACK operations such as reducing a symmetric matrix to similar tridiagonal form, reducing a general square matrix to similar Hessenberg form, or reducing a rectangular matrix by Householder transformations to bidiagonal form, only about half the operations are BLAS-3, with almost all the rest in BLAS-2 matrix vector multiplications. For matrices too large to fit in cache, these algorithms run at rates well below theoretical peak.

K. Stanley [24] showed that BLAS 2.5 operators which combine two or more BLAS-2 operations, e.g., a matrix vector and transposed matrix vector multiply performed simultaneously, can halve data transfer in tridiagonalization of symmetric matrices. Here we use BLAS 2.5 operators to halve data transfer in Householder bidiagonalization of a rectangular matrix, compared to the current LAPACK implementation.[2]

The two BLAS 2.5 routines we use are _GEMVER, which performs the operations:

$$\hat{A} \leftarrow A + u_1 v_1^T + u_2 v_2^T$$

$$x \leftarrow \beta \hat{A}^T y + z$$

$$w \leftarrow \alpha \hat{A} x$$

and _GEMVT, which performs the operations:

$$x \leftarrow \beta A^T y + z$$

give much faster runs. The ATLAS project allows the user to create their own tuned BLAS, see: (http://math-atlas.sourceforge.net/).

[2]Householder bidiagonalization of an $m \times n$ matrix, $m > n$ requires $4n^2m - 4/3n^3$ flops. Since determining the singular values requires only an additional $O(n^2)$ operations, savings in bidiagonalization time are also savings in determination of singular values. Formerly, determination of singular vectors was more time consuming than bidiagonalization. In the current version of LAPACK, singular vectors are determined in time comparable to bidiagonalization. Dhillon and Parlett's work [14, 22, 10] further speeds the process of determining singular vectors, so that bidiagonalization is the predominant computation in singular value decomposition.

Algorithm	BLAS-2	_GEBRD BLAS-2 and 3	Algorithm I BLAS 2.5	Algorithm III BLAS 2.5 and 3
READS	4	2 + 2/k	1	1 + 2/k
WRITES	2	2/k	1	2/k

Table 1: Floating Point Reads and Writes of the Trailing Matrix in One Column-Row Elimination

$$w \leftarrow \alpha A x$$

Specifications for these two routines are part of the new BLAS standard [7, 8].

This paper details variants of the classic Golub and Kahan Householder bidiagonalization algorithm [16, 17]. Section 2 describes the reduction in terms of Level 2 BLAS. Section 3 shows how the two BLAS 2.5 routines lessen data transfer. Section 4 describes Algorithm I bidiagonalization using the Level 2.5 routine _GEMVER, which works well in terms of reads, but not so well in terms of writes. Section 5 presents Algorithm II, which reduces the leading k rows and columns of a matrix to bidiagonal form, using the Level 2.5 routine _GEMVT. Section 6 uses Algorithm II in order to develop Algorithm III, half BLAS-2.5 and half BLAS-3. Section 7 gives some timing results and discusses tuning Algorithm III for cache size; Section 8 reports some results of running our new version of routine _GEBRD through the testing routines provided in the LAPACK distribution and discusses the algorithm variants for the complex case and the case of more columns than rows. Finally, Section 9 summarizes our work and compares it to the bidiagonalization proposed by B. Grösser and B. Lang.

Table 1 summarizes the results of the paper by comparing the required data transfer for several Householder bidiagonalization algorithms in terms of the frequency of reads from main memory to cache, and writes from cache to main memory. The table assumes that a trailing part of a matrix is too large too fit in cache memory. Different arrangements of Householder bidiagonalization have markedly different levels of data transfer.

2 Algorithm BLAS-2 – Classical Reduction to Bidiagonal Form

In this section, we express Householder bidiagonalization as a BLAS-2 level algorithm alternating matrix vector multiplies with rank-one updates. The

BLAS-2 algorithm corresponds to the Householder bidiagonalization first introduced by Golub and Kahan [16][3] in 1965. The BLAS-2 algorithm is simple to implement, but for matrices too large to fit in cache has more data transfer than necessary.

Let A be an $m \times n$ matrix with $m \geq n$. First select a left Householder vector $u^{(1)}$ of length m such that

$$\hat{A}_1 = \left(I - \tau_{q_1} u^{(1)} u^{(1)^T}\right) A = \begin{pmatrix} x & x & \dots & x \\ 0 & x & \dots & x \\ \vdots & \vdots & & \vdots \\ 0 & x & \dots & x \end{pmatrix} \tag{2.1}$$

has zeros below the diagonal in the first column, and then a right Householder vector $v^{(1)}$ of length n such that

$$\left(I - \tau_{q_1} u^{(1)} u^{(1)^T}\right) A \left(I - \tau_{p_1} v^{(1)} v^{(1)^T}\right) = \begin{pmatrix} x & x & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & & \vdots \\ 0 & x & \dots & & x \end{pmatrix} \tag{2.2}$$

has zeros to the right of the superdiagonal element of the first row. Here we assume the LAPACK _GEBRD normalization $u^{(1)}(1) = 1$ and (since $v^{(1)}$ has a leading zero in the first component) $v^{(1)}(2) = 1$. This is the first column-row elimination. The second step is to zero the second column and row. After i–1 steps the leading $(i$–$1) \times (i$–$1)$ matrix B_{i-1} of A is bidiagonal, with structure illustrated by:

$$A^{(i)} = \left(\begin{array}{c|c|c} & \text{col } i & \\ & 0 & \\ B_{i-1} & & 0 \\ & x & \\ \hline & x & \\ 0 & \vdots & A_i \\ & x & \end{array}\right)$$

[3]The Golub and Kahan version of course pre-dates the BLAS and LAPACK Householder operator _LARFG, but using these ideas allows the introduction of notation used throughout.

$$= \left(\begin{array}{ccccc|c|c} & & & & & \text{col } i & \\ \hline d_1 & e_1 & & 0 & & & \\ & d_2 & e_2 & & & & \\ & & d_3 & e_3 & & 0 & 0 \\ & & & \ddots & \ddots & & \\ & & 0 & & d_{i-1} & e_{i-1} & \\ \hline & & & & & x & \\ & & & & & \vdots & \\ & & & 0 & & \vdots & A_i \\ & & & & & x & \end{array}\right) \tag{2.3}$$

where $A_i = A^{(i)}$(i:m,i+1:n).[4]

In the case when $m > n$, there are n columns to eliminate and $n-2$ rows. The final bidiagonal matrix satisifies

$$B = Q^T A P \tag{2.4}$$

where

$$Q^T = \prod_{i=1}^{n} H_i, \quad P = \prod_{i=1}^{n-2} G_i \tag{2.5}$$

and

$$H_i = I - \tau_{q_i} u^{(i)} u^{(i)^T}, \; G_i = I - \tau_{p_i} v^{(i)} v^{(i)^T}. \tag{2.6}$$

Here the left Householder vectors $u^{(i)}$ of length m have $i-1$ leading zeros, and the right Householder vectors $v^{(i)}$ of length n have i leading zeros, that is,

$$u^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ u_i \end{pmatrix} \quad \text{and} \quad v^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ v_i \end{pmatrix}, \tag{2.7}$$

where the normalization for u_i and v_i is $u_i(1) = 1$, $v_i(1) = 1$. The ith step of the BLAS-2 reduction is accomplished by two _LARFG calls to generate the left and right Householder vectors, plus four BLAS-2 operations. Introducing some notation useful in the rest of the paper, the ith column elimination $(I_{m-i+1} - \tau_{q_i} u_i u_i^T)$ A(i:m,i+1:n) is[5]

[4] We use MATLAB style notation for which A(i:m,i+1:n) denotes the submatrix of A consisting of rows i to m and columns $i+1$ to n. In this and subsequent algorithms we make the simplifying assumption $m \geq n$, where m is the number of rows and n the number of columns of A.

[5] At the right we list the corresponding BLAS operation.

1. $y_i^T \leftarrow u_i^T\ A$(i:m, i+1:n) BLAS-2 _GEMV
   followed by the rank-one update,
2. A(i:m,i+1:n) $\leftarrow A$(i:m,i+1:n)$-\tau_{q_i} u_i y_i^T$, BLAS-2 _GER

and the ith row elimination A(i+1:m,i+1:n)$(I_{n-i} - \tau_{p_i} v_i v_i^T)$ is

3. $w_i \leftarrow A$(i+1:m,i+1:n)v_i BLAS-2 _GEMV
   followed by the rank-one update
4. A(i+1:m,i+1:n) $\leftarrow A$(i+1:m, i+1:n)$-\tau_{p_i} w_i v_i^T$. BLAS-2 _GER

Following from (2.3), the result of the ith column and ith row elimination is

$$A^{(i+1)} = (I - \tau_{q_i} u^{(i)} u^{(i)^T}) A^{(i)} (I - \tau_{p_i} v^{(i)} v^{(i)^T})$$

$$= \left(\begin{array}{c|c|c} & \text{col } i & \\ & 0 & \\ B_i & & 0 \\ & e(i) & \\ \hline & x & \\ 0 & \vdots & A_{i+1} \\ & x & \end{array}\right) \tag{2.8}$$

where $A_{i+1} = A$(i+1:m,i+2:n). In terms of the left Householder vector u_i of length $m-i+1$ and the right Householder vector v_i of length $n-i$ from (2.7),

$$H_i' = (I - \tau_{q_i} u_i u_i^T),\ \ G_i' = (I - \tau_{p_i} v_i v_i^T).$$

In terms of these $m-i+1 \times m-i+1$ and $n-i \times n-i$ Householder matrices the formula (2.8) updating the trailing, unreduced part of the A-matrix is

$$H_i' A_i G_i' = \quad (I - \tau_{q_i} u_i u_i^T) A(i:m, i+1:n) (I - \tau_{p_i} v_i v_i^T) \tag{2.9}$$

$$= \left[\begin{array}{c|ccc} e_i & 0 & \dots & 0 \\ x & & & \\ \vdots & & A_{i+1} & \\ x & & & \end{array}\right] \tag{2.10}$$

Pseudo Code for Algorithm BLAS-2

For $i = 1 : n-1$,

1. Construct u_i of length $m-i+1$ to zero
elements A(i+1:m,i) _LARFG
2. $y_i^T \leftarrow u_i^T\, A$(i:m,i+1:n) BLAS-2 _GEMV
3. A(i:m,i+1:n) $\leftarrow A$(i:m,i:n) $- \tau_{q_i} u_i y_i^T$ BLAS-2 _GER
If $(i < n-1)$
4. Construct v_i of length $n-i$ to zero
elements A(i,i+2:n) _LARFG
5. $w_i \leftarrow A$(i+1:m,i+1:n) v_i BLAS-2 _GEMV
6. A(i+1:m,i+1:n) $\leftarrow A$(i+1:m,i+1:n) $- \tau_{p_i} w_i\, v_i^T$ BLAS-2 _GER
Endif
End for
If (m > n),
Choose u_n of length $m-n+1$ to eliminate
A(n+1:m,n) _LARFG
End if

The routine _GEBD2 in the LAPACK library [1] (used to "clean-up", i.e., bidiagonalize the last rows and columns of a matrix for which the leading rows and columns have been done by a blocked algorithm) is an implementation of the BLAS-2 algorithm.

Suppose that the BLAS-2 algorithm is used for A too large to fit in cache and consider the data transfer entailed. For each _GEMV, A must be read from main memory to cache. For each _GER, A must be read from main memory and then written back from cache to main memory. Thus for one column-row elimination, the BLAS-2 algorithm requires four reads of A_i to RAM from cache and two writes of A_i from cache to RAM. The next section introduces some new BLAS operators that can be used to reduce data transfer in bidiagonalization. Since they combine several BLAS 2 operations, we refer to them as BLAS 2.5.

3 BLAS 2.5 Operators _GEMVER and _GEMVT

The BLAS 2.5 operator _GEMVT performs two matrix-vector multiplications. Given an $m \times n$ rectangular matrix A, an m-vector u, an n-vector z and scalars α, β, _GEMVT performs the operations

$$1. \qquad x \leftarrow \beta A^T u + z \tag{3.1}$$

$$2. \qquad w \leftarrow \alpha A x \tag{3.2}$$

The inputs are A, u, z, α, β and the outputs are x and w. When A is larger than the effective cache size, implementing _GEMVT by two calls to _GEMV requires two reads of A from RAM to cache.

Suppose that A is partitioned so that column blocks fit in cache. Then each element of A need only be read to cache once. Explicitly, partition the $m \times n$ matrix A into column blocks

$$A = [A_1|A_2|\dots|A_b] \tag{3.3}$$

where each A_i has k columns of the A-matrix. Similarly, the n vector z is partitioned into b segments, each having k components,

$$z = \begin{bmatrix} z_1 \\ \vdots \\ z_b \end{bmatrix} \quad . \tag{3.4}$$

For simplicity, suppose $kb = n$, that is, the block size k evenly divides n; otherwise the last block A_b, (last z_b-vector) has less than k columns (components). _GEMVT can be implemented by:

_GEMVT Pseudo Code

$w \leftarrow 0$
For $i = 1 : b$,
  $x_i^T \leftarrow \beta u^T A_i + z_i^T$ _GEMV
  $w \leftarrow w + \alpha A_i x_i$ _GEMV
End for

For the computation $A_i x_i$, A_i is already in cache, so elements of A are read only once and not written.

The BLAS 2.5 operator _GEMVER performs two matrix-vector multiplications and a rank-2 update (or two rank-1 updates). Given an $m \times n$ rectangular matrix A, m-vectors u_0, w_0, u, n-vectors v_0, z_0, z and scalars α, β, _GEMVER performs the operations

$$1. \quad \hat{A} \leftarrow A + u_0 z_0^T + w_0 v_0^T \quad \text{2 BLAS-2 \_GERs} \tag{3.5}$$
$$2. \quad x \leftarrow \beta \hat{A}^T u + z \quad \text{BLAS-2 \_GEMV} \tag{3.6}$$
$$3. \quad w \leftarrow \alpha \hat{A} x. \quad \text{BLAS-2 \_GEMV} \tag{3.7}$$

The inputs are A, u, z, u_0, z_0, w_0, v_0, α, β. $\hat{A}$, x, and w are output.

When A is too large to fit in cache, _GEMVER by (3.5 - 3.7) requires one read of A from RAM to cache and one write of A from cache to RAM for each of the two _GER calls, and one read of $\hat{A}$ from RAM to cache for each of the two _GEMV calls, a total data transfer of four reads of A from RAM to cache and two writes of A from cache to RAM. For A too large for cache (and small enough that a column block can stay in cache), reuse of in-cache data improves by the same blocking as in _GEMVT. Let A and $\hat{A}$ be partitioned into column blocks as in (3.3) and let the z be partitioned into b segments as in (3.4), and for simplicity suppose $kb = n$. Similarly, let the vectors z_0^T and v_0^T of length n be partitioned into b segments, each having k components,

$$z_0^T = \begin{bmatrix} z_{01}^T \\ \vdots \\ z_{0b}^T \end{bmatrix} \text{ and } v_0^T = \begin{bmatrix} v_{01}^T \\ \vdots \\ v_{0b}^T \end{bmatrix} \tag{3.8}$$

_GEMVER can be implemented as:

_GEMVER Pseudo Code

$w \leftarrow 0$
For $i = 1 : b$,
$\quad \hat{A}_i \leftarrow A_i + u_0 z_{0i}^T$ _GER
$\quad \hat{A}_i \leftarrow \hat{A}_i + w_0 v_{0i}^T$ _GER
$\quad x_i^T \leftarrow \beta u^T \hat{A}_i + z_i^T$ _GEMV
$\quad w \leftarrow w + \alpha \hat{A}_i x_i$ _GEMV
End for

For appropriately sized blocks A_i, the "cache-efficient" version of _GEMVER reads the A matrix from RAM to cache once and writes it back to RAM once.

4 Algorithm I - Efficient Reduction Without Blocking

Let A be an $m \times n$ matrix with $m \geq n$. Let $d(i)$, $1 \leq i \leq n$, and $e(i)$, $1 \leq i \leq n-1$, be the diagonal and superdiagonal elements i obtained from

bidiagonalization. Recalling (2.1) and (2.2):

$$\hat{A}_1 = (I - \tau_{q_1} u^{(1)} u^{(1)^T})\, A = \begin{pmatrix} d(1) & x & \dots & x \\ 0 & x & \dots & x \\ \vdots & & & \\ 0 & x & \dots & x \end{pmatrix} \tag{4.1}$$

and

$$H_1 A G_1 = \quad (I - \tau_{q_1} u^{(1)} u^{(1)^T}) A (I - \tau_{p_1} v^{(1)} v^{(1)^T}) \tag{4.2}$$

$$= \quad \begin{pmatrix} d(1) & e(1) & 0 & \dots & 0 \\ 0 & x & x & \dots & x \\ \vdots & \vdots & & \vdots & \\ 0 & x & \dots & \dots & x \end{pmatrix}. \tag{4.3}$$

helps with understanding the notation in the following.

Algorithm I, developed in this section, reorders the sequence of operations in Algorithm BLAS-2 of Section 2 so that the ith column-row elimination is a single call to _GEMVER instead of two matrix vector multiplications (_GEMVs) and two rank-one updates (_GERs). Packaging the _GEMV and _GER calls into one _GEMVER call reduces the traffic on the data bus from four reads and two writes per column-row elimination to one read and one write. The rearrangement takes some work. As a first (incorrect) algorithm consider the following:

First Cut Algorithm BLAS-2.5

For $i = 1$,

1. Construct u_1 of length m to zero A(2:m,1) _LARFG
2. Construct v_1 of length $n-1$ to zero A(1,3:n) _LARFG
3. $y_1^T \leftarrow u_1^T A$(1:m,2:n) BLAS-2.5
   $w_1 \leftarrow A$(2:m,2:n) v_1 _GEMVT

End For

For $i = 2 : n-2$,

4. Build u_i of length $m-i+1$ to zero A(i+1:m,i) _LARFG
5. Construct v_i of length $n-i$ to zero A(i,i+2:n) _LARFG
6. A(i:m,i:n)$\leftarrow A$(i:m,i:n)
   $-\tau_{q_{i-1}} u_{i-1}\, y_{i-1}^T - \tau_{p_{i-1}} w_{i-1}\, v_{i-1}^T$
   $y_i^T \leftarrow u_i^T A$(i:m,i+1:n) BLAS-2.5

$w_i \leftarrow A$(i+1:m,i+1:n)v_i _GEMVER

End For

For $i = n-1$,

7. Build u_i of length $m-i+1$ to zero A(i+1:m,i) _LARFG

8. A(n-1:m,n) $\leftarrow A$(n-1:m,n) BLAS-2.5

$-\tau_{q_{n-2}} u_{n-2} y_{n-2}^T - \tau_{p_{n-2}} w_{n-2} v_{n-2}^T$ ' _GEMVER

A(n-1:m,n)$\leftarrow (I - \tau_{q_{n-1}} u_{n-1} u_{n-1}^T) A$(n-1:m,n).

End For

9. If $(m > n)$,

Construct u_n to zero A(n+1:m,n) _LARFG

End If

The main loop over $i = 2:n-2$ does the same steps as Algorithm BLAS-2, but in a different order. Steps 1 and 4 (of Algorithm BLAS-2) generating the left and right Householder vectors are done first, then the updates to the trailing part of the A matrix, steps 3 and 6 (of Algorithm BLAS-2) are performed, and finally the two matrix-vector multiplications (Steps 2 and 5) are performed.

But if you look closely, there is a problem with the re-ordering of operations: for $i = 1$ the first right Householder vector v_1 isn't yet known (since the rank-1 update producing the first row in the form to be eliminated has not yet been performed). Similarly, for $i = 2:n-2$ the ith right Householder vector v_i in step 5 is not yet known since the update $(I - \tau_{q_i} u_i u_i^T) A_i$ has not been performed. Fortunately we can delay the computation of the right Householder vector; instead of using the unavailable v_1 and v_i vectors we use substitutes in the _GEMVT and _GEMVER calls, recovering the actual v_1, Av_1, v_i, and Av_i vectors in a clean-up step. Details follow.

First Column-Row Update:

Start with $i = 1$. Assume $u^{(1)} = u_1$ is normalized with $u_1(1) = 1$ and $v^{(1)} = \begin{pmatrix} 0 \\ v_1 \end{pmatrix}$ is normalized with $v_1(1) = 1$ in (4.1) and (4.2). Assume these are already the actual left and right Householder vectors, so that $H_1 A G_1$ in (4.2) is identical to (2.2). In terms of the trailing parts of the A matrix defined in (2.3), (2.8), the first update formula (4.2) takes the form

$$H_1' A_1 G_1' = (I - \tau_{q_1} u_1 u_1^T) A(1{:}m,2{:}n)(I - \tau_{p_1} v_1 v_1^T) \tag{4.4}$$

$$= \left[\begin{array}{c|ccc} e(1) & 0 & \dots & 0 \\ \hline x & & & \\ \vdots & & A_2 & \\ x & & & \end{array}\right]. \tag{4.5}$$

This is equation (2.10) with $i = 1$, where u_1 is of length m and, v_1 is of length $n-1$. Simple algebra gives (4.4) in the form of two rank-1 updates as

$$H_1' A_1 G_1' = A_1 - u_1 z_1^T - w_1 v_1^T \tag{4.6}$$

where

$$x_1 = \tau_{q_1} A_1^T u_1 \tag{4.7}$$

$$w_1 = \tau_{p_1} A_1 v_1 \tag{4.8}$$

$$z_1^T = x_1^T - \tau_{p_1}(x_1^T v_1) v_1^T \tag{4.9}$$

To pack the matrix vector multiplications (4.7) and (4.8) into a single call to _GEMVT, we would need the Householder vector v_1 to be either an input to the call, or produced during the call. Fortunately, after the first rank-one update, A(1,2:n) is, save for the first element, a scalar multiple of the Householder vector v_1. So we can use it as a "pre-Householder" vector $\tilde{v}_1$ (produced a block at a time inside _GEMVT) in computing a "pre-w_1" vector $\tilde{w}_1$ in (4.8) and then can recover v_1 and w_1 from $\tilde{v}_1$ and $\tilde{w}_1$.

Explicitly, define the pre-Householder vector $\tilde{v}_1$ by

$$\begin{aligned} \tilde{v}_1 &:= \text{1st row } \{(I - \tau_{q_1} u^{(1)} u^{(1)^T}) A(1{:}m,2{:}n)\} \\ &= A(1,2{:}n) - \tau_{q_1} u_1^T A(1{:}m,2{:}n), \end{aligned} \tag{4.10}$$

where the elements of the A-matrix are the original matrix elements. The vector $\tilde{v}_1$ is the first row of $\hat{A}_1$ (to be to be eliminated by the first right Householder matrix[6] $G_1 = I - \tau_{p_1} v_1 v_1^T$). Analogously to (4.8) take the pre-w_1 vector as

$$\tilde{w}_1(1{:}m) := A(1{:}m,2{:}n)\, \tilde{v}_1(1{:}n\text{-}1), \tag{4.11}$$

again using the original matrix elements of the A matrix. To recover v_1 and w_1, compute v_1 from the LAPACK Householder routine _LARFG as[7]

$$v_1(1{:}n\text{-}1) := \frac{\tilde{v}_1 - s e_1}{\tilde{v}_1(1) - s} \tag{4.12}$$

where

$$s := \begin{cases} -\|\tilde{v}_1\|, & \text{if } \tilde{v}_1(1) \geq 0 \\ +\|\tilde{v}_1\|, & \text{if } \tilde{v}_1(1) < 0\,. \end{cases} \tag{4.13}$$

[6] As described in pseudo-code in Section 3, the _GEMVT operator multiplies each block of $\tilde{v}_1$ to perform (4.11) with a column block A_j of A still in cache; if we computed all of $\tilde{v}_1$ to have the actual Householder vector v_1, then we would read all the rest of A, flushing the current column block A_j.

[7] The LAPACK routine _LARFG computes the Householder vector by algorithm 5.1.1. in the second edition of Golub and Van Loan [18].

We recover $A\,v_1$ from $A\,\tilde{v}_1$ by using linearity:

$$\begin{aligned} A(1{:}\mathrm{m},2{:}\mathrm{n})\,v_1 & \\ &= \left(\tfrac{1}{\tilde{v}_1(1)-s}\right) A(1{:}\mathrm{m},2{:}\mathrm{n})\,\tilde{v}_1 - \left(\tfrac{s}{\tilde{v}_1(1)-s}\right) A(1{:}\mathrm{m},2{:}\mathrm{n})\,e_1 \\ &= \left(\tfrac{1}{\tilde{v}_1(1)-s}\right) \{\tilde{w}_1(1{:}\mathrm{m}) - sA(1{:}\mathrm{m},2)\}\,. \end{aligned} \tag{4.14}$$

Formulas (4.11) and (4.14) make use of the original matrix elements in columns 2 to n. The missing quantity for the rank-2 update in (4.6) is τ_{p_1}, the scalar factor for the first right Householder vector, needed for w_1 and z_1^T in (4.8)-(4.9). τ_{p_1} is returned by _LARFG but can also be computed as

$$\tau_{p_1} := \frac{s - \tilde{v}_1(1)}{s}\,. \tag{4.15}$$

Now that we see how to recover v_1 and Av_1, return to the _GEMVT call which outputs $\tilde{v}_1$ and $\tilde{w}_1$. After calling _LARG to get u_1 and τ_{q_1}, the matrix-vector multiplications in (4.10) and (4.11) can be made by a single call to _GEMVT:

$$\tilde{v}_1^T(1{:}\mathrm{n}\text{-}1) \leftarrow -\tau_{q_1} A^T(1{:}\mathrm{m},2{:}\mathrm{n})\,u_1 + A^T(1,2{:}\mathrm{n}) \tag{4.16}$$

$$\tilde{w}_1(1:m) \leftarrow A(1{:}\mathrm{m},2{:}\mathrm{n})\,\tilde{v}_1^T, \tag{4.17}$$

where the A matrix in both cases consists of the original matrix elements in columns 2 to n. Then we use the _GEMVT outputs $\tilde{v}_1$ and $\tilde{w}_1$ to get τ_{p_1}, v_1 and w_1 from equations (4.15), (4.12), and (4.14), (4.7) respectively. The vector $x_1 = \tau_{q_1} A^T u_1$ in (4.9) is available from the output (4.16) of _GEMVT as

$$x_1 = \tau_{q_1} A^T(1{:}\mathrm{m},2{:}\mathrm{n})\,u_1 = -\tilde{v}_1^T + A^T(1,2{:}\mathrm{n})\,, \tag{4.18}$$

The vector z_1^T is obtained from x_1 in (4.9). Now the four vectors u_1, z_1, w_1 and v_1 are known. The rank-2 update (4.6) could now be made. However, to avoid a superfluous read of A from RAM, we defer the update to be part of a _GEMVER call in the $i=2$ step.

Algorithm I: Pseudo Code for $i = 1$

1st Column-Row Update

1. Build an m-vector u_1, to zero $A(2{:}\mathrm{m},1)$ _LARFG
   and compute scalars τ_{q_1} and $d(1)$ such that
   $d(1)\;e_1 = (I_m - \tau_{q_1} u_1 u_1^T)\,A(1{:}\mathrm{m},1)$.
   Then store

$A(1,1) \leftarrow d(1)$ and $A(2{:}m,1) \leftarrow u_1(2{:}m)$.

2. (i) Using Eqns (4.16) and (4.17), compute — BLAS-2.5
   $\tilde{v}_1^T(1{:}n\text{-}1) \leftarrow -\tau_{q_1} A^T(1{:}m,2{:}n)\, u_1 + A^T(1,2{:}n)$ — _GEMVT
   $\tilde{w}_1 \leftarrow A(1{:}m,2{:}n)\, \tilde{v}_1^T$
   (ii) $x_1 \leftarrow -\tilde{v}_1^T + A^T(1,2{:}n)$ Eqn (4.18)) — _AXPY
3. Compute v_1 of length $n-1$ to zero $A(1,3{:}n)$ — _LARFG
   and the scalars τ_{p_1} and superdiagonal $e(1)$.
   $A(1,3{:}n) \leftarrow v_1(2{:}n\text{-}1)$, $A(1,2) \leftarrow e(1)$.
4. $t \leftarrow \tilde{v}_1(1) - s$ where — _AXPY
   $$s \leftarrow \begin{cases} -\|\tilde{v}_1\|, & \text{if } \tilde{v}_1(1) \geq 0 \\ +\|\tilde{v}_1\| & \text{if } \tilde{v}_1(1) < 0 \end{cases}$$
   and compute (Eqns (4.7) and (4.14)), — _AXPY, _SCAL
   $$w_1 \leftarrow \tau_{p_1} \left(\frac{\tilde{w}_1 - sA(1{:}m,2)}{t} \right).$$
5. $z_1 \leftarrow x_1 + \tau_{p_1}(x_1^T v_1)\, v_1$ (Eqn (4.8)). — _DOT, _AXPY
6. Store w_1 and z_1 for the first rank-2 update (Eqn (4.6)):
   $w_0 \leftarrow w_1, \;\; z_0 \leftarrow z_1$

In the next section, we will use the notation $u_0 = u_1, \quad v_0 = v_1$. Actually, u_1 and v_1 are stored in the first column and row of A, respectively.

ith Column-Row Update, $2 \leq i \leq n-2$:

To proceed with the second (and, in general, the ith) column-row update, use the rank-2 update analogous to the $i = 1$ case of (4.6) to compute the updated matrix in columns i to n and rows i to m . For $i = 2$, compute u_2, τ_{q_2}, $d(2)$, $\tilde{v}_2$, $\tilde{w}_2$, x_2, v_2, τ_{p_2}, $e(2)$, w_2, z_2 in that order. Instead of using _GEMVT in Step 2, make use of _GEMVER so that the trailing part of the matrix A_2 (4.4) is updated from the first column-row elimination.

Explicitly, suppose the updated matrix $A^{(i)}$ in (2.3) is available so that we could compute

$$\hat{A}^{(i+1)} := (I - \tau_{q_i} u^{(i)} u^{(i)^T}) A^{(i)} \tag{4.19}$$

and as in (2.8)

$$A^{(i+1)} = H_i A^{(i)} G_i = (I - \tau_{q_i} u^{(i)} u^{(i)^T}) A^{(i)} (I - \tau_{p_i} v^{(i)} v^{(i)^T})$$

$$= \left(\begin{array}{c|c|c} & \text{col } i & \\ B_i & & 0 \\ & e(i) & \\ \hline & x & \\ 0 & \vdots & A_{i+1} \\ & x & \end{array} \right). \tag{4.20}$$

As for $i = 1$, express (4.20) by generating the four vectors $u^{(i)}$, $z^{(i)}$, $w^{(i)}$, and $v^{(i)}$, so that the the column-row update in (4.20) becomes the two rank-1 updates

$$H_i A^{(i)} G_i = A^{(i)} - u^{(i)} z^{(i)^T} - w^{(i)} v^{(i)^T}. \tag{4.21}$$

In terms of the trailing part of the A matrix, $A_i = A$(i:m,i+1:n) and $A_{i+1} = A$(i+1:m,i+2:n) equations (4.19) and (4.20) are

$$\hat{A}_{i+1} := (I - \tau_{q_i} u_i u_i^T) A_i \tag{4.22}$$

and

$$H_i' A_i G_i' = (I - \tau_{q_i} u_i u_i^T) A_i (I - \tau_{p_i} v_i v_i^T)$$

$$= \left(\begin{array}{c|ccc} e(i) & 0 & \cdots & 0 \\ \hline x & & & \\ \vdots & & A_{i+1} & \\ x & & & \end{array} \right) \tag{4.23}$$

Equation (4.23) corresponds to equation (2.10). The vectors u_i, v_i are defined as in (2.7) with the normalization $u_i(1) = v_i(1) = 1$.

Rewrite (4.23) as two rank-1 updates

$$H_i' A_i G_i' = A_i - u_i z_i^T - w_i v_i^T \tag{4.24}$$

where

$$x_i = \tau_{q_i} A_i^T u_i \tag{4.25}$$

$$w_i = \tau_{p_i} A_i v_i \tag{4.26}$$

$$z_i^T = x_i^T - \tau_{p_i} (x_i^T v_i) v_i^T \tag{4.27}$$

As for $i = 1$, defer the update in (4.22). Compute vectors u_i, z_i, w_i, v_i so that the update for both ith column and ith row in (4.23), (4.24) can be done at once.[8] Then the $i-1$st column-row update and the two matrix-vector multiplications associated with the ith column-row update are made by a single call to _GEMVER.

Explicitly, as in (4.23), the matrix $A_i = A(\text{i:m,i+1:n})$ (to be updated) is the trailing part of the A-matrix after the updates from the first $i-1$ left and right Householder vectors have been performed. In order to package the updates to A_i from the $i-1$ left and right Householder vectors together with the two matrix-vector multiplications in (4.26) and (4.25) in a single call to _GEMVER, defer the update which produces A_i from A_{i-1} using u_{i-1}, z_{i-1}, w_{i-1}, v_{i-1} until after the ith left Householder vector is computed. Following the same order of computation as in the $i=1$ case, do the $i-1$st update only for the ith column of A (1st column of A_i). Then make a call to _LARFG to compute the ith left Householder vector u_i, its scalar factor τ_{q_i}, and $d(i)$. Define the ith right pre-Householder vector $\tilde{v}_i$ by

$$\begin{aligned}\tilde{v}_i &:= \text{1st row } \{(I - \tau_{q_i} u_i u_i^T) A(\text{i:m,i+1:n})\} \\ &= A(\text{i,i+1:n}) - \tau_{q_i} u_i^T A(\text{i:m,i+1:n}),\end{aligned} \tag{4.28}$$

where the elements of the A matrix are the elements of A_i which have been updated through the $i-1$st left and right Householder matrices H'_{i-1} and G'_{i-1}. Thus $\tilde{v}_i$ is the first row of $\hat{A}_{i+1}$ in (4.22) which is to be eliminated by the ith right Householder matrix $G'_i = I = \tau_{p_i} v_i v_i^T$. As in the _GEMVER pseudo-code $\tilde{v}_i$ is obtained a block at a time and immediately used to increment the partial sum of the pre-w_i vector defined by

$$\tilde{w}_i(\text{1:m-i+1}) := A(\text{i:m,i+1:n})\tilde{v}_i(\text{1:n-i}) \tag{4.29}$$

where the elements of the A matrix are the elements of the A_i matrix.

Having computed $\tilde{v}_i$ and $\tilde{w}_i$ from a _GEMVER call, the right Householder vector v_i can be obtained from a call to _LARFG and is related to $\tilde{v}_i$ by

$$v_i(\text{1:n-1}) := \frac{\tilde{v}_i - s e_1}{\tilde{v}_i(1) - s} \tag{4.30}$$

where

$$s := \begin{cases} -\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) \geq 0 \\ +\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) < 0. \end{cases} \tag{4.31}$$

[8] Defer (4.24) till step $i+1$ so that the trailing part of A is accessed only once on step i.

The scalar factor τ_{p_i}

$$\tau_{p_i} := \frac{s - \tilde{v}_i(1)}{s}. \tag{4.32}$$

is computed by a call to _LARFG. To recover Av_i from $\tilde{w}_i := A\tilde{v}_i$, use linearity:

$$\begin{aligned} \text{A(i:m,i+1:n)}v_i &= \left(\frac{1}{\tilde{v}_i(1) - s}\right) A\text{(i:m,i+1:n)}\tilde{v}_i \\ &\quad - \left(\frac{s}{\tilde{v}_i(1) - s}\right) A\text{(i:m,i+1:n)}e_1 \\ &= \frac{\tilde{w}_i\text{(1:m-i+1)} - sA\text{(i:m,i+1)}}{\tilde{v}_i(1) - s)}. \end{aligned} \tag{4.33}$$

Summary: The $i-1$st column-row update to produce A_i can be deferred until we are ready to perform the two matrix-vector multiplications in (4.28) and (4.29) to compute $\tilde{v}_i$ and $\tilde{w}_i$. Accordingly, we can generate the ith left Householder vector u_i and its scalar factor τ_{q_i} before the $i-1$st column-row update, and then package the two matrix-vector multiplications in (4.28) and (4.29) along with the $i-1$st column-row updates for A_i in a single call to _GEMVER:

$$\begin{aligned} A\text{(i:m,i+1:n)} \leftarrow A\text{(i:m,i+1:n)} &\ - u_{i-1}\text{(2:m-i+2)}\, z_{i-1}^T\text{(2:n-i+1)} \\ &\ - w_{i-1}\text{(2:m-i+2)}\, v_{i-1}^T\text{(2:n-i+1)} \qquad (4.34) \\ \tilde{v}_i^T\text{(1:n-i)} \leftarrow -\tau_{q_i} A^T\text{(i:m,i+1:n)}\, u_i &+ A^T\text{(i,i+1:n)} \qquad (4.35) \\ \tilde{w}_i\text{(1:m-i+1)} \leftarrow A\text{(i:m,i+1:n)}\, \tilde{v}_i^T. & \qquad (4.36) \end{aligned}$$

Following the _GEMVER call, recover v_i, τ_{p_i}, and w_i from the outputs $\tilde{v}_i$ and $\tilde{w}_i$ using equations (4.30), (4.32), and (4.33) respectively. The quantity $x_i = \tau_{q_i}\ A^T u_i$ in (4.25) is available from the output (4.35) of _GEMVER as

$$x_i = \tau_{q_i} A^T\text{(i:m,i+1:n)}\, u_i = -\tilde{v}_i^T + A^T\text{(i,i+1:n)}. \tag{4.37}$$

The vector z_i^T is then computed using x_i in (4.27). Now the four vectors u_i, z_i, w_i, and v_i have been computed. As before, instead of immediately performing the rank-2 update in (4.23) and (4.24), delay the update until the _GEMVER call generating $\tilde{v}_{i+1}$ and $\tilde{w}_{i+1}$.

Algorithm I: Pseudo Code for $i = 2 : n-2$

For $i = 2 : n-2$,

1 (i) Update the ith column of A with the vectors u_0,z_0,
w_0,v_0 (Eqn (4.24) with $i-1$ in place of i:)
A(i:m,i) $\leftarrow A$(i:m,i) $- u_0$(1:m-i+1) $z_0(1)$
$-w_0$(1:m-i+1) $v_0(1)$ 2 _AXPYs

(ii) Get the current ith row of A with the vectors
u_0, z_0, w_0, v_0 (Eqn (4.24) for $i-1$)
Does not overwrite A, but is needed as
an input to _GEMVER)
$z_{input} \leftarrow A$(i,i+1:n) $- u_0(2) z_0$(1:n-i)
$-w_0(2) v_0$(1:n-i). 2 _AXPYs

(iii) Compute u_i of length $m-i+1$ to zero
A(i+1:m,i). Also compute scalars τ_{q_i} and $d(i)$. _LARFG
A(i+1:m,i) $\leftarrow u_i$(2:m-i+1), A(i,i)$\leftarrow d$(i).

2. (i) _GEMVER call (Eqns (4.34)-(4.36))
A(i:m,i+1:n) $\leftarrow A$(i:m,i+1:n)
$- u_0$(2:m-i+2) z_0^T(2:n-1+1)
$- w_0$(2:m-i+2) v_0^T(2:n-1+1)
$\tilde{v}_i \leftarrow z_{input} - \tau_{q_i} (u_i^T A$(i:m,i+1:n)) BLAS-2.5
$\tilde{w}_i \leftarrow A$(i:m,i+1:n) $\tilde{v}_i^T$, _GEMVER
(ii) $x_i \leftarrow -\tilde{v}_i^T + z$ (Eqn (4.37)). _AXPY

3 Compute v_i of length $n-i$ to zero $\tilde{v}_i$(2:n-i),
Also compute the scalars τ_{p_i} and $e(i)$. _LARFG
A(i,i+2:m)$\leftarrow v_i$(2:n-i), e(i)$\leftarrow A$(i,i+1)

4 Put $t \leftarrow \tilde{v}_i(1) - s$ where

$$s := \begin{cases} -\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) \geq 0 \\ +\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) < 0 \end{cases}$$

and compute (Eqns (4.33) and (4.26)),

$$w_i \leftarrow \tau_{p_i} \left(\frac{\tilde{w}_i - sA(\text{i:m,i+1})}{t} \right). \quad \text{\_AXPY}$$

5 $z_i \leftarrow x_i - \tau_{p_i}(x_i^T v_i) v_i$ (Eqn (4.27)) _DOT, _AXPY

6 Store w_i and z_i for the ith rank-2 update
(Eqn (4.24)): $w_0 \leftarrow w_i$, $z_0 \leftarrow z_i$
u_i, v_i are already stored in the ith column and row
of A respectively. For convenience of notation consider
$u_0 \leftarrow u_1$, $v_0 \leftarrow v_i$).

End For.

When $i = n-1$, we only need to compute the left Householder vector and multiply the A matrix on the left by the corresponding Householder matrix. There is no right Householder vector, so the _GEMVER call and Steps 2-6 are not needed. After the $i = n-2$ steps are completed, the four vectors u_0, z_0, w_0, v_0 contain u_{n-2}, z_{n-2}, w_{n-2}, v_{n-2}, so we need only use them to update the trailing $(m-n+2) \times 2$ matrix, A(n-1:m,n-1:n), call _LARFG to compute the $n-1$st left Householder vector, and then apply the left Householder matrix to update the last column of A.

Algorithm I: Pseudo Code for $i = n-1$

1 (i) Update the $n-1$st and n th columns of A with
u_{n-2}, z_{n-2}, w_{n-2} and v_{n-2}
(Eqn (4.24) with $n-2$ in place of i):
A(n-1:m,n-1) $\leftarrow A$(n-1:m,n-1)
$-u_0$(1:m-n+2)$z_0(1)$
$-w_0$(1:m-n+2)$v_0(1)$, 2 _AXPYs
A(n-1:m,n) = A(n-1:m,n) $- u_0$(1:m-n+2)$z_0(2)$
$-w_0$(1:m-n+2)$v_0(2)$. 2 _AXPYs
(ii) Compute u_{n-1} of length $m-n+2$ to
zero A(n:m,n-1) and compute the _LARFG
scalars $\tau_{q_{n-1}}$, and d(n-1).

2 A(n-1:m,n) $\leftarrow (I - \tau_{q_{n-1}} u_{n-1} u_{n-1}^T) A$(n-1:m,n),
e(n-1) $\leftarrow A$(n-1,n), A(n,n)$\leftarrow d(1)$,
and A(n:m,n-1)$\leftarrow u_{n-1}$.

If $m > n$, we need one last left Householder vector to zero out the entries A(n+1:m,n).

Algorithm I: Pseudo Code for $i = n$

If $m = n$,
Set d(n)$\leftarrow A$(n,n) and $\tau_{q_n} \leftarrow 0$,
Else
If $m > n$
Compute u_n of length $m-n+1$ to zero
A(n+1:m,n); compute τ_{q_n} and d(n). _LARFG
A(n+1:m,n)$\leftarrow u_n$(2:m-n+1), A(n,n)$\leftarrow d$(n)
Endif ($m > n$)
Endif ($m = n$)

As in Algorithm BLAS-2 of Section 2, the trailing part of the A-matrix is updated after each paired column-row elimination. The next algorithm does not update rows and columns until they are due to be eliminated.

5 Algorithm II - A-Matrix Never Updated

As the first operation (4.34) of a _GEMVER call, Algorithm I updates the trailing part of the matrix on each column-row elimination. Matrix updates on each column-row elimination have several drawbacks. Writes of the matrix from cache to RAM are slow, typically somewhat slower than reads from RAM to cache. Moreover, updates may not preserve structure, such as sparsity, of the original matrix.

Algorithm II defers matrix updates. The trailing matrix is accessed (read) only to perform matrix-vector multiplications. The set of update vectors is incremented as the elimination proceeds. BLAS-2 matrix-vector multiplies update the ith column and row from the original matrix on the step before they are eliminated.

First, we give some notation. Let u_i, z_i, w_i, and v_i be the vectors for the rank-2 update of (4.24) in Algorithm I:

$$\begin{aligned} H_i' A_i G_i' &= A_i u_i z_i^T - w_i v_i^T \\ &= \left(\begin{array}{c|ccc} e(i) & 0 & \dots & 0 \\ \hline x & & & \\ \vdots & & A_{i+1} & \\ x & & & \end{array} \right). \end{aligned} \tag{5.1}$$

Here u_i is the left Householder vector of length $m-i+1$ with $u_i(1) = 1$ and v_i is the right Householder vector of length $n-i$ with $v_i(1)= 1$, which arise from the two _LARFG calls in Steps 1 and 3 of Algorithm I, and z_i, w_i are the vectors of length $n-i$ and $m-i+1$ defined by (4.27) and (4.26), respectively, in Algorithm I. The corresponding full length vectors are

$$u^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ u_i \end{pmatrix} \text{ and } w^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ w_i \end{pmatrix} \text{ with } i-1 \text{ initial zeros}, \tag{5.2}$$

both of length m, and

$$z^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ z_i \end{pmatrix} \text{ and } v^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ v_i \end{pmatrix} \text{ with } i \text{ initial} \quad \text{zeros}, \tag{5.3}$$

both of length n.

Suppose the first $i-1$ of the above vectors have been computed. If A^{orig} denotes the original matrix, the matrix $A^{(i)}$ of (2.3) (updated by the first $i-1$ left and right Householder matrices) is

$$\begin{aligned} A^{(i)}(1{:}\mathrm{m},1{:}\mathrm{n}) &= \prod_{j=1}^{i-1}[I - \tau_{q_j} u^{(j)}\, u^{(j)^T}]\, A^{orig}(1{:}\mathrm{m},1{:}\mathrm{n}) \prod_{j=1}^{i-1}[I - \tau_{p_j} v^{(j)}\, v^{(j)^T}] \\ &= A^{orig}(1{:}\mathrm{m},1{:}\mathrm{n}) - \sum_{j=1}^{i-1} u^{(j)} z^{(j)^T} - \sum_{j=1}^{i-1} w^{(j)} v^{(j)^T} \\ &= A^{orig}(1{:}\mathrm{m},1{:}\mathrm{n}) - (u^{(1)}, u^{(2)}, \ldots, u^{(i-1)}) \begin{pmatrix} z^{(1)^T} \\ \vdots \\ z^{(i-1)^T} \end{pmatrix} \\ &\quad - (w^{(1)}, w^{(2)}, \ldots, w^{(i-1)}) \begin{pmatrix} v^{(1)^T} \\ \vdots \\ v^{(i-1)^T} \end{pmatrix}. \end{aligned} \tag{5.4}$$

The last representation follows from repeated application of the rank-2 update formula (4.21). Take the full length vectors $u^{(j)}$, $w^{(j)}$, $z^{(j)}$ and $v^{(j)}$ as in (5.2)-(5.3). Define the matrices

$$U_{mat} = [u^{(1)}, u^{(2)}, \ldots, u^{(n)}], \;\; W_{mat} = [w^{(1)}, w^{(2)}, \ldots, w^{(n)}]$$

$$(m \times n) \text{ (lower trapezoidal)} \tag{5.5}$$

and

$$Z_{mat} = \begin{bmatrix} z^{(1)^T} \\ z^{(2)^T} \\ \vdots \\ z^{(n-2)^T} \end{bmatrix}, \; V_{mat} = \begin{bmatrix} v^{(1)^T} \\ v^{(2)^T} \\ \vdots \\ v^{(n-2)^T} \end{bmatrix} (n-2 \times n)$$

(strictly upper trapezoidal)

The first $i-1$ rank-2 updates can be written as

$$A^{(i)}(\text{1:m,1:n}) = A^{orig}(\text{1:m,1:n}) \quad - U_{mat}(\text{1:m,1:i-1})Z_{mat}(\text{1:i-1,1:n}) \quad - W_{mat}(\text{1:m,1:i-1})V_{mat}(\text{1:i-1,1:n}). \tag{5.6}$$

Similarly, any portion $A^{(i)}$(k:p,1:q), of the $m \times n$ matrix $A^{(i)}$ (updated by $i-1$ rank-2 updates) may be expressed as

$$A^{(i)}(\text{k:p,1:q}) = A^{orig}(\text{k:p,1:q}) \quad - U_{mat}(\text{k:p,1:i-1})Z_{mat}(\text{1:i-1,1:q}) \quad - W_{mat}(\text{k:l,1:i-1})V_{mat}(\text{1:i-1,l:q}). \tag{5.7}$$

In particular, $A_i = A^{(i)}$(i:m,i+1:n) is

$$\begin{aligned} A_i = \quad & A^{(i)}(\text{i:m,i+1:n}) \\ = \quad & A^{orig}(\text{i:m,i+1:n}) \quad - U_{mat}(\text{i:m,1:i-1})Z_{mat}(\text{1:i-1,i+1:n}) \\ & \qquad\qquad\quad - W_{mat}(\text{i:m,1:i-1})V_{mat}(\text{1:i-1,i+1:n}) \end{aligned} \tag{5.8}$$

Algorithm II consists of the same basic steps as Algorithm I. The main difference is that rows and columns are not updated until just before they are eliminated. Thus the two matrix-vector multiplications accomplished by calls to _GEMVT ($i = 1$) and _GEMVER ($i = 2$ to $n-2$) are done using elements of the original A-matrix.[9] Whenever the updated ith column or row is required, or whenever any submatrix of the $A^{(i)}$(1:m,1:n) matrix is required, formulas of the type (5.7), (5.8) are substituted for the updated quantities and the ensuing calculations performed using the original elements of the A-matrix, along with the necessary correction terms which arise from the vectors

$$u_j, z_j, w_j, v_j, j = 1 \ldots i-1$$

stored (as in (5.2-5.5)) in the arrays U_{mat}, Z_{mat}, W_{mat} and V_{mat}.

In Algorithm II, the original matrix A is overwritten only if we choose to overwrite eliminated rows and columns by the Householder vectors used to eliminate them. Matrix vector multiplications are performed with A^{orig} and with the arrays U_{mat}, Z_{mat}, W_{mat} and V_{mat}. In the sparse case, eliminating updates of A^{orig} eliminates matrix fill and allows matrix vector multiplications to be made with the original sparse matrix.[10].

[9] Because the trailing part of the matrix is not updated, _GEMVT calls are used in place of _GEMVER calls.

[10] In the dense case, the U_{mat} and V_{mat} arrays are unnecessary since there is enough room below the diagonal of A to store the u_i vectors and enough room to the right of the superdiagonal of A to store the v_i vectors.

To see how bidiagonalization can proceed without updating the trailing part of the A-matrix, observe that when $i=1$, Steps 1-5 are exactly the same as in Algorithm I. In Step 6, the w_1 and z_1 vectors are stored in the first column and row of W_{mat} and Z_{mat} instead of in w_0 and z_0. If the original matrix is not sparse, the vectors u_i, v_i can be stored in the ith column and row of A.

_GEMVT when the trailing part of the matrix is not updated: The _GEMVER update of the trailing part of the A-matrix (first part of Step 2) is not done. The other portion of Step 2 is a _GEMVT set of two matrix multiplications. Step 2 still performs the _GEMVT operations, complicated by the need to also perform matrix vector multiplications by the update vectors.

Reconsider the discussion of the vectors $\tilde{v}_i$ and $\tilde{w}_i$ used in the _GEMVT computation. By substituting the matrix $A^{(i)}$(i:m,i+1:n) from (5.8) for the updated A(i:m,i+1:n) in equation (4.28), the two matrix-vector multiplications in (4.35) and (4.36) can be performed using the matrix A^{orig}(i+1:m, i+1:n) in a call to _GEMVT. More explicitly, assume that in Algorithm I we have updated the ith column and the ith row of the A-matrix using all the left and right Householder vectors up through $i-1$, and that the trailing part, A(i+1:m,i+1:n), has been updated by the first $i-1$ rank-2 updates. Starting from equation (4.28),

$$\begin{aligned}
\tilde{v}_i(1{:}n\text{-}i) &= A^{(i)}(i,i{+}1{:}n) - \tau_{q_i} u_i^T A^{(i)}(i{:}m,i{+}1{:}n) \\
&= A^{(i)}(i,i{+}1{:}n) - \tau_{q_i} u_i(1) A^{(i)}(i,i{+}1{:}n) \\
&\qquad\qquad -\tau_{q_i} u_i^T(2{:}m\text{-}i{+}1) A^{(i)}(i{+}1{:}m,i{+}1{:}n) \\
&= (1-\tau_{q_i}) A^{(i)}(i,i{+}1{:}n) \\
&\qquad\qquad -\tau_{q_i} u_i^T(2{:}m\text{-}i{+}1)[A^{orig}(i{+}1{:}m,i{+}1{:}n) - B_i] \quad (5.9)
\end{aligned}$$

where, using (5.7) with $k=l=i+1$, $p=m$, and $q=n$,

$$\begin{aligned}
B_i := U_{mat}(i{+}1{:}m,1{:}i\text{-}1) Z_{mat}(1{:}i\text{-}1,i{+}1{:}n) \\
+ W_{mat}(i{+}1{:}m,1{:}i\text{-}1) V_{mat}(1{:}i\text{-}1,i{+}1{:}n). \quad (5.10)
\end{aligned}$$

B_i is not explicitly computed[11]. Instead, compute

$$\begin{aligned}
u_i^T B_i = (u_i^T U_{mat}(i{+}1{:}m,1{:}i\text{-}1)) Z_{mat}(1{:}i\text{-}1,i{+}1{:}n) \\
+ (u_i^T W_{mat}(i{+}1{:}m,1{:}i\text{-}1)) V_{mat}(1{:}i\text{-}1,i{+}1{:}n). \quad (5.11)
\end{aligned}$$

[11] Computing B_i would require extra storage and increase the number of floating point operations.

Similarly, for $\tilde{w}_i$, we have, starting with equation (4.29),

$$\begin{aligned}\tilde{w}_i(1{:}\text{m-i+1}) &= A^{(i)}(\text{i:m,i+1:n})\tilde{v}_i^T \\ &= [A^{(i)}(\text{i,i+1:n})\tilde{v}_i^T, A(\text{i+1:m,i+1:n})\tilde{v}_i^T] \\ &= [A^{(i)}(\text{i,i+1:n})\tilde{v}_i^T, A^{orig}(\text{i+1:m,i+1:n})\tilde{v}_i^T - B_i\tilde{v}_i^T] \qquad (5.12)\end{aligned}$$

where the B_i matrix is the same as in (5.10) and $B_i\tilde{v}_i^T$ is performed analogously to (5.11). Using

$$z_{input}(1{:}\text{n-i}) := (1-\tau_{q_i})A^{(i)}(\text{i,i+1:n}) + \tau_{q_i}u_i^T(2{:}\text{m-i+1})B_i \qquad (5.13)$$

in the _GEMVT call, _GEMVT does the following two matrix-vector multiplications.

$$\begin{aligned}\tilde{v}_i^T(1{:}\text{n-i}) &:= -\tau_{q_i}u_i^T(2{:}\text{m-i+1})\,A^{orig}(\text{i+1:m,i+1:n}) + z_{input}^T \qquad (5.14)\\ \tilde{w}_i(2{:}\text{m-i+1}) &:= A^{orig}(\text{i+1:m,i+1:n})\,\tilde{v}_i^T. \qquad (5.15)\end{aligned}$$

From (5.12), complete the computation of $\tilde{w}_i$ (as in (5.11) do not actually form B_i) by the update

$$\tilde{w}_i(2{:}\text{m-i+1}) = \tilde{w}_i(2{:}\text{m-i+1}) - B_i\tilde{v}_i^T. \qquad (5.16)$$

Adjoin the first component by computing the dot product

$$\tilde{w}_i(1) = A^{(i)}(\text{i,i+1:n})\tilde{v}_i^T. \qquad (5.17)$$

From the first equation in (5.9), x_i (equation (4.25)) is computable as

$$x_i(1{:}\text{n-i}) = -\tilde{v}_i^T(1{:}\text{n-i}) + A^{(i)^T}(\text{i,i+1:n}) \qquad (5.18)$$

where $A^{(i)}$(i,i+1:n) is the updated ith row of A. This is the same as equation (4.37) obtained for Algorithm I.

Performing the deferred updates:
Having deferred updates of the trailing part of the matrix, the ith step still requires updated rows and columns to compute the corresponding Householder vectors. The original ith row is updated just before the call to _GEMVT by using all the left and right Householder vectors up to $i-1$ (equation (5.7) with $k=p=i$, $l=i+1$, and $q=n$)

$$\begin{aligned}A(\text{i,i+1:n}) \leftarrow A^{orig}(\text{i,i+1:n}) &+ U_{mat}(\text{i,1:i-1})Z_{mat}(\text{1:i-1,i+1:n}) \\ &- W_{mat}(\text{i,1:i-1})V_{mat}(\text{1:i-1,i+1:n}). \qquad (5.19)\end{aligned}$$

Also, obtaining w_i from $\tilde{w}_i$ requires an updated ith column. Specifically, the ith column is required on column row elimination $i-1$. Since the formula (4.33) needed for recovering w_i from $\tilde{w}_i$ requires the updated $i+1$st column of A and since the only element of this column which has been updated is A(i,i+1) (done in Step 2(i) in the pseudo-code below) we must first update A(i+1:m,i+1) using equation (5.7) with $k = i+1, p = m$, and $l = i+1$,

$$A^{(i)}(\text{i+1:m,i+1}) \leftarrow A^{orig}(\text{i+1:m,i+1}) - U_{mat}(\text{i+1:m,1:i-1})Z_{mat}(\text{1:i-1,i+1}) - W_{mat}(\text{i+1:m,1:i-1})V_{mat}(\text{1:i-1,i+1}). \quad (5.20)$$

On the ith column row elimination, the ith column requires the same _AXPYs performed in Algorithm I, now phrased in terms of the update arrays.

$$A(\text{i:m,i}) \leftarrow A(\text{i:m,i}) - U_{mat}(\text{i:m,i-1})\, Z_{mat}(\text{i-1,i}) - W_{mat}(\text{i:m,i-1})\, V_{mat}(\text{i-1,i}).$$

Algorithm II: Pseudo Code

For $i = 3 : n-3$,

1. (i) Complete the update of the ith column of A
   $A(\text{i:m,i}) \leftarrow A(\text{i:m,i}) - U_{mat}(\text{i:m,i-1})\, Z_{mat}(\text{i-1,i})$
   $-W_{mat}(\text{i:m,i-1})\, V_{mat}(\text{i-1,i})$. 2 _AXPYs
   (ii) Compute u_i of length $m-i+1$ to zero A(i+1:m,i), and compute the scalars τ_{q_i} and d(i). _LARFG
   $A(\text{i+1:m,i}) \leftarrow u_i(\text{2:m-i+1}), \quad A(\text{i,i}) \leftarrow d(\text{i})$.
   (iii) Update the original ith row of A using all the left and right Householder vectors up to $i-1$ (Eqn. (5.19))
   $A(\text{i,i+1:n}) \leftarrow A^{orig}(\text{i,i+1:n})$
   $-U_{mat}(\text{i,1:i-1})Z_{mat}(\text{1:i-1,i+1:n})$
   $-W_{mat}(\text{i,1:i-1})V_{mat}(\text{1:i-1,i+1:n})$. 2 _GEMVs

2. (i) This step corresponds to a _GEMVT using the updated matrix. Generate $\tilde{v}_i$ and $\tilde{w}_i$.
   (A) Use the updated ith row of A from 2(i), the ith left Householder vector u_i, the scalar τ_{q_i} from 1(ii), and the vectors u_j, z_j, w_j, v_j stored for $j = 1, 2, \ldots, i-1$ in U_{mat}, Z_{mat}, W_{mat}, and V_{mat} : to compute as in (5.11),
   $z_{input} \leftarrow (1 - \tau_{q_i})A(\text{i,i+1:n})$ 4 _GEMVs
   $+ \tau_{q_i}[u_i^T(\text{2:m-i+1})U_{mat}(\text{i+1:m,1:i-1})]\, Z_{mat}(\text{1:i-1,i+1:n})$
   $+ \tau_{q_i}[u_i^T(\text{2:m-i+1})W_{mat}(\text{i+1:m,1:i-1})]\, V_{mat}(\text{1:i-1,i+1:n})$.
   (B) Compute (Eqns (5.14), (5.15)),

$\tilde{v}_i$(1:n-i) $\leftarrow z_{input}$(1:n-i) $- \tau_{q_i} u_i^T\, A^{orig}$(i:m,i+1:n)
and BLAS 2.5
$\tilde{w}_i$(2:m-i+1) $\leftarrow A^{orig}$(i+1:m,i+1:n) $\tilde{v}_i^T$. _GEMVT
The vector $\tilde{v}_i$ now contains the same entries
as in 2(ii) of Algorithm I.

(C) Complete the computation of $\tilde{w}_i$(1:m-i+1) (Eqns (5.16) and (5.17)).
$\tilde{w}_i$(2:m-i+1) $\leftarrow \tilde{w}_i$(2:m-i+1) 4 _GEMVs
$- \tau_{q_i} U_{mat}$(i+1:m,1:i-1) [Z_{mat}(1:i-1,i+1:n) $\tilde{v}_i^T$]
$- \tau_{q_i} W_{mat}$(i+1:m,1:i-1) [V_{mat}(1:i-1,i+1:n) $\tilde{v}_i^T$].
$\tilde{w}_i$ now contains the same entries
as in Step 2(ii) of Algorithm I.

(ii) $x_i \leftarrow -\tilde{v}_i^T + A^{(i)^T}$(i,i+1:n) (Eqn.(5.18)) _AXPY

3. Construct v_i of length $n-i$ to zero $\tilde{v}_i$(2:n-i) _LARFG
and compute the scalars τ_{p_i} and e(i).
A(i,i+2:n) $\leftarrow v_i$(2:n-i), A(i,i+1)$\leftarrow e$(i).

4. (i) Recovering w_i from $\tilde{w}_i$ requires the updated
$i+1$st column of A (Eqn. (5.20))
$A^{(i)}$(i+1:m,i+1) $\leftarrow A^{orig}$(i+1:m,i+1) 2 _GEMVs
$- U_{mat}$(i+1:m,1:i-1)Z_{mat}(1:i-1,i+1)
$- W_{mat}$(i+1:m,1:i-1)V_{mat}(1:i-1,i+1).

(ii) $t \leftarrow \tilde{v}_i(1) - s$ where
$s \leftarrow \begin{cases} -\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) \geq 0 \\ +\|\tilde{v}_i\|, & \text{if } \tilde{v}_i(1) < 0 \end{cases}$
and compute (Eqns (4.33) and (4.26)),
$w_i \leftarrow \tau_{p_i} \left(\frac{\tilde{w}_i - sA^{(i)}(\text{i:m,i+1})}{t} \right)$. _AXPY

(iii) After $A^{(i)}$(i,i+1) is used to compute w_i,
A(i,i+1)$\leftarrow e$(i).
e(i) was computed in Step 3. Now the ith row
has been overwritten from $i+1$ to n.

5. $z_i \leftarrow x_i - \tau_{p_i}(x_i^T v_i)\, v_i$ (Eqn (4.27)). _DOT, _AXPY

6. W_{mat}(i:m,i)$\leftarrow w_i$, Z_{mat}(i,i:n)$\leftarrow z_i$
(If A was originally in dense storage no separate storage
is needed for u_i and v_i in the virtual arrays U_{mat} and

V_{mat} . They were loaded into the ith column and
ith row of the A matrix in Steps 1(ii) and Step 3.)
End For

This completes the pseudo-code for Algorithm II.[12] The processing of the $(n-1)$th and nth columns are similar to Algorithm I (on the eliminating the $(n-1)$th column, the update Eqn (5.20) must be performed.

Since Algorithm II does not change any row or column of A until the step on which they are eliminated, the only access of the trailing (perhaps sparse) matrix is for matrix vector multiplications. Thus Algorithm II gives a stable alternative to sparse Lanczos bidiagonalization. As such, it is useful for obtaining the first entries of a bidiagonal matrix, e.g. for a least squares algorithm such as LSQR [21] or for approximating a few singular values as in SVDPACK [2, 3, 4]. Obtaining the first k diagonal and k superdiagonal bidiagonal entries requires $6(m+n)k^2-8k^3$ flops _GEMV flops. For a sparse matrix with n_z nonzero entries, $4n_zk$ _GEMVT flops are needed. If the matrix is dense, then $4nmk-2(m+n)k+4k^3/3$ _GEMVT flops are required.

For complete bidiagonalization of a dense matrix, Algorithm II is impractical. For $m > n$, the total number of flops becomes $8mn^2 - 8/3n^3$, double that of Algorithms BLAS-2 and I. The stored update matrices U_{mat}, V_{mat}, W_{mat}, and Z_{mat} require (even if A is overwritten by U_{mat} and V_{mat}) twice the storage of the original matrix A. As the algorithm proceeds, the matrices $U_{mat}, V_{mat}, W_{mat}$, and Z_{mat} no longer fit in cache, so these _GEMV operations are "out-of-cache", performing only two flops for each read of a double precision number from RAM.

The following section applies Algorithm II as a border update in blocking bidiagonalization. Updates of the trailing matrix are BLAS-3. Algorithm II accesses of $U_{mat}, V_{mat}, W_{mat}$, and Z_{mat} are "in-cache".

6 Algorithm III - Block Updates

Algorithm III is a block partitioned algorithm modeled after the LAPACK algorithm _GEBRD of Dongarra, Hammarling, and Sorensen [11]. As with _GEBRD, writes of the trailing part of the matrix occur only after a border of rows and columns have been bidiagonalized. Algorithm III calls Algorithm II to reduce the borders. Comparing Algorithm I and Algorithm III, Algorithm

[12] In block diagonalization (Algorithm III in the next section) , k steps of Algorithm II are performed in a bordering algorithm. On the last step (kth) step, Steps 4 to 6 are unnecessary.

III writes the trailing part of A only one time for each border block (k rows and columns), so is typically faster than Algorithm I, which writes the trailing part of the matrix once for each column-row elimination. (Algorithm I is useful for the last unblocked step).

If blocks are of size N_b, then _GEBRD or Algorithm III requires $O(mnN_b)$ flops additional to the flops required by Algorithm I, where the additional flops are mainly (in-cache) _GEMV flops. For either _GEBRD or Algorithm III, $2mn^2 - 2/3n^3$ flops are BLAS-3 _GEMMs. For _GEBRD $2mn^2 - 2/3n^3$ flops are "out-of-cache" BLAS-2 _GEMVs where for Algorithm III these $2mn^2 - 2/3n^3$ flops are BLAS-2.5 _GEMVTs. Calling a _GEMVT as opposed to two _GEMVs requires a read of the trailing part of the matrix once per column-row elimination as opposed to twice. Thus Algorithm III requires only about half as many reads of A compared to _GEBRD. Summing up, Algorithm III has roughly half the number of reads of the LAPACK bidiagonalization _GEBRD and about the same number of writes.

Algorithm III consists of applying Algorithm II on each of k_{max} blocks of the A matrix, each block having N_b columns of A. Here k_{max} is the greatest integer less than or equal to n/N_b, where N_b is the block size. The last block consists of the last $(n - N_b\, k_{max})$ columns of A. The first k_{max} blocks are processed by successive calls to Algorithm II and the last block is processed by a single call to Algorithm I. On step I, let $m_{now} = m - N_b(I-1)$, $n_{now} = n - N_b(I-1)$. Algorithm III generates an ($m_{now} \times N_b$ matrix W_{mat} of N_b vectors $w^{(i)}$ and an $N_b \times n_{now}$ matrix Z_{mat} of N_b vectors $z^{(i)}$. The corresponding N_b vectors u_i are stored below the diagonal in the reduced part of A, and the corresponding N_b vectors v_i are stored to the right of the superdiagonal in the reduced part of A (so whenever the virtual arrays U_{mat} and V_{mat} are referenced, the appropriate storage locations of A must be accessed.) As each L-shaped block of N_b rows and N_b columns is processed, the trailing, unreduced part of the A matrix is updated all at once using equation (5.7) with $k = I\, N_b{+}1$, $p = m$, $l = I\, N_b{+}1$, $q = n$ where I denotes the Ith block of N_b rows and N_b columns. The block updates are accomplished by two calls to the BLAS-3 matrix-matrix multiplication routine _GEMM.

Note: Since the $w^{(i)}$ and $z^{(i)}$ vectors for the Ith block are needed only to update the trailing, unreduced part of the A matrix, the W_{mat} and Z_{mat} arrays can be zeroed out after the Ith block update, and reloaded with the $w^{(i)}$ and $z^{(i)}$ vectors for the $I+1$st block. Hence, the main additional storage requirement for Algorithm III in addition to the $m \times n$ A matrix is an $w \times N_b$ array for W_{mat} and a $N_b \times n$ array for Z_{mat}.

Algorithm III: Pseudo Code

1. Given m, n and matrix A and a block size $N_b < n$,
   $k_{max} \leftarrow [n/N_b]$.

For $I = 1 : k_{max}$,

2. For $i = 1$,
   Perform the Steps 1-6 of Algorithm I
   $W_{mat}((I\text{-}1)N_b\text{+}1\text{:m},1) \leftarrow w_1, Z_{mat}((1,(I\text{-}1)N_b\text{+}1\text{:n}) \leftarrow z_1$
   End For
3. For $i = 2 : N_b$,
   Perform steps 1-6 of Algorithm II
   $W_{mat}((I\text{-}1)N_b\text{+i:m},i) \leftarrow w_i, Z_{mat}((i,(I\text{-}1)N_b\text{+i:n}) \leftarrow z_i$
   End For
4. Update the trailing unreduced $IN_b+1 : m$ rows
   and $IN_b+1 : n$ columns of A using the block
   update formula (5.7) with $j = IN_b+1$ and $l = IN_b+1$:
   $A(IN_b + 1 : m, IN_b+1 : n) \leftarrow$
   $A^{orig}(IN_b+1 : m, IN_b + 1 : n)$ 2 BLAS-3
   $- U_{mat}(IN_b + 1 : m, 1 : N_b)\, Z_{mat}(1 : N_b, IN_b + 1 : n)$ _GEMMs
   $- W_{mat}(IN_b+1 : m, 1 : N_b)\, V_{mat}(1 : N_b, IN_b + 1 : n)$.
   where
   $U_{mat}(IN_b + 1 : m, 1 : N_b) := A(IN_b + 1 : m, 1 : N_b)$
   $V_{mat}(1 : N_b, IN_b + 1 : n) := A(1 : N_b, IN_b + 1 : n)$.
   Then $W_{mat} \leftarrow 0, Z_{mat} \leftarrow 0$

End For

5. [A] For $i = k_{max}N_b+1 : n-2$,
   Perform Steps 1-6 of Algorithm I.
   End For
   [B] For $i = n-1 : n$,
   As in Algorithm I, perform the last two column updates of A.
   End For

Algorithms I, II, and III were developed in Matlab and converted to Fortran 77 for inclusion in a future release of the LAPACK library. The Algorithm III pseudo code corresponds to the new routine _GEBRD2. _GEBRD2 has the same calling sequence as the current LAPACK routine _GEBRD and is interchangeable with _GEBRD in all LAPACK routines which call _GEBRD. Algorithm II corresponds to the new routine _LABR2 which accomplishes the same border reduction as LAPACK routine _LABRD. The Algorithm I pseudo code corresponds to the new routine _GEBD3 which accomplishes

the same unblocked bidiagonalization as LAPACK routine _GEBD2 (which implements the Algorithm BLAS-2 of Section 2). There are differences between the new routine _GEBD3 and the current routine _DGEBD2, and also between the new routine _LABR2 and the old routine _LABRD. .

7 Timing Data for Algorithm III and Some Notes on Algorithm Tuning

We have compiled and run DGEBRD2 on a variety of platforms, including SGI Irix, Compaq Alphas under Tru64, IBM Power5s under AIX, and Sun Solaris on UltraSparcs, and with several BLAS packages (used for DGEMM calls and for BLAS 2 calls from DGEMVT and DGEMVER). In almost all cases, DGEBRD2 executed significantly more quickly than did LAPACK using DGEBRD.

For Pentium III processors with 256K cache running Redhat 7.3, speedups were comparable to those reported here, but mainly disappeared when the operating system was changed to RHEL 3.1. We hypothesize that RHEL 3.1 uses the 256K cache to cache the operating system and program stack, so that little data caching can occur. The least speedup (only 4% to 5%) of the other processors was on the Power 5, which has a high bandwidth to RAM relative to CPU speed so that _GEMV matrix vector multiplications run fairly fast even if data is out of cache.

We report here our most recent results running under Linux on 2.8 GHz Xeons and 2.0 GHz Opterons at North Carolina State University. The Xeon processor has a cache size of 512 Kb and the Opteron has 2 Mb of cache. For the Xeons, the Atlas-tuned BLAS [25] (compiled with the gnu gcc) was linked to Intel ifc compiled versions of DGEBRD2 and the results compared to ifc compiled versions of DGEBRD2 linked to the same BLAS library. The DGEMVT and DGEMVER operators were implemented on top of the BLAS-2 DGEMV and DGER operators as described in Section 3. For the Opterons, we obtained our fastest DGEBRD2 results with ifc compiled code linked to the Intel provided library. The DGEBRD2 (Algorithm III) code is compiled with the same ifc flags and links to the same BLAS library.

Recall that _GEMVT, uses column blocking, making two _GEMV calls on each column block, (as in the _GEMVT Pseudo Code in Section 3). Each call to _GEMVT determines the number of columns in a column block by a call to ILAENV[13]. Tuning to a given processor is by fixing two parameters

[13]ILAENV is the LAPACK environment routine that returns information such as the block size N_b.

in ILAENV.

The first parameter is an upper bound C_1 on the number of double precision numbers for a column block of a matrix. An initial guess for the number of columns in a block is $k = C_1/m$ where m is the number of rows in the call to _GEMVT. The integer k is then truncated to be a multiple of C_2 where C_2 has been chosen by numerical experiment to give good performance in calls to _GEMV for a given BLAS library. Generally, one expects to have to choose C_2 so that the underlying _GEMV calls do not have to make clean-up steps, i.e., C_2 should be an even multiple of the _GEMV loop unrolling parameters.

As an additional refinement, if the initial guess $C_2/2 \leq k = C_1/m < C_2$, then take the number of columns as $C_2/2$ (allowing smaller column block sizes when the number of rows m is large.) If $k < C_2/2$, then take the block size as n, the number of columns in the call to _GEMVT. Thus if $m > 2C_1C_2$, no column blocking is used.

Fortran lines encapsulating this discussion for a Xeon with a 512Kbyte = 64K double precision number cache are

```
C1 = 40000
C2 = 12
NBMIN = C2
NX = M/C1
NB = MIN(NBMIN*(NX/NBMIN),160)
IF (NX.LT.NBMIN .AND.NX.GE.NBMIN/2) THEN
   NB = NBMIN/2
ELSEIF (NX.LT.NBMIN/2) THEN
   NB = N
ENDIF
```

Generally, algorithm performance is not very sensitive to the choice of C_1, but does fall off rapidly if C_1 is chosen large enough that column blocks do not fit in cache. If C_1 is chosen too small, then matrices with number of rows $m > C_1C_2/2$ do not use column blocking even though a usable column block might fit in cache. C_2 can be found either by timing _GEMVT for a fixed m and various block sizes, or by knowing the loop-unrolling parameters in the underlying BLAS. Generally, for each given architecture and choice of BLAS library, each flavor of arithmetic (complex, double complex, single precision, and double precision) will need its own choice of C_1 and C_2.

Tables 2 to 4 compare LAPACK timings to the current code. As detailed above, a primary difference between the LAPACK code and that im-

plemented here is that LAPACK uses two calls to _GEMV which this code accomplishes in one call to _GEMVT.

Timing comparisons of DGEBRD2 with LAPACK routine DGEBRD on the Xeon processor for square matrices ranging in size from 400 to 2000 are given in Table 3. All times are CPU times in seconds. The blocksize C_1 was 40000 double precision numbers (320KBytes) compared to an L2 cache size of 512Kbyes. C_2 = NBMIN is taken as 12.

Matrix Size	LAPACK DGEBRD Time	NEW DGEBRD2 Time	RATIO DGEBRD2 DGEBRD
400	0.21	0.17	0.81
600	0.70	0.56	0.80
800	1.57	1.19	0.76
1000	3.05	2.36	0.77
1200	4.91	3.80	0.77
1400	8.01	6.10	0.76
1600	11.52	9.52	0.83
1800	16.54	12.92	0.78
2000	23.06	17.45	0.76

Table 2: Times for Bidiagonalization on 2.8 GHz Xeon

For the same runs done in the above table, the megaflop rates for both DGEBRD2 and DGEBRD were computed. Since both algorithms have a total flop count on the order of $\frac{8}{3}\ N^3$ for square matrices, the megaflop rate was computed by

$$Mflops = \frac{\frac{8}{3}N^3}{10^6 * CpuSeconds} \tag{7.1}$$

Matrix Size	LAPACK DGEBRD MFLOPS	TIME	NEW DGEBRD2 MFLOPS	TIME	RATIO DGEBRD2 DGEBRD
200	711	.03	1391	.02	.67
400	1067	.16	1422	.12	0.75
600	1067	.54	1340	.43	0.72
800	1128	1.21	1339	1.02	0.80
1000	1175	2.27	1361	1.96	0.84
1200	1209	3.81	1392	3.31	0.87
1400	1242	5.89	1396	5.24	0.87
1600	1279	8.54	1435	7.61	0.89
1800	1288	12.1	1438	10.8	0.89
2000	1312	16.3	1439	14.9	0.91
2200	1326	21.4	1445	19.6	0.92
2400	1338	27.6	1452	25.4	0.92
2600	1391	33.7	1468	25.4	0.95
2800	1351	43.2	1463	40.0	0.93
3000	1364	52.8	1490	48.3	0.92

Table 4: Megaflop Rates and Times for Bidiagonalization on 2.0 GHz Opteron

Matrix Size	LAPACK DGEBRD MFLOPS	NEW DGEBRD2 MFLOPS	RATIO DGEBRD2 DGEBRD
400	812	1004	0.81
600	822	1029	0.80
800	870	1147	0.76
1000	874	1140	0.77
1200	938	1213	0.77
1400	914	1200	0.76
1600	948	1147	0.83
1800	940	1204	0.78
2000	925	1222	0.76

Table 3: Megaflop Rates for Bidiagonalization on a 2.8 GHz Xeon

Table 4 summarizes runs on on an Opteron 2 GHz processor with 2

Mbytes of cache memory. For these runs, _GEMVT took the number of columns in a block as a multiple of $C_2 = 12$. _GEMVT column blocks were taken to have $C_1 = 120K$ double precision numbers, i.e., 960KBytes, compared to an L2 cache size of 2Mbytes. The LAPACK routine and BLAS are taken from the Intel supplied libraries. Similar results are obtained with the AMD supplied libraries.

8 Complex and Transposed Cases, Stability and Robustness

The timing results are for the case of upper bidiagonalization of double precision matrices with more rows than columns. The case of lower bidiagonalization occurs when there are more columns than rows. We can handle this case is an analogous manner to that described, but with transposed BLAS-2.5 operations _GEMVTT and _GEMVERT. These operators use row blocking as opposed to column blocking. Timings for the algorithm coded in this fashion are not always better than the corresponding LAPACK codes, perhaps because the Fortran row blocking reads too many matrix entries in each column (cache-lines do not typically break evenly on the row blocks). Alternatively, if adequate storage is available, an inplace matrix transpose can be performed, in negligible time compared to bidiagonalization time. Then upper bidiagonalization can be performed, and afterwards another inplace transpose can recover the lower bidiagonal form (and the same returned entries corresponding to the Householder vectors). For the double precision case, both versions of the lower bidiagonalization code exist.

For complex bidiagonalization, the algorithm is slightly changed from the double precision case. Fortran 77 (and Matlab) versions of this code also exist, with the Matlab code also working for floating point reals. Speedups are less marked in the complex case, reflecting a quadrupling of computations, compared to a doubling of data transferred, so that for the complex algorithm data transfer takes a lesser part of the total algorithm execution time.

The _GEBRD2 code exists as DGEBRD2, ZGEBRD2, SGEBRD2 and CGEBRD2 (double precision, complex double precision, single precision, and complex single precision) forms. Similarly, there are four versions of each of _GEMVER, _GEMV, _GEBD2, and _LABRD3.

Testing routines for the Singular Value Decomposition routines in LAPACK [6] supply numerous tests of stability and robustness for 16 different types of test matrices. We inserted our new routine DGEBRD2 into the

LAPACK package of routines, and ran the testing routines for LAPACK routines DGESVD and DGESDD ([6], Sec. 7.8), using the input file *svd.in* supplied by the LAPACK distribution which generates a number of $m \times n$ test matrices with m and n ranging from 1 to 50. All tests were passed.

9 Summary and Conclusions

In this paper the sequence of operations for the usual Golub-Kahan Householder bidiagonalization [16] of a general $m \times n$ matrix is reorganized so that the two (_GEMV) vector-matrix multiplications can be done with one pass of the unreduced trailing part of the matrix through cache. Left and right matrix-vector multiplications by respective right and left Householder vectors are required at each column-row elimination. In order to accomplish both matrix-vector multiplications in one pass of the matrix through cache, the reorganized code makes the right multiplication using a pre-Householder vector which is the row vector to be eliminated. Scaling and a BLAS-1 correction are used to obtain the matrix vector product for the actual right Householder vector. The rearrangement of the order of computations gives rise to three algorithms: Algorithm I does a straightforward rank-2 update on the trailing part of the matrix at each column row elimination, Algorithm II never performs the rank-2 updates (at the cost of doubling the operation count if run to completion), and Algorithm III defers the rank-2 updates performing them as two BLAS-3 matrix multiplications (_GEMM) after each block of k columns and rows have been eliminated. The latter, a blocked version of Algorithm II, represents a version of the new algorithm which enjoys a blocking similar to LAPACK routine _GEBRD [9]. The two new BLAS 2.5 operations (_GEMVER and _GEMVT) from the new BLAST standard [7], which cut in half the data transfer from main memory to cache, were introduced by the first author.

The above timing data on 2.8 GHz Xeon processors and 2.0 GHz Opteron processors shows that our new Algorithm III as implemented in our new subroutine DGEBR3 executes significantly faster than the LAPACK routine DGEBRD. For square matrices of size 400 to 2000 with increment 200, the CPU time for subroutine DGEBR3 was 75% to 80% of the time required by LAPACK routine DGEBRD on the Xeon processors, and ranged from 75% to 90% on the Opteron processors.

Grösser and Lang [19] and Lang [20] have implemented a parallel reduction to bidiagonal form which subdivides the reduction into two stages, dense to banded, and banded to bidiagonal. The two stage bidiagonalization

[19] requires $4mn^2 - 4/3n^3$ almost all BLAS-3 flops for a reduction too small band form, then a further slow $8k^2n$ flops for a reduction from bandwidth k to bidiagonal. For sufficiently large matrices, the Grõsser-Lang algorithm is likely to be faster than the algorithms described here. When singular vectors are desired, the Grösser-Lang decomposition requires an extra stage in reconstructing matrix singular vectors from the singular vectors of bidiagonal matrices, so that Algorithm III presented here is likely to require less time.

The column oriented Algorithm I may be useful for matrices so small that the entire matrix fits in L2 cache, and a few columns fit in L1 cache, and is used as a clean-up step. Algorithm II is useful for sparse matrices for which only the first few entries of the bidiagonal matrix are required. Algorithm III has been adapted for inclusion in the LAPACK package.

10 Acknowledgments

We wish to thank Ken Stanley for his advice on cache-efficient algorithms.

References

[1] E. Andersen, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchev, D. Sorensen, *LAPACK User's Guide*, 3rd. Ed. 1999 SIAM, Philadelphia.

[2] M. Berry, *Large scale singular value computations*, Internat. J. Supercomputer Appl., 6:13-49, 1992.

[3] M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan, *SVDPACKC: Version 1.0 User's Guide*, Tech. Report CS-93-194, University of Tennessee, Knoxville, TN, October, 1993.

[4] M. Berry, S. Dumais, and G. O'Brien, *Using Linear Algebra for Intelligent Information Retrieval*, SIAM Review, 37(4):573–595, 1995.

[5] C. H. Bischof and C. F. Van Loan, *The WY Representaion of Products of Householder Matrices*, SIAM J. Sci. Stat. Comput, 8:s2-s13, 1987.

[6] S. Blackford and J. Dongarra, *Installation Guide for LAPACK*, LAPACK Working Note 41, June, 1999.

[7] BLAST 2000-Standard. available at http://www.netlib.org/cgi-bin/checkout/blast/blast.p1 . Completed on November 1, 2000.

[8] BLAS technical forum, *www.netlib.org/utk/papers/blast-forum.html*, 1999.

[9] J. Choi, J. Dongarra, and D. Walker *The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form.* (LAPACK Working Note # 92) Num. Alg., 10:379–399, 1995.

[10] I.S. Dhillon *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* PhD thesis, University of California, Berkeley, California, May 1997.

[11] J. Dongarra, S. Hammarling, and D. Sorensen, *Block Reduction of matrices to condensed forms for eigenvalue computations.* J. Comput. Appl. Math., 27:215–227, 1989.

[12] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*, 1998, SIAM, Philadelphia.

[13] C.C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss,*Portable Memory Heirarchy Techniques for PDE Solvers: Part I*, SIAM News, 33, 5, June 2000.

[14] V. Fernando, B. Parlett, and I. Dhillon, *A way to find the most redundant equation in a tridiagonal system.* Berkeley Mathematics Dept. Preprint, 1995.

[15] S. Goedecker and A. Hoise,*Performance Optimization for Numerically Intensive Codes*, 2001, SIAM, Philadelphia.

[16] G. Golub and W. Kahan, *Calculating the Singular Values and Pseudo-Inverse of a Matrix*, *SIAM J. Num. Anal.* 2:205–24, 1965.

[17] G. Golub and C. Reinsch, *Singular Value Decomposition and Least Squares Solution Matrix.* Numer. Math., 14:403–420, 1970.

[18] G. Golub and C. F. Van Loan, *Matrix Computations*, 3rd Ed., The Johns Hopkins University Press, Baltimore, 1996.

[19] B. Grösser and B. Lang, *Efficient Parallel Reduction to Bidiagonal Form*, Preprint BUGHW-SC 98/2 (Available from http://www.math.uni-wuppertal/)

[20] B. Lang, *Parallel reduction of banded matrices to bidiagonal form* Parallel Comput., 22 (1996), 1-18.

[21] C. Paige and M. Saunders. *An Algorithm for Sparse Linear Equations and Sparse Least Squares*. ACM Trans. on Math. Software, 8(1), 43–71, 1982.

[22] B. Parlett and I. Dhillon, *Fernando's solution to Wilkinson's problem: An application of double factorization* , Lin. Alg. Appl., 267:247–279, 1997.

[23] R. Schreiber and C. F. Van Loan, *A Storage-Efficient WY Representation for Products of Householder Transformations*, SIAM Scientific and Statistical Computing, 10:53-57, 1989.

[24] K. Stanley,*Execution Time of Symmetric Eigensolvers*, Ph.D. dissertation, 1997, CS, University of California, Berkeley.

[25] C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999, available on CD-ROM from SIAM.