Algorithms for Reducing a Matrix to Condensed Form

FLAME Working Note #53

Field G. Van Zee*
Robert A. van de Geijn*
Gregorio Quintana-Ortí†
G. Joseph Elizondo*

October 30, 2010
Revised January 13, 2012

Abstract

In a recent paper it was shown how memory traffic can be diminished by reformulating the classic algorithm for reducing a matrix to bidiagonal form, a preprocess when computing the singular values of a dense matrix. The key is a reordering of the computation so that the most memory-intensive operations can be "fused". In this paper, we show that other operations that reduce matrices to condensed form (reduction to upper Hessenberg form and reduction to tridiagonal form) can be similarly reorganized, yielding different sets of operations that can be fused. By developing the algorithms with a common framework and notation, we facilitate the comparing and contrasting of the different algorithms and opportunities for optimization on sequential architectures. We discuss the algorithms, develop a simple model to estimate the speedup potential from fusing, and showcase performance improvements consistent with the what the model predicts.

1 Introduction

For many dense linear algebra operations, such as Cholesky, LU, and QR factorizations, there exist algorithms that cast most of the computation in terms of matrix-matrix operations that can overcome the memory bandwidth bottleneck common to most modern processors [12, 11, 9, 1]. Reduction to condensed form operations—specifically, reduction to upper Hessenberg, tridiagonal, and bidiagonal form—are important exceptions. For these operations, reducing the number of times data must be brought in from memory is the key to optimizing performance since inherently $\mathcal{O}(n^3)$ reads to and writes from memory are incurred while $\mathcal{O}(n^3)$ floating-point operations are performed on an $n \times n$ matrix.

It should be noted that there are algorithms for reduction to condensed form based on successive band reduction that cast most computation in terms of cache-efficient matrix-matrix operations [5, 17, 7, 3]. Such algorithms are much faster than those presented in the present paper. However, reduction to condensed form is typically not a useful operation in isolation. While successive band reduction yields a faster reduction to condensed form, it adversely affects the performance of other parts of eigensolvers and/or SVD computations. The present paper does not compare against successive band reduction precisely because the authors believe that such a comparison is only meaningful in the context of a complete eigensolver or SVD solver. Thus, we only give a comprehensive treatment of *direct* algorithms for reduction to condensed form.

The Basic Linear Algebra Subprograms (BLAS) [19, 10, 9] provide an interface to commonly used computational kernels in terms of which linear algebra routine can be written. The idea is that if these kernels are optimized, then implementations of algorithms for computing more complex operations benefit in a portable

*Department of Computer Science, The University of Texas at Austin, Austin, TX, 78712.
†Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Campus Riu Sec, 12.071, Castellón, Spain,

fashion. As we will see, the problem is that the interface itself is limiting and can stand in the way of minimizing memory traffic. In response, as part of the BLAST Forum [8], additional, more complex, operations were suggested for inclusion in the BLAS. Unfortunately, the extensions proposed by the BLAST forum are not as well-supported as the original BLAS. In [15], it was shown how one of the reduction to condensed form operations, reduction to bidiagonal form, benefits from this new functionality in the BLAS.

This paper presents algorithms for all three major reduction to condensed form operations (reduction to upper Hessenberg, tridiagonal, and bidiagonal form) with the FLAME notation [13]. This facilitates comparing and contrasting different algorithms for the same operation and similar algorithms for different operations [22, 13, 2, 27]. The paper shows how the techniques used to reduce memory traffic in the reduction to bidiagonal form algorithm, already reported in [15], can be applied to similarly reduce such traffic when computing a reduction to upper Hessenberg or tridiagonal form (although each has different potential for improvement). It identifies sequences of operations within the algorithms for reduction to condensed form that can be "fused." (A sequence of operations is eligible for fusing when the operations share one or more operands in common, allowing the computations to be merged in an effort to reduce the cost due to memory traffic.) Such compound operations have been referred to as "Level-2.5 BLAS." It demonstrates the relative merits of different algorithms and optimizations that combine algorithms on a recent sequential architecture. Additionally, the paper illustrates the difference between two styles of fusing, "cache-level" fusing and "register-level" fusing, and in doing so exposes why the latter yields superior performance. All the presented algorithms are implemented as part of the `libflame` library [28, 29]. Thus the paper also provides documentation for that library's support of the target operations. The family of implementations and related benchmarking codes are available as part of `libflame` so that others can experiment with optimizations of the fused operations and the effect on performance. And finally, we include an electronic appendix that (1) redefines Householder transformations in the complex domain, and (2) gives examples of how the algorithms would change to accomodate complex matrices.

2 Householder transformations (Reflectors)

We start by reviewing a few basic properties of Householder transformations. For simplicity, we focus only on computation over real matrices. However, the algorithms and results presented in this paper generalize to the complex domain, and a related technical report [30] gives examples of how to express the computation accordingly.

2.1 Computing Householder vectors and transformations

Definition 1 *Let $u \in \mathbb{R}^n$, $\tau \in \mathbb{R}$. Then $H = H(u) = I - uu^T/\tau$, where $\tau = \frac{1}{2}u^T u$, is said to be a reflector or Householder transformation.*

We observe:

- Let z be any vector that is perpendicular to u. Applying a Householder transform $H(u)$ to z leaves the vector unchanged: $H(u)z = z$.
- Let any vector x be written as $x = z + u^T xu$, where z is perpendicular to u and $u^T xu$ is the component of x in the direction of u. Then $H(u)x = z - u^T xu$.

This can be interpreted as follows: The space perpendicular to u acts as a "mirror": any vector in that space (along the mirror) is not reflected, while any other vector has the component that is orthogonal to the space (the component outside and orthogonal to the mirror) reversed in direction. Notice that a reflection preserves the length of the vector. Also, it is easy to verify that:

1. $HH = I$ (reflecting the reflection of a vector results in the original vector);
2. $H = H^T$, and so $H^T H = HH^T = I$ (a reflection is an orthogonal matrix and thus preserves the norm); and

3. if $H_0, \cdots, H_{k-1}$ are Householder transformations and $Q = H_0 H_1 \cdots H_{k-1}$, then $Q^T Q = QQ^T = I$ (an accumulation of reflectors is an orthogonal matrix).

As part of the reduction to condensed form operations, given a vector x we will wish to find a Householder transformation, $H(u)$, such that $H(u)x$ equals a vector with zeros below the first element: $H(u)x = \mp\|x\|_2 e_0$ where e_0 equals the first column of the identity matrix. It can be easily checked that choosing $u = x \pm \|x\|_2 e_0$ yields the desired $H(u)$. Notice that any nonzero scaling of u has the same property, and the convention is to scale u so that the first element equals one. Let us define $[u, \tau, h] = \textsc{Housev}(x)$ to be the function that returns u with first element equal to one, $\tau = \frac{1}{2}u^T u$, and $h = H(u)x$.

2.2 Computing Au from Ax

Later, we will see that given a matrix A, we will need to form Au where u is computed by $\textsc{Housev}(x)$, but we will do so by first computing Ax. Let

$$x \rightarrow \left(\begin{array}{c}\chi_1 \\ \hline x_2\end{array}\right), \quad v \rightarrow \left(\begin{array}{c}\nu_1 \\ \hline v_2\end{array}\right), \quad u \rightarrow \left(\begin{array}{c}\upsilon_1 \\ \hline u_2\end{array}\right),$$

$v = x - \alpha e_0$, and $u = v/\nu_1$, with $\alpha = -\text{sign}(\chi_1)\|x\|_2$ (and thus $\upsilon_1 = 1$). Then

$$\begin{aligned}
\|x\|_2 &= \left\| \left(\begin{array}{c}\chi_1 \\ \hline \|x_2\|_2\end{array}\right) \right\|_2, \quad \|v\|_2 = \left\| \left(\begin{array}{c}\chi_1 - \alpha \\ \hline \|x_2\|_2\end{array}\right) \right\|_2, \quad \|u\|_2 = \|v\|_2/(\chi_1 - \alpha), && (1)\\
\tau &= \frac{u^T u}{2} = \frac{\|u\|_2^2}{2} = \frac{\|v\|_2^2}{2(\chi_1 - \alpha)^2}, && (2)\\
w &= Ax \quad \text{and} \quad Au = \frac{A(x - \alpha e_0)}{(\chi_1 - \alpha)} = \frac{(w - \alpha A e_0)}{(\chi_1 - \alpha)}. && (3)
\end{aligned}$$

We note that Ae_0 simply equals the first column of A. We will assume that various results in Eq. (1)–(2) are computed by the function $\textsc{Houses}(x)$ where $[\chi_1 - \alpha, \tau, \alpha] = \textsc{Houses}(x)$.[1] Then, the desired vector Au can be computed via Eq. (3).

2.3 Accumulating transformations

Consider the transformation formed by multiplying b Householder transformations $\left(I - u_j u_j^T/\tau_j\right)$, for $0 \leq j < b-1$. If $U = \left(\begin{array}{c|c|c|c} u_0 & u_1 & \cdots & u_{b-1} \end{array} \right)$, then

$$\left(I - u_0 u_0^T/\tau_0\right)\left(I - u_1 u_1^T/\tau_1\right) \cdots \left(I - u_{b-1} u_{b-1}^T/\tau_{b-1}\right) = (I - UT^{-1}U^T).$$

Here $T = \frac{1}{2}D + S$ where D and S equal the diagonal and strictly upper triangular parts of $U^T U = S^T + D + S$. Later we will use the fact that if

$$U = \left(\begin{array}{c|c} U_0 & u_1 \end{array} \right) \quad \text{and} \quad T = \left(\begin{array}{c|c} T_{00} & t_{01} \\ \hline 0 & \tau_{11} \end{array}\right)$$

then

$$t_{01} = U_0^T u_1, \quad \tau_{11} = \frac{u_1^T u_1}{2}, \quad \text{and} \quad \left(\begin{array}{c|c} T_{00} & t_{01} \\ \hline 0 & \tau_{11} \end{array}\right)^{-1} = \left(\begin{array}{c|c} T_{00}^{-1} & -T_{00}^{-1} t_{01}/\tau_{11} \\ \hline 0 & \tau_{11}^{-1} \end{array}\right).$$

For further details, see [16, 21, 26, 32]. Alternative ways for accumulating transformations are the WY-transform [6] and compact WY-transform [24].

[1] Here, $\textsc{Houses}$ stands for "Householder scalars", in contrast to the function $\textsc{Housev}$ which provides the Householder vector u.

3 Reduction to upper Hessenberg form

In the first step towards computing the Schur decomposition of a matrix A, the matrix is reduced to upper Hessenberg form: $A \rightarrow QBQ^T$ where B is an upper Hessenberg matrix (zeros below the first subdiagonal) and Q is orthogonal.

3.1 Unblocked algorithm

The basic algorithm for reducing the matrix to upper Hessenberg form, overwriting the original matrix with the result, can be explained as follows.

- Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
- Let $[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$.[2]
- Update

$$\left(\begin{array}{c|c} a_{01} & A_{02} \\ \hline \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & H \end{array}\right) \left(\begin{array}{c|c} a_{01} & A_{02} \\ \hline \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & H \end{array}\right) = \left(\begin{array}{c|c} a_{01} & A_{02}H \\ \hline \alpha_{11} & a_{12}^T H \\ \hline Ha_{21} & HA_{22}H \end{array}\right)$$

  where $H = H(u_{21})$. Note that $a_{21} := Ha_{21}$ need not be executed since this update was performed by the instance of $\textsc{Housev}$ above.[3]
- Continue this process with the updated A_{22}.

This is captured in the algorithm in Figure 1 (top), in which it is recognized that as the algorithm proceeds beyond the first iteration, the submatrix A_{20} must also be updated. As formulated, the submatrix A_{22} has to be read and written in the first highlighted operation and submatrices A_{02}, a_{12}^T, and A_{22} must be read and written in the second highlighted operation in Figure 1 (top) *assuming the operations in the highlighted boxed are fused*. Thus, the bulk of memory operations then lie with A_{22} being read and written twice and A_{20} being read and written once.

Let us look at the update of A_{22} in Figure 1 (top) in more detail:

$$\begin{aligned} A_{22} &:= HA_{22}H = (I - u_{21}u_{21}^T/\tau)A_{22}(I - u_{21}u_{21}^T/\tau) \\ &= A_{22} - u_{21}(\underbrace{A_{22}^T u_{21}}_{v_{21}})^T/\tau - (\underbrace{A_{22}u_{21}}_{w_{21}})u_{21}^T/\tau + (u_{21}^T \underbrace{A_{22}u_{21}}_{w_{21}})u_{21}u_{21}^T/\tau^2 \\ &= A_{22} - u_{21}v_{21}^T/\tau - w_{21}u_{21}^T/\tau + \underbrace{u_{21}^T w_{21}}_{2\beta} u_{21}u_{21}^T/\tau^2 \\ &= A_{22} - u_{21}\ (\underbrace{(v_{21} - \beta u_{21}/\tau)/\tau)}_{y_{21}})^T - \underbrace{((w_{21} - \beta u_{21}/\tau)/\tau)}_{z_{21}} u_{21}^T \\ &= A_{22} - (u_{21}y_{21}^T + z_{21}u_{21}^T). \end{aligned}$$

This motivates the algorithm in Figure 1 (left). The problem with this algorithm is that, when implemented using traditional level-2 BLAS, it requires A_{22} to be read four times and written twice. If the operations in the highlighted boxes are instead fused, then A_{22} needs only be read twice and written once.

[2]Note that the semantics here indicate that a_{21} is overwritten by Ha_{21}.

[3]In practice, the zeros below the first element of Ha_{21} are not actually written. Instead, the implementation overwrites these elements with the corresponding elements of the vector u_{21}.

Algorithm: $[A] := \textsc{HessRed\_unb}(b, A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $u \rightarrow \left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, $z \rightarrow \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right)$

where A_{TL} is 0×0 and u_T, y_T, and z_T have 0 rows

while $m(A_{TL}) < b$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right) \rightarrow \left(\begin{array}{c} u_{01} \\ \hline \upsilon_{11} \\ \hline u_{21} \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_{01} \\ \hline \psi_{11} \\ \hline y_{21} \end{array}\right), \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \rightarrow \left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right)$$

where α_{11}, υ_{11}, ψ_{11}, ζ_{11} are scalars

Basic unblocked 1:

$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$

$A_{22} := (I - u_{21}u_{21}^T/\tau)A_{22} = A_{22} - u_{21}u_{21}^T A_{22}/\tau$

$$\left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right)(I - u_{21}u_{21}^T/\tau) = \left(\begin{array}{c} A_{02} - A_{02}u_{21}u_{21}^T/\tau \\ \hline a_{12}^T - a_{12}^T u_{21}u_{21}^T/\tau \\ \hline A_{22} - A_{22}u_{21}u_{21}^T/\tau \end{array}\right)$$

Basic unblocked 2:

$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$

$y_{21} := A_{22}^T u_{21}$
$z_{21} := A_{22} u_{21}$

$\beta := u_{21}^T z_{21}/2$
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$
$z_{21} := (z_{21} - \beta u_{21}/\tau)/\tau$
$A_{22} := A_{22} - u_{21}y_{21}^T - z_{21}u_{21}^T$
$a_{12}^T := a_{12}^T - a_{12}^T u_{21}u_{21}^T/\tau$
$A_{02} := A_{02} - A_{02}u_{21}u_{21}^T/\tau$

Rearranged unblocked:

$\alpha_{11} := \alpha_{11} - \upsilon_1\psi_1 - \zeta_1\upsilon_1$ $(\star)$
$a_{12}^T := a_{12}^T - \upsilon_1 y_{21}^T - \zeta_1 u_{21}^T$ $(\star)$
$a_{21} := a_{21} - u_{21}\psi_1 - z_{21}\upsilon_1$ $(\star)$
$[x_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$
$A_{22} := A_{22} - u_{21}y_{21}^T - z_{21}u_{21}^T$ $(\star)$
$v_{21} := A_{22}^T x_{21}$
$w_{21} := A_{22} x_{21}$
$u_{21} := x_{21}$; $y_{21} := v_{21}$
$z_{21} := w_{21}$
$\beta := u_{21}^T z_{21}/2$
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$
$z_{21} := (z_{21} - \beta u_{21}/\tau)/\tau$

$a_{12}^T := a_{12}^T - a_{12}^T u_{21}u_{21}^T/\tau$
$A_{02} := A_{02} - A_{02}u_{21}u_{21}^T/\tau$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right) \leftarrow \left(\begin{array}{c} u_{01} \\ \hline \upsilon_{11} \\ \hline u_{21} \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_{01} \\ \hline \psi_{11} \\ \hline y_{21} \end{array}\right), \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \leftarrow \left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right)$$

endwhile

Figure 1: Unblocked algorithms for reduction to upper Hessenberg form. The first and second fused operations in the "Basic unblocked 2" algorithm correspond to the BLAS 2.5 operations GEMVT and GER2, respectively [8]. Operations marked with $(\star)$ are not executed during the first iteration.

What we will show next is that by delaying the update $A_{22} := A_{22} - (u_{21}y_{21}^T + z_{21}u_{21}^T)$ until the next iteration, we can reformulate the algorithm so that A_{22} needs only be read and written once per iteration. Let us focus on the update $A_{22} := A_{22} - (u_{21}y_{21}^T + z_{21}u_{21}^T)$. Partition

$$A_{22} \rightarrow \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right), \quad u_{21} \rightarrow \left(\begin{array}{c} \upsilon_1^+ \\ \hline u_{21}^+ \end{array}\right), \quad y_{21} \rightarrow \left(\begin{array}{c} \psi_1^+ \\ \hline y_{21}^+ \end{array}\right), \quad z_{21} \rightarrow \left(\begin{array}{c} \zeta_1^+ \\ \hline z_{21}^+ \end{array}\right),$$

where + indicates the partitioning in the next iteration. Then $A_{22} := A_{22} - (u_{21}y_{21}^T + z_{21}u_{21}^T)$ translates to

$$\begin{aligned} \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right) &:= \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right) - \left(\left(\begin{array}{c} \upsilon_1^+ \\ \hline u_{21}^+ \end{array}\right) \left(\begin{array}{c} \psi_1^+ \\ \hline y_{21}^+ \end{array}\right)^T + \left(\begin{array}{c} \zeta_1^+ \\ \hline z_{21}^+ \end{array}\right) \left(\begin{array}{c} \upsilon_1^+ \\ \hline u_{21}^+ \end{array}\right)^T \right) \\ &= \left(\begin{array}{c|c} \alpha_{11}^+ - (\upsilon_1^+\psi_1^+ + \zeta_1^+\upsilon_1^+) & a_{12}^{+T} - (\upsilon_1^+ y_{21}^{+T} + \zeta_1^+ u_{21}^{+T}) \\ \hline a_{21}^+ - (u_{21}^+\psi_1^+ + z_{21}^+\upsilon_1^+) & A_{22}^+ - (u_{21}^+ y_{21}^{+T} + z_{21}^+ u_{21}^{+T}) \end{array}\right), \end{aligned}$$

which shows what computation would need to be performed if the update of A_{22} is delayed until the next iteration. Now, before $v_{21} = A_{22}^T u_{21}$ and $z_{21} = A_{22}u_{21}$ can be computed in the next iteration, $\textsc{Housev}(a_{21})$ has to be computed, which requires a_{21} to be updated. But what is important is that A_{22} can be updated by the two rank-1 updates from the previous iterations just before $v_{21} = A_{22}^T u_{21}$ and $w_{21} = A_{22}u_{21}$ are computed, which allows them to be "fused" into one operation that reads and writes A_{22} to and from memory only once. The algorithm in Figure 1 (right) takes advantage of these insights. To our knowledge it has not been previously published.

3.2 Lazy algorithm

We now show how the reduction to upper Hessenberg form can be restructured so that the update $A_{22} := A_{22} - (u_{21}y_{21}^T + z_{21}u_{21}^T)$ during each step can be avoided. This algorithm by itself is not practical, since (1) it requires too much temporary space, and (2) intermediate matrix-vector multiplications, which incur additional memory reads, eventually begin to dominate the operation. But it will become an integral part of the blocked algorithm discussed in Section 3.4. This algorithm was first reported in [12].

The rather curious choice of subscripts for u_{21}, and y_{21}, and z_{21} now becomes apparent: By passing matrices U, Y, and Z into the algorithm in Figure 1, and partitioning them just like we do A in that algorithm, we can accumulate the subvectors u_{21}, y_{21} and z_{21} into those matrices. Now, let us assume that at the top of the loop A_{BR} has not yet been updated. Then α_{11}, a_{21}, a_{12}^T and A_{22} have not yet been updated, which means we cannot perform many of the computations in the current iteration. However, if we let $\hat{\alpha}_{11}$, $\hat{a}_{21}$, $\hat{a}_{12}^T$, and $\hat{A}_{22}$ denote the original values in A in those locations, then the desired α_{11}, a_{21}, and a_{12}^T are given by

$$\begin{aligned} \alpha_{11} &= \hat{\alpha}_{11} - u_{10}^T y_{10} - z_{10}^T u_{10} \\ a_{21} &= \hat{a}_{21} - U_{20}^T y_{10} - Z_{20}^T u_{10} \\ a_{12}^T &= \hat{a}_{12}^T - u_{10}^T Y_{20}^T - z_{10}^T U_{20}^T \\ A_{22} &= \hat{A}_{22} - U_{20}Y_{20}^T - Z_{20}U_{20}^T. \end{aligned}$$

Thus, we start the iteration by updating in this fashion these parts of A.

Next, we observe that the updated A_{22} itself is not actually needed in updated form: We need to be able to compute $A_{22}^T u_{21}$ and $A_{22}u_{21}$. But this can be done via the alternative computations

$$\begin{aligned} y_{21} &:= A_{22}^T u_{21} = \hat{A}_{22}^T u_{21} - Y_{20}(U_{20}^T u_{21}) - U_{20}(Z_{20}^T u_{21}) \\ z_{21} &:= A_{22}u_{21} = \hat{A}_{22}u_{21} - U_{20}(Y_{20}^T u_{21}) - Z_{20}(U_{20}^T u_{21}) \end{aligned}$$

which requires only matrix-vector multiplications. This inspires the algorithm in Figure 2.

Algorithm: $[A, U, Y, Z] := \textsc{HessRed\_lazy\_unb}(b, A, U, Y, Z)$

Partition $X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$

for $X \in \{A, U, Y, Z\}$

where X_{TL} is 0×0

while $n(U_{TL}) < b$ **do**

Repartition

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, v), (Y, y, \psi), (Z, z, \zeta)\}$

where χ_{11} is a scalar

$\alpha_{11} := \alpha_{11} - u_{10}^T y_{10} - z_{10}^T u_{10}$
$a_{21} := a_{21} - U_{20} y_{10} - Z_{20} u_{10}$
$a_{12}^T := a_{12}^T - u_{10}^T Y_{20}^T - z_{10}^T U_{20}^T$
$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$

$y_{21} := A_{22}^T u_{21}$
$z_{21} := A_{22} u_{21}$

$y_{21} := y_{21} - Y_{20}(U_{20}^T u_{21}) - U_{20}(Z_{20}^T u_{21})$
$z_{21} := z_{21} - U_{20}(Y_{20}^T u_{21}) - Z_{20}(U_{20}^T u_{21})$

$\beta := u_{21}^T z_{21}/2$
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$
$z_{21} := (z_{21} - \beta u_{21}/\tau)/\tau$
$a_{12}^T := a_{12}^T - a_{12}^T u_{21} u_{21}^T/\tau$
$A_{02} := A_{02} - A_{02} u_{21} u_{21}^T/\tau$

Continue with

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, v), (Y, y, \psi), (Z, z, \zeta)\}$

endwhile

Figure 2: Lazy unblocked algorithm for reduction to upper Hessenberg form. The first fused operation corresponds to the BLAS 2.5 operation GEMVT [8].

3.3 GQvdG unblocked algorithm

The lazy algorithm discussed above requires at each step a matrix-vector and a transposed matrix-vector multiply which can be fused so that the matrix only needs to be brought into memory once. In this section, we show how the bulk of computation (and associated memory traffic) can be cast in terms of a single matrix multiplication per iteration with a much simpler algorithm that does not require fusing and thus no special implementation of the fused operation. This algorithm was first proposed by G. Quintana and van de Geijn in [23], which is why we call it the GQvdG unblocked algorithm. It is summarized in Figure 3.

The underlying idea builds upon how Householder transformations can be accumulated: The first b updates can be accumulated into a lower trapezoidal matrix U and upper triangular matrix T so that

$$\left(I - u_0 u_0^T/\tau_0\right)\left(I - u_1 u_1^T/\tau_1\right)\cdots\left(I - u_{b-1} u_{b-1}^T/\tau_{b-1}\right) = (I - UT^{-1}U^T).$$

Algorithm: $[A, U, Z, T] := \textsc{HessRed\_GQvdG\_unb}(b, A, U, Z, T)$

Partition $X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$

for $X \in \{A, U, Z, T\}$

where X_{TL} is 0×0

while $n(U_{TL}) < b$ **do**

Repartition

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (Z, z, \zeta), (T, t, \tau)\}$

where χ_{11} is a scalar

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) - \left(\begin{array}{c} Z_{00} \\ \hline z_{10}^T \\ \hline Z_{20} \end{array}\right) T_{00}^{-1} u_{10}$$

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right) T_{00}^{-1} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array}\right)^T \right)^T \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$[u_{21}, \tau_{11}, a_{21}] := \textsc{Housev}(a_{21})$

$$\left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right) := \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) u_{21}$$

$t_{01} := U_{20}^T u_{21}$

Continue with

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (Z, z, \zeta), (T, t, \tau)\}$

endwhile

Figure 3: GQvdG unblocked algorithm for the reduction to upper Hessenberg form.

After b iterations the basic unblocked algorithm overwrites matrix A with

$$\begin{aligned} A^{(b)} &= H(u_{b-1}) \cdots H(u_0) \hat{A} H(u_0) \cdots H(u_{b-1}) \\ &= \left(I - u_{b-1} u_{b-1}^T / \tau_{b-1}\right) \cdots \left(I - u_0 u_0^T / \tau_0\right) \hat{A} \left(I - u_0 u_0^T / \tau_0\right) \cdots H(u_{b-1}) \\ &= (I - U T^{-1} U^T)^T \hat{A} (I - U T^{-1} U^T) = (I - U T^{-1} U^T)^T (\hat{A} - \underbrace{\hat{A} U}_{Z} T^{-1} U^T) \\ &= (I - U T^{-1} U^T)^T (\hat{A} - Z T^{-1} U^T), \end{aligned}$$

where $\hat{A}$ denotes the original contents of A.

Let us assume that this process has proceeded for k iterations. Partition

$$X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \text{ for } X \in \{A, \hat{A}, U, Z, T\},$$

where X_{TL} is $k \times k$. Then

$$A^{(k)} = \left(\begin{array}{c|c} A_{TL}^{(k)} & A_{TR}^{(k)} \\ \hline A_{BL}^{(k)} & A_{BR}^{(k)} \end{array} \right) =$$

$$\left(I - \left(\frac{U_{TL}}{U_{BL}} \right) T_{TL}^{-1} \left(\frac{U_{TL}}{U_{BL}} \right)^T \right)^T \left(\left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array} \right) - \left(\frac{Z_{TL}}{Z_{BL}} \right) T_{TL}^{-1} \left(\frac{U_{TL}}{U_{BL}} \right)^T \right).$$

Now, assume that after the first k iterations our algorithm leaves our variables in the following states:

- $A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$ contains $\left(\begin{array}{c|c} A_{TL}^{(k)} & \hat{A}_{TR} \\ \hline A_{BL}^{(k)} & \hat{A}_{BR} \end{array} \right)$. In other words, the first k columns have been updated and the rest of the columns are untouched.
- Only $\left(\frac{U_{TL}}{U_{BR}} \right)$, T_{TL}, and $\left(\frac{Z_{TR}}{Z_{BR}} \right)$ have been updated.

The question is how to advance the computation. Now, at the top of the loop, we expose

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array} \right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (\hat{A}, \hat{a}, \hat{\alpha}), (U, u, \upsilon), (Z, z, \zeta), (T, t, \tau)$. In order to compute the next Householder transformation, the next column of A must be updated according to prior computation:

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) = \left(I - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right) T_{00}^{-1} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right)^T \right)^T \left(\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) - \underbrace{\left(\begin{array}{c} Z_{00} \\ \hline z_{10}^T \\ \hline Z_{20} \end{array} \right) T_{00}^{-1} u_{10}}_{\text{column } k \text{ of } Z_k T_k^{-1} U_k^T} \right),$$

which means first updating

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) := \left(\begin{array}{c} a_{01} - Z_{00} w_{10} \\ \hline \alpha_{11} - z_{10}^T w_{10} \\ \hline a_{21} - Z_{20} w_{10} \end{array} \right),$$

where $w_{10} = T_{00}^{-1} u_{10}$. Next, we need to perform the update

$$\begin{aligned} \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) &:= \left(I - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right) T_{00}^{-1} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right)^T \right)^T \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) \\ &= \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) - \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right) T_{00}^{-T} \left(\begin{array}{c} U_{00} \\ \hline u_{10}^T \\ \hline U_{20} \end{array} \right)^T \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array} \right) = \left(\begin{array}{c} a_{01} - U_{00} y_{10} \\ \hline \alpha_{11} - u_{10}^T y_{10} \\ \hline a_{21} - U_{20} y_{10} \end{array} \right), \end{aligned}$$

where $y_{10} = T_{00}^{-T} (U_{00}^T a_{01} + u_{10} \alpha_{11} + U_{20}^T a_{21})$. After these computations we can compute the next Householder transform from a_{21}, updating a_{21}:

- $[u_{21}, \tau, a_{21}] := \text{HOUSEV}(a_{21})$.

The next column of Z is computed by

$$\left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right) := \left(\begin{array}{c|c|c} \hat{A}_{00} & \hat{a}_{01} & \hat{A}_{02} \\ \hline \hat{a}_{10}^T & \hat{\alpha}_{11} & \hat{a}_{12}^T \\ \hline \hat{A}_{20} & \hat{a}_{21} & \hat{A}_{22} \end{array}\right) \left(\begin{array}{c} 0 \\ \hline 0 \\ \hline u_{21} \end{array}\right) = \left(\begin{array}{c} \hat{A}_{02} u_{21} \\ \hline \hat{a}_{12}^T u_{21} \\ \hline \hat{A}_{22} u_{21} \end{array}\right).$$

As in Section 2.3, we finish by computing the next column of T:

$$\left(\begin{array}{c|c|c} T_{00} & \hat{t}_{01} & \hat{T}_{02} \\ \hline 0 & \hat{\tau}_{11} & \hat{t}_{12}^T \\ \hline 0 & 0 & \hat{T}_{22} \end{array}\right) := \left(\begin{array}{c|c|c} T_{00} & U_{20}^T u_{21} & \hat{T}_{02} \\ \hline 0 & \frac{1}{2} u_{21}^T u_{21} & \hat{t}_{12}^T \\ \hline 0 & 0 & \hat{T}_{22} \end{array}\right).$$

Note that $\frac{1}{2} u_{21}^T u_{21}$ is equal to the τ computed by $\text{Housev}(a_{21})$, and thus it need not be recomputed to update τ_{11}.

3.4 Blocked algorithms

We now discuss how much of the computation can be cast in terms of matrix-matrix multiplication. The first such blocked algorithm was reported in [12]. That algorithm corresponds roughly to our blocked Algorithm 1.

In Figure 4 we give four blocked algorithms which differ by how computation is accumulated in the body of the loop:

- Two correspond to using the unblocked algorithms in Figure 1.
- A third results from using the lazy algorithm in Figure 2. For this variant, we introduce matrices U, Y, and Z of width b in which vectors computed by the lazy unblocked algorithm are accumulated. We are not aware of this algorithm having been reported before.
- The fourth results from using the algorithm in Figure 3. It returns matrices U, Z, and T. It was first reported in [23] and we will call it the GQvdG blocked algorithm.

Let us consider having progressed through the matrix so that it is in the state

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \quad U = \left(\begin{array}{c} U_T \\ \hline U_B \end{array}\right), \quad Y = \left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right), \quad Z = \left(\begin{array}{c} Z_T \\ \hline Z_B \end{array}\right),$$

where A_{TL} is $b \times b$. Assume that the factorization has completed with A_{TL} and A_{BL} (meaning that A_{TL} is upper Hessenberg and A_{BL} is zero except for its top-right most element), and A_{TR} and A_{BR} have been updated so that only an upper Hessenberg factorization of A_{BR} has to be completed, updating the A_{TR} submatrix correspondingly. In the next iteration of the blocked algorithm, we perform the following steps:

- Perform the first b iterations of the lazy algorithm with matrix A_{BR}, accumulating the appropriate vectors in U_B, Y_B, and Z_B.
- Apply the resulting Householder transformations from the right to A_{TR}. In Section 2.3 we discussed that this requires the computation of $U^T U = S^T + D + S$, where D and S equal the diagonal and strictly upper triangular part of $U^T U$, after which $A_{TR} := A_{TR}(I - U T^{-1} U^T) = A_{TR} - A_{TR} U T^{-1} U^T$ with $T = \frac{1}{2} D + S$.
- Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} U_T \\ \hline U_B \end{array}\right) \rightarrow \left(\begin{array}{c} U_0 \\ \hline U_1 \\ \hline U_2 \end{array}\right), \quad \ldots$$

Algorithm: $[A] := \textsc{HessRed\_blk}(A, T)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $X \rightarrow \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right)$ **for** $X \in \{T, U, Y, Z\}$

  where A_{TL} is 0×0 and T_T, U_T, Y_T, and Z_T have 0 rows

while $m(A_{TL}) < m(A)$ **do**

  Determine block size b

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) \rightarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array}\right)$$

  for $X \in \{T, U, Y, Z\}$

    where A_{11} is $b \times b$ and T_1, U_1, Y_1, and Z_1 have b rows

---

  Algorithm 1, 2: (blocked + basic unblocked, blocked + rearranged unblocked)

  $[A_{BR}, U_B] := \textsc{HessRed\_unb}(b, A_{BR})$

  $T_1 = \frac{1}{2} D + S$ where $U_B^T U_B = S^T + D + S$

  $A_{TR} := A_{TR}(I - U_B T_1^{-1} U_B^T)$

  Algorithm 3: (blocked + lazy unblocked)

  $[A_{BR}, U_B, Y_B, Z_B] := \textsc{HessRed\_lazy\_unb}(b, A_{BR}, U_B, Y_B, Z_B)$

  $T_1 = \frac{1}{2} D + S$ where $U_B^T U_B = S^T + D + S$

  $A_{TR} := A_{TR}(I - U_B T_1^{-1} U_B^T)$

  $A_{22} := A_{22} - U_2 Y_2^T - Z_2 U_2^T$

  Algorithm 4: (GQvdG blocked + GQvdG unblocked)

  $[A_{BR}, U_B, Z_B, T_1] := \textsc{HessRed\_GQvdG\_unb}(b, A_{BR}, U_B, Z_B, T_1)$

  $A_{TR} := A_{TR}(I - U_B T_1^{-1} U_B^T)$

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(I - \left(\begin{array}{c} U_1 \\ \hline U_2 \end{array}\right) T_1^{-1} \left(\begin{array}{c} U_1 \\ \hline U_2 \end{array}\right)^T\right)^T \left(\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) - \left(\begin{array}{c} Z_1 \\ \hline Z_2 \end{array}\right) T_1^{-1} U_2^T\right)$$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) \leftarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array}\right)$$

  for $X \in \{T, U, Y, Z\}$

endwhile

Figure 4: Blocked reduction to Hessenberg form based on original or rearranged algorithm. The call to HessRed_unb performs the first b iterations of one of the unblocked algorithms in Figures 1 or 2. In the case of the algorithms in Figure 1, U_B accumulates and returns the vectors u_{21} encountered in the computation and Y_B and Z_B are not used.

- Update $A_{22} := A_{22} - U_2 Y_2^T - Z_2 U_2^T$.
- Move the thick line (which denotes how far the factorization has proceeded) forward by the block size:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} U_T \\ \hline U_B \end{array}\right) \leftarrow \left(\begin{array}{c} U_0 \\ \hline U_1 \\ \hline U_2 \end{array}\right), \quad \ldots$$

Proceeding like this block-by-block computes the reduction to upper Hessenberg form while reducing the size of the matrices U, Y, and Z, casting some of the computation in terms of matrix-matrix multiplications that are known to achieve high performance.

When one of the unblocked algorithms in Figure 1 is used instead, A_{22} is already updated upon return from HessRed_unb and thus only the update of A_{TR} can be accelerated by calls to level-3 BLAS operations.

The GQvdG blocked algorithm, which uses the GQvdG unblocked algorithm, was incorporated into recent releases of LAPACK, modulo a small change that accumulates T^{-1} instead of T. Prior to this, an algorithm that used the lazy unblocked algorithm but also updated A_{TR} as part of that unblocked algorithm (and thus cast less computation in terms of level-3 BLAS) was part of LAPACK [12]. A comparison between the GQvdG blocked algorithm and this previously used algorithm can be found in [23].

3.5 Fusing operations

We now discuss how the eligible sets of operations encountered in the various algorithms can be fused to reduce memory traffic.

In the rearranged algorithm, delaying the update of A_{22} yields the following three operations that can be fused (here we drop the subscripts):

$$\begin{array}{l} A := A - (uy^T + zu^T) \\ v := A^T x \\ w := Ax \end{array} \tag{4}$$

Note that the first operation may be implemented as two calls to the level-2 BLAS routine GER, while the remaining two operations are instances of the GEMV operation.

By inspecting the three operations, we notice that only one column of A needs to be read and updated at a time. So, let us partition

$$A \rightarrow \left(a_0 | \cdots | a_{n-1} \right), \quad u \rightarrow \begin{pmatrix} \upsilon_0 \\ \vdots \\ \upsilon_{n-1} \end{pmatrix}, \quad v \rightarrow \begin{pmatrix} \nu_0 \\ \vdots \\ \nu_{n-1} \end{pmatrix}, \quad x \rightarrow \begin{pmatrix} \chi_0 \\ \vdots \\ \chi_{n-1} \end{pmatrix}, \quad y \rightarrow \begin{pmatrix} \psi_0 \\ \vdots \\ \psi_{n-1} \end{pmatrix}.$$

Then the following steps, for $0 \leq j < n$, compute the desired result (provided that initially $w = 0$):

$$\begin{array}{ll} a_j := a_j - \psi_j u - \upsilon_j z & (2\times\text{AXPY}) \\ \nu_j := a_j^T x & (\text{DOT}) \\ w := w + \chi_j a_j & (\text{AXPY}) \end{array}$$

However, if we implement this fused operation by looping over the level-1 BLAS operations (parenthesized above), each element of A is still accessed six times—no fewer than if we had simply called GER and GEMV in sequence (twice each). We would only benefit (hopefully) from the current column of A, a_j, residing in the cache after the first call to AXPY, thus allowing the second AXPY, the DOT, and third AXPY routine invocations to more readily access the elements of a_j. We refer to this as "cache-level" fusing, as it promotes increased temporal locality of subparts of matrix A within the cache hierarchy and thus allows these memory-limited operations to complete in less time. The authors of [15] demonstrate the benefits of cache-level fusing, except they express the computation as a sequence of level-2 BLAS subproblems rather than in terms of level-1 operations.[4] But the purpose and effect is similar.

But on many architectures, accessing cached data—even data in the highest levels of the cache hierarchy—still incurs some cost. So ideally, we would want to avoid these redundant memory operations altogether. In order to do this, we need to further partition the level-1 subproblems to allow fusing of individual scalar arithmetic operations.

If we coded the operations at a very low level, controlling individual load and store instructions, we could implement the algorithm in Figure 5 (right). We consider this algorithm to be fused at the register-level because certain memory operations are avoided by reusing data when they are still loaded in the processor core's registers. We provide an unfused algorithm on the left-hand side of the figure and a cache-level fusing in the middle for contrast. Note that the cache-level algorithm fuses only the outer loops (over n) while the register-level algorithm goes a step further and also fuses the inner loops (over m). It is easy to see that register-level fusing reduces the number of memory accesses to each element of matrix A to the absolute minimum: one load and one store.

The other fusable operations present in Figures 1 and 2, and throughout the remainder of this paper, can be fused in a similar manner.

[4] The blocking employed by authors' cache-level technique uses the same algorithmic blocksize specified in the top-level blocked algorithm.

```
for j = 0 : n-1
    LOAD y_j → β
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD u_i → υ_1
        α_11 := α_11 - β υ_1
        STORE A_ij ← α_11
    endfor
endfor
for j = 0 : n-1
    LOAD u_j → γ
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD z_i → ζ_1
        α_11 := α_11 - γ ζ_1
        STORE A_ij ← α_11
    endfor
endfor
for j = 0 : n-1
    ρ := 0
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD x_i → χ_1
        ρ := ρ + α_11 χ_1
    endfor
    STORE ν_j ← ρ
endfor
SETTOZERO( w )
for j = 0 : n-1
    LOAD x_j → κ
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD w_i → ω_1
        ω_1 := ω_1 + κ α_11
        STORE w_i ← ω_1
    endfor
endfor
```

```
SETTOZERO( w )
for j = 0 : n-1
    LOAD y_j → β
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD u_i → υ_1
        α_11 := α_11 - β υ_1
        STORE A_ij ← α_11
    endfor
    LOAD u_j → γ
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD z_i → ζ_1
        α_11 := α_11 - γ ζ_1
        STORE A_ij ← α_11
    endfor
    ρ := 0
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD x_i → χ_1
        ρ := ρ + α_11 χ_1
    endfor
    STORE ν_j ← ρ
    LOAD x_j → κ
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD w_i → ω_1
        ω_1 := ω_1 + κ α_11
        STORE w_i ← ω_1
    endfor
endfor
```

```
SETTOZERO( w )
for j = 0 : n-1
    LOAD y_j → β
    LOAD u_j → γ
    LOAD x_j → κ
    ρ := 0
    for i = 0 : m-1
        LOAD A_ij → α_11
        LOAD u_i → υ_1
        LOAD z_i → ζ_1
        LOAD x_i → χ_1
        LOAD w_i → ω_1
        α_11 := α_11 - β υ_1
        α_11 := α_11 - γ ζ_1
        ρ := ρ + α_11 χ_1
        ω_1 := ω_1 + κ α_11
        STORE A_ij ← α_11
        STORE w_i ← ω_1
    endfor
    STORE ν_j ← ρ
endfor
```

Figure 5: Algorithms for computing the fusable set of operations present in Eq. 4 using no fusing (left), cache-level fusing (middle), and register-level fusing (right). Whereas the unfused and cache-level fused algorithms access each element of matrix A six times, the register-level fused algorithm avoids redundant memory instructions and thus touches each element only twice.

4 Reduction to tridiagonal form

The first step towards computing the eigenvalue decomposition of a symmetric matrix is to reduce the matrix to tridiagonal form.

Let $A \in \mathbb{R}^{n \times n}$ be symmetric. If $A \rightarrow QBQ^T$ where B is upper Hessenberg and Q is orthogonal, then B is symmetric and therefore tridiagonal. In this section we show how to take advantage of symmetry, assuming that matrix A is stored in only the lower triangular part of A and only the lower triangular part of that matrix is overwritten with B.

When matrix A is symmetric, and only the lower triangular part is stored and updated, the unblocked

Algorithm: $[A] := \text{TriRed\_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

for $x \in \{u, y\}$

  where A_{TL} is 0×0 and u_T, y_T have 0 rows

while $m(A_{TL}) < m(A)$ **do**

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

  for $(x, \chi) \in \{(u, \upsilon), (y, \psi)\}$

    where α_{11}, υ_{11}, and ψ_{11} are scalars

Basic unblocked:	Rearranged unblocked:	
	$\alpha_{11} := \alpha_{11} - 2\upsilon_{11}\psi_{11}$	$(\star)$
	$a_{21} := a_{21} - (u_{21}\psi_{11} + y_{21}\upsilon_{11})$	$(\star)$
$[u_{21}, \tau, a_{21}] := \text{Housev}(a_{21})$	$[x_{21}, \tau, a_{21}] := \text{Housev}(a_{21})$	
	$A_{22} := A_{22} - u_{21}y_{21}^T - y_{21}u_{21}^T$	$(\star)$
$y_{21} := A_{22}u_{21}$	$v_{21} := A_{22}x_{21}$	
	$u_{21} := x_{21}$; $y_{21} := v_{21}$	
$\beta := u_{21}^T y_{21}/2$	$\beta := u_{21}^T y_{21}/2$	
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$	$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$	
$A_{22} := A_{22} - u_{21}y_{21}^T - y_{21}u_{21}^T$		

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

  for $(x, \chi) \in \{(u, \upsilon), (y, \psi)\}$

endwhile

Figure 6: Unblocked algorithms for reduction to tridiagonal form. Left: basic algorithm. Right: rearranged to allow fusing of operations. Operations marked with $(\star)$ are not executed during the first iteration.

algorithms for reducing A to upper Hessenberg form can be changed by noting that $v_{21} = w_{21}$ and $y_{21} = z_{21}$. This motivates the algorithms in Figures 6–8, which correspond respectively to Figures 1 (left and right), 2, and 4 when taking advantage of symmetry. The blocked algorithm and associated unblocked algorithm was first reported in [12].

In the rearranged algorithm, delaying the update of A_{22} allows the highlighted operations in Figure 6 (right) to be fused. We leave it as an exercise to the reader to fuse the highlighted operations in Figure 7.

5 Reduction to bidiagonal form

The previous sections were inspired by the paper [15] that discusses how fused operations can benefit algorithms for the reduction of a matrix to bidiagonal form. The purpose of this section is to present the basic and rearranged unblocked algorithms for this operation with our notation to facilitate the comparing and contrasting of the reduction to upper Hessenberg and tridiagonal form algorithms to those for the reduction to bidiagonal form.

The first step towards computing the Singular Value Decomposition (SVD) of $A \in \mathbb{R}^{m \times n}$ is to reduce the matrix to bidiagonal form: $A \rightarrow Q_L B Q_R^T$ where B is a bidiagonal matrix (nonzero diagonal and superdiagonal) and Q_L and Q_R are again square and orthogonal.

Algorithm: $[A, U, Y] := \textsc{TriRed\_lazy\_unb}(b, A, U, Y)$

Partition $X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$

for $X \in \{A, U, Y\}$

  where X_{TL} is 0×0

while $n(U_{TL}) < b$ **do**

  Repartition

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

  for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (Y, y, \psi)\}$

    where χ_{11} is a scalar

$\alpha_{11} := \alpha_{11} - u_{10}^T y_{10} - y_{10}^T u_{10}$
$a_{21} := a_{21} - U_{20} y_{10} - Y_{20} u_{10}$
$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$
$y_{21} := A_{22} u_{21}$
$y_{21} := y_{21} - Y_{20}(U_{20}^T u_{21}) - U_{20}(Y_{20}^T u_{21})$
$\beta := u_{21}^T y_{21}/2$
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$

Continue with

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

  for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (Y, y, \psi)\}$

endwhile

Figure 7: Lazy unblocked reduction to tridiagonal form.

For simplicity, we explain the algorithms for the case where A is square.

5.1 Basic algorithm

The basic algorithm for this operation, overwriting A with the result B, can be explained as follows:

- Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
- Let $\left[\left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ \hline 0 \end{array}\right)\right] := \textsc{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)\right)$.[5]
- Update

$$\begin{array}{rcl} \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) & := & \left(I - \left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right)\left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right)^T / \tau_L\right)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \\ & = & \left(\begin{array}{c|c} \alpha - \psi_{11}/\tau_L & a_{12}^T - y_{21}^T/\tau_L \\ \hline 0 & A_{22} - u_{21} y_{21}^T/\tau_L \end{array}\right), \end{array}$$

  where $\psi_{11} = \alpha_{11} + u_{21}^T a_{21}$ and $y_{21}^T = a_{12}^T + u_{21}^T A_{22}$. Note that $\alpha_{11} := \alpha - \psi_{11}/\tau_L$ does not need to be executed since this update was performed by the instance of Housev above.

- Let $[v_{21}, \tau_R, a_{12}] := \textsc{Housev}\,(a_{12})$.

[5]Note that the semantics here indicate that α_{11} is overwritten by the first element of $\left(\begin{array}{c} \alpha_{11} \\ \hline 0 \end{array}\right)$.

Algorithm: $[A, U, Y] := \text{TriRed\_blk}(A, U, Y)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $X \rightarrow \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right)$
for $X \in \{U, Y\}$
  where A_{TL} is 0×0 and U_T, Y_T have 0 rows
while $m(A_{TL}) < m(A)$ **do**
  Determine block size b
  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) \rightarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array}\right)$$

  for $X \in \{U, Y\}$
   where A_{11} is $b \times b$ and U_1, and Y_1 have b rows

  $[A_{BR}, U_B, Y_B] := \text{TriRed\_lazy\_unb}(b, A_{BR}, U_B, Y_B)$
  $A_{22} := A_{22} - U_2 Y_2^T - Y_2 U_2^T$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array}\right) \leftarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array}\right)$$

  for $X \in \{U, Y\}$
endwhile

Figure 8: Blocked reduction to tridiagonal form based on original or rearranged algorithm. TriRed_unb performs the first b iterations of the lazy unblocked algorithm in Figure 7.

- Update $A_{22} := A_{22}(I - v_{21}v_{21}^T/\tau_R) = A_{22} - z_{21}v_{21}^T/\tau_R$, where $z_{21} = A_{22}v_{21}$.
- Continue this process with the updated A_{22}.

The resulting algorithm, slightly rearranged, is given in Figure 9 (left).

5.2 Rearranged algorithm

We now show how, again, the loop can be restructured so that multiple updates of, and multiplications with, A_{22} can be fused. Focus on the update $A_{22} := A_{22} - (u_{21}y_{21}^T + z_{21}v_{21}^T)$. Partition

$$A_{22} \rightarrow \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right), \quad u_{21} \rightarrow \left(\begin{array}{c} \upsilon_{11}^+ \\ \hline u_{21}^+ \end{array}\right), \quad y_{21} \rightarrow \left(\begin{array}{c} \psi_{11}^+ \\ \hline y_{21}^+ \end{array}\right), \quad z_{21} \rightarrow \left(\begin{array}{c} \zeta_{11}^+ \\ \hline z_{21}^+ \end{array}\right), \quad v_{21} \rightarrow \left(\begin{array}{c} \nu_{11}^+ \\ \hline v_{21}^+ \end{array}\right),$$

where $+$ indicates the partitioning in the next iteration. Then

$$\begin{aligned} \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right) &:= \left(\begin{array}{c|c} \alpha_{11}^+ & a_{12}^{+T} \\ \hline a_{21}^+ & A_{22}^+ \end{array}\right) - \left(\begin{array}{c} \upsilon_{11}^+ \\ \hline u_{21}^+ \end{array}\right)\left(\begin{array}{c} \psi_{11}^+ \\ \hline y_{21}^+ \end{array}\right)^T - \left(\begin{array}{c} \zeta_{11}^+ \\ \hline z_{21}^+ \end{array}\right)\left(\begin{array}{c} \nu_{11}^+ \\ \hline v_{21}^+ \end{array}\right)^T \\ &= \left(\begin{array}{c|c} \alpha_{11}^+ - \upsilon_{11}^+\psi_{11}^+ - \zeta_{11}^+\nu_{11}^+ & a_{12}^{+T} - \upsilon_{11}^+ y_{21}^{+T} - \zeta_{11}^+ v_{21}^{+T} \\ \hline a_{21}^+ - u_{21}^+\psi_{11}^+ - z_{21}^+\nu_{11}^+ & A_{22}^+ - u_{21}^+ y_{21}^{+T} - z_{21}^+ v_{21}^{+T} \end{array}\right), \end{aligned}$$

Algorithm: $[A] := \textsc{BiRed\_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

for $x \in \{u, v, y, z\}$

  where A_{TL} is 0×0, u_T, v_T, y_T, z_T have 0 elements

while $m(A_{TL}) < m(A)$ **do**

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

  for $(x, \chi) \in \{(u, \upsilon), (v, \nu), (y, \psi), (z, \zeta)\}$

   where α_{11}, υ_{11}, ν_{11}, ψ_{11}, and ζ_{11} are scalars

---

Basic unblocked:

$$\left[\left(\begin{array}{c} 1 \\ u_{21} \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ 0 \end{array}\right)\right] := \textsc{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ a_{21} \end{array}\right)\right)$$

$y_{21} := a_{12} + A_{22}^T u_{21}$
$a_{12}^T := a_{12}^T - y_{21}^T / \tau_L$

$[v_{21}, \tau_R, a_{12}] := \textsc{Housev}\,(a_{12})$

$\beta := y_{21}^T v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (A_{22} v_{21} - \beta u_{21} / \tau_L) / \tau_R$

$A_{22} := A_{22} - u_{21} y_{21}^T - z_{21} v_{21}^T$

Rearranged unblocked:

$\alpha_{11} := \alpha_{11} - \upsilon_{11}\psi_{11} - \zeta_{11}\nu_{11}$ ($\star$)
$a_{21} := a_{21} - u_{21}\psi_{11} - z_{21}\nu_{11}$ ($\star$)
$a_{12}^T := a_{12}^T - \upsilon_{11} y_{21}^T - \zeta_{11} v_{21}^T$ ($\star$)

$$\left[\left(\begin{array}{c} 1 \\ u_{21}^+ \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ 0 \end{array}\right)\right] := \textsc{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ a_{21} \end{array}\right)\right)$$

$a_{12}^+ := a_{12} - a_{12} / \tau_L$
$A_{22} := A_{22} - u_{21} y_{21}^T - z_{21} v_{21}^T$ ($\star$)
$y_{21} := A_{22}^T u_{21}^+$
$a_{12}^+ := a_{12}^+ - y_{21} / \tau_L$
$w_{21} := A_{22} a_{12}^+$
$y_{21} := y_{21} + a_{12}$
$[\psi_{11} - \alpha_{12}, \tau_R, \alpha_{12}] := \textsc{Houses}(a_{12}^+)$
$v_{21} := (a_{12}^+ - \alpha_{12} e_0) / (\psi_{11} - \alpha_{12})$
$a_{12}^T := \alpha_{12} e_0^T$
$u_{21} := u_{21}^+$
$\beta := y_{21}^T v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (w_{21} - \alpha_{12} A_{22} e_0) / (\psi_{11} - \alpha_{12})$
$z_{21} := z_{21} - \beta u_{21} / \tau_L$
$z_{21} := z_{21} / \tau_R$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

  for $(x, \chi) \in \{(u, \upsilon), (v, \nu), (y, \psi), (z, \zeta)\}$

endwhile

Figure 9: Unblocked algorithms for reduction to bidiagonal form. Left: basic algorithm. Right: rearranged to allow fusing of operations (this is essentially Algorithm I from [15]). The fused operation in the "Basic unblocked" algorithm corresponds to the BLAS 2.5 operation GER2 while the fused operation in the "Rearranged unblocked" algorithm corresponds to GEMVER [8]. Operations marked with ($\star$) are not executed during the first iteration.

which shows how the update of A_{22} can be delayed until the next iteration. If $u_{21} = y_{21} = z_{21} = v_{21} = 0$ during the first iteration, the body of the loop may be changed to

$$\begin{array}{l}
\alpha_{11} := \alpha_{11} - \upsilon_{11}\psi_{11} - \zeta_{11}\nu_{11} \\
a_{21} := a_{21} - u_{21}\psi_{11} - z_{21}\nu_{11} \\
a_{12}^T := a_{12}^T - \upsilon_{11}y_{21}^T - \zeta_{11}v_{21}^T \\
\left[\left(\frac{1}{u_{21}^+}\right), \tau_L, \left(\frac{\alpha_{11}}{0}\right)\right] := \text{HOUSEV}\left(\left(\frac{\alpha_{11}}{a_{21}}\right)\right) \\
A_{22} := A_{22} - u_{21}y_{21}^T - z_{21}v_{21}^T \\
y_{21} := a_{12} + A_{22}^T u_{21}^+ \\
a_{12}^T := a_{12}^T - y_{21}^T/\tau_L \\
[v_{21}, \tau_R, a_{12}] := \text{HOUSEV}\,(a_{12}) \\
\beta := y_{21}^T v_{21} \\
y_{21} := y_{21}/\tau_L \\
z_{21} := (A_{22}v_{21} - \beta u_{21}^+/\tau_L)/\tau_R
\end{array}$$

Now, the goal becomes to bring the three highlighted updates together. The problem is that the last update, which requires v_{21}, cannot commence until after the second call to HOUSEV completes. This dependency can be circumvented by observing that one can perform a matrix-vector multiply of A_{22} with the vector $a_{12}^T = a_{12}^T - y_{21}^T/\tau_L$ instead of with v_{21}, after which the result can be updated as if the multiplication had used the output of the HOUSEV, as indicated by Eq. (3) in Section 2. These observations justify the rearrangement of the computations as indicated in Figure 9 (right).

5.3 Lazy algorithms

A lazy algorithm can be derived by not updating A_{22} at all, and instead accumulating the updates in matrix U, V, Y, and Z, much like was done for the other reduction to condensed form operations.

We start with the rearranged algorithm to make sure that

$$\begin{array}{l}
y_{21} := A_{22}^T u_{21}^+ \\
a_{12}^+ := a_{12}^+ - y_{21}/\tau_L \\
w_{21} := A_{22}a_{12}^+
\end{array}$$

can still be fused. Next, the key is to realize that what was previously a multiplication by A_{22} must now be replaced by a multiplication by $A_{22} - U_{20}Y_{20}^T - Z_{20}V_{20}^T$. This yields the algorithm in Figure 10 (right) which was first proposed by Howell et al. [15].

For completeness, we include in Figure 10 (left) a basic algorithm which does not rearrange operations for fusing, but still has the "lazy" property whereby A_{22} is never updated.

5.4 Blocked algorithms

Finally, a blocked algorithm is given in Figure 11. The basic lazy unblocked algorithm in conjunction with the blocked algorithm was first published in [12] and is part of LAPACK. The rearranged lazy unblocked algorithm in conjunction with the blocked algorithm was proposed as Algorithm III in [15].

5.5 Fusing operations

Once again, we leave it as an exercise to the reader to construct loop-based fusings of the operations highlighted in Figures 9 and 10.

6 Accumulating Householder transformations

In Section 2.3, we briefly discussed how to accumulate the triangular factors T of the block Householder transformations. The need for computing and storing T is clear in the unblocked and blocked GQvdG

Algorithm: $[A, U, V, Y, Z] := \textsc{BiRed\_lazy\_unb}(b, A, U, V, Y, Z)$

Partition $X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$
for $X \in \{A, U, V, Y, Z\}$
where X_{TL} is 0×0
while $n(U_{TL}) < b$ **do**
Repartition

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (V, v, \nu), (Y, y, \psi), (Z, z, \zeta)\}$
where χ_{11} is a scalar

Lazy basic unblocked:

$\alpha_{11} := \alpha_{11} - u_{10}^T y_{10} - z_{10}^T v_{10}$
$a_{21} := a_{21} - U_{20} y_{10} - Z_{20} v_{10}$
$a_{12}^T := a_{12}^T - u_{10}^T Y_{20}^T - z_{10}^T V_{20}^T$
$\left[\left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ \hline 0 \end{array}\right)\right] := \textsc{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)\right)$
$y_{21} := a_{12} + A_{22}^T u_{21} - Y_{20} U_{20}^T u_{21} - V_{20} Z_{20}^T u_{21}$
$a_{12}^T := a_{12}^T - y_{21}^T / \tau_L$
$[v_{21}, \tau_R, a_{12}] := \textsc{Housev}(a_{12})$
$\beta := y_{21}^T v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (A_{22} v_{21} - U_{20} Y_{20}^T v_{21} - Z_{20} V_{20}^T v_{21} - \beta u_{21} / \tau_L) / \tau_R$

Lazy rearranged unblocked:

$\alpha_{11} := \alpha_{11} - u_{10}^T y_{10} - z_{10}^T v_{10}$
$a_{21} := a_{21} - U_{20} y_{10} - Z_{20} v_{10}$
$a_{12}^T := a_{12}^T - u_{10}^T Y_{20}^T - z_{10}^T V_{20}^T$
$\left[\left(\begin{array}{c} 1 \\ \hline u_{21}^+ \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ \hline 0 \end{array}\right)\right] := \textsc{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)\right)$
$a_{12}^+ := a_{12} - a_{12} / \tau_L$
$y_{21} := -Y_{20} U_{20}^T u_{21}^+ - V_{20} Z_{20}^T u_{21}^+$
$y_{21} := y_{21} + A_{22}^T u_{21}^+$
$a_{12}^+ := a_{12}^+ - y_{21} / \tau_L$
$w_{21} := A_{22} a_{12}^+$
$w_{21} := w_{21} - U_{20} Y_{20}^T a_{12}^+ - Z_{20} V_{20}^T a_{12}^+$
$a_{22l} := A_{22} e_0 - U_{20} Y_{20}^T e_0 - Z_{20} V_{20}^T e_0$
$y_{21} := a_{12} + y_{21}$
$[\psi_{11} - \alpha_{12}, \tau_R, \alpha_{12}] := \textsc{Houses}(a_{12}^+)$
$v_{21} := (a_{12}^+ - \alpha_{12} e_0) / (\psi_{11} - \alpha_{12})$;
$a_{12}^T := \alpha_{12} e_0^T$
$u_{21} := u_{21}^+$
$\beta := y_{21}^T v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (w_{21} - \alpha_{12} a_{22l}) / (\psi_{11} - \alpha_{12})$
$z_{21} := z_{21} - \beta u_{21} / \tau_L$
$z_{21} := z_{21} / \tau_R$

Continue with

$$\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & x_{01} & X_{02} \\ \hline x_{10}^T & \chi_{11} & x_{12}^T \\ \hline X_{20} & x_{21} & X_{22} \end{array}\right)$$

for $(X, x, \chi) \in \{(A, a, \alpha), (U, u, \upsilon), (V, v, \nu), (Y, y, \psi), (Z, z, \zeta)\}$
endwhile

Figure 10: Lazy unblocked versions of the algorithms in Figure 9. Left: lazy basic algorithm. Right: lazy rearranged algorithm (this is essentially Algorithm III from [15]). The first fused operation in the "Lazy rearranged unblocked" algorithm, modulo a slight reordering of the computation vis-à-vis y_{21}, corresponds to the BLAS 2.5 operation GEMVER [8]. Note that upon entry to both algorithms, matrix A is $n \times n$ and matrices U, V, Y, and Z are $n \times b$. Also note that the multiplications $A_{22} e_0$, $Y_{20}^T e_0$, and $U_{20}^T e_0$ do not require computation: they simply extract the first column or row of the given matrix.

Algorithm: $[A] := \textsc{BiRed\_blk}(A, U, V, Y, Z)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right)$, $X \rightarrow \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right)$

for $X \in \{U, V, Y, Z\}$

  where A_{TL} is 0×0 and U_T, V_T, Y_T, Z_T have 0 rows

while $m(A_{TL}) < m(A)$ **do**

  Determine block size b

  Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) \rightarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array} \right)$$

  for $X \in \{U, V, Y, Z\}$

   where A_{11} is $b \times b$ and U_1, V_1, Y_1, and Z_1 have b rows

  $[A_{BR}, U_B, V_B, Y_B, Z_B] := \textsc{BiRed\_lazy\_unb}(b, A_{BR}, U_B, V_B, Y_B, Z_B)$

  $A_{22} := A_{22} - U_2 Y_2^T - Z_2 V_2^T$

  Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right), \left(\begin{array}{c} X_T \\ \hline X_B \end{array} \right) \leftarrow \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array} \right)$$

  for $X \in \{U, V, Y, Z\}$

endwhile

Figure 11: Blocked algorithm for reduction to bidiagonal form. For simplicity, it is assumed that A is $n \times n$ where n is an integer multiple of b. Matrices U, V, Y, and Z are all $n \times b$.

algorithms for reducing a matrix to upper Hessenberg form, shown in Figures 3 and 4. However, none of the other algorithms (blocked or unblocked) for reduction to condensed form use the triangular factors, because none of the other algorithms apply block Householder transforms. So at first glance, computing and storing T within these algorithms may seem unnecessary.

But typically reduction to condensed form is not a terminal operation. The triangular factors will be needed when forming (or applying) the orthogonal matrix Q after a reduction to upper Hessenberg or tridiagonal form, or the matrices Q_L and Q_R subsequent to a reduction to bidiagonal form. So for most applications, it is not a matter of if these factors will be computed, but when.

Note that we would normally compute T by columns, via $t_{01} := U_{20}^T u_{21}$, as shown in the GQvdG algorithm in Figure 3, and the scalar τ_{11} is computed as part of the $\textsc{Housev}$ function. Upon careful inspection, we find that each lazy unblocked algorithm (shown in Figures 2, 7, and 10) computes $U_{20}^T u_{21}$ as an intermediate product in the course of its normal computation. Indeed, for reduction to upper Hessenberg and tridiagonal forms, this intermediate product is computed within fusable sets of operations. And for reduction to bidiagonal form, if the intermediate product $V_{20}^T a_{12}^+$ is saved from the second set of fusable operations (see Figure 10), then the t_{01} vectors associated with the right-hand orthogonal matrix Q_R may easily be computed in a manner similar to that used to compute v_{21}. This technique saves approximately $\frac{1}{4}bn^2$ floating-point operations every time Q (or Q_L or Q_R) is formed or applied. Thus, given that the triangular factors can fit within a relatively small $b \times n$ matrix (or two such matrices for bidiagonal reduction), it is easy to make the case that these values should be stored for later use.

Notwithstanding the obvious advantage to storing T within the lazy unblocked algorithms, we have chosen to omit these statements from the algorithm figures (except in the case of the GQvdG algorithm) since they relate more to subsequent computations outside the scope of our discussion than the reduction to condensed form operations themselves.

Fused operation	BLAST name	dependent algorithms	flops	memory operations	
				unfused	fused
$v := A^T x$ $w := Ax$	GEMVT	Hessenberg	$4n^2$	$2n^2$	n^2
$A := A - a^T b - c^T d$	GER2	Hessenberg, bidiagonal	$4n^2$, $4mn$	$4n^2$, $4mn$	$2n^2$, $2mn$
$A := A - a^T b - c^T d$ $v := A^T x$ $w := Ax$	N/A	Hessenberg	$8n^2$	$6n^2$	$2n^2$
$y := y - YU^T u - UZ^T u$ $z := z - UY^T u - ZU^T u$	N/A	Hessenberg	$14mn$	$7mn$	$5mn$
$A := A - u^T y - y^T u$ $v := Ax$	N/A	tridiagonal	$4n^2$	$5n^2$	$2n^2$
$y := y - YU^T u - UY^T u$	N/A	tridiagonal	$8mn$	$4mn$	$3mn$
$A := A - a^T b - c^T d$ $b := A^T u$ $a := a + \beta b$ $w := Aa$	GEMVER	bidiagonal	$8mn$	$6mn$	$2mn$
$b := b + \alpha A^T u$ $a := a + \beta b$ $w := Aa$	GEMVT	bidiagonal	$4mn$	$2mn$	mn
$w := w - UY^T a - ZV^T a$ $t := Ae_0 - UY^T e_0 - ZV^T e_0$	N/A	bidiagonal	$6mn$	$6mn$	$4mn$

Figure 12: A summary of the fused operations one could potentially use within various reduction to condensed form algorithms and their floating-point and memory operation costs. The highlighted sets of fused operations are those present in the algorithms which exhibited the highest performance.

7 Estimating the impact of fusing

Before presenting performance results of actual implementations, we will first estimate the impact of fusing on performance.

The table in Figure 12 summarizes all of the fused operations used by all algorithms presented in this paper and lists the corresponding routine names given by the BLAST Forum [8]. The table also includes the approximations for the floating-point and memory operation counts, which may be used to derive the total number of memory and floating-point operations incurred within a given unfused or fused unblocked algorithm implementation. These totals are summarized in Figure 13. Similarly, the table in Figure 14 shows the number of floating-point operations (flops) required by unblocked and blocked components of various algorithms. The table also quantifies the number of flops executed by fusable sets of operations within a given unblocked algorithm.

Combining the analyses summarized in Figure 13 and Figure 14 allows us to estimate an upper bound for the asymptotic speedup one would observe from fusing operations within a given algorithm. We need only make a few mild assumptions concerning the computation to construct a model to predict actual performance improvement:

- The level-3 computation in a blocked algorithm executes s times faster than the level-1 and level-2 computation in the corresponding unblocked algorithm.[6]

[6]Note that this assumption typically does not hold for small problem sizes due to data caching, which is why we only attempt

Algorithm (unblocked only)	memory operations		
	unfused	fused	$r = \frac{\text{unfused}-\text{fused}}{\text{unfused}}$
Reduction to upper Hessenberg form			
Basic 2	$2n^3+\frac{1}{2}bn^2$	$n^3+\frac{1}{2}bn^2$	$\approx 50\%$
Rearranged	$2n^3+\frac{1}{2}bn^2$	$\frac{2}{3}n^3+\frac{1}{2}bn^2$	$\approx 66\%$
Lazy	$\frac{2}{3}n^3+\frac{15}{4}bn^2$	$\frac{1}{3}n^3+\frac{13}{4}bn^2$	$\approx 50\%$
Reduction to tridiagonal form			
Rearranged	$\frac{1}{2}n^3$	$\frac{1}{3}n^3$	$\approx 33\%$
Lazy	$\frac{1}{6}n^3+\frac{3}{2}bn^2$	$\frac{1}{6}n^3+\frac{5}{4}bn^2$	$\approx 1\%$
Reduction to bidiagonal form			
Basic	$3(mn^2-\frac{1}{3}n^3)$	$2(mn^2-\frac{1}{3}n^3)$	$\approx 33\%$
Rearranged	$3(mn^2-\frac{1}{3}n^3)$	$(mn^2-\frac{1}{3}n^3)$	$\approx 66\%$
Lazy rearranged	$(mn^2-\frac{1}{3}n^3)+$ $4b(mn-\frac{1}{2}n^2)$	$\frac{1}{2}(mn^2-\frac{1}{3}n^3)+$ $3bmn$	$\approx 50\%$
Howell's Algorithm III	$(mn^2-\frac{1}{3}n^3)+$ $4b(mn-\frac{1}{2}n^2)$	$(mn^2-\frac{1}{3}n^3)+$ $4b(mn-\frac{1}{2}n^2)$	0%[7]

Figure 13: A summary of the number of memory operations required by unfused and fused implementations of various unblocked algorithms for reducing a matrix to condensed form.

- An unblocked algorithm's execution is limited by memory accesses rather than its floating-point operations. This allows us to assume that reducing a fraction n of memory operations within an unblocked algorithm will result in the a corresponding speedup of $\frac{1}{1-n}$, or a $\frac{1}{1-n}$ speedup contribution to the overall algorithm if it is part of a blocked algorithm.

Thus, the expected asymptotic speedup α due to fusing is given by

$$\alpha = \frac{\text{Execution time without fusing}}{\text{Execution time with fusing}} = \frac{t^{\text{unfused}}_{\text{unblocked}} + t_{\text{blocked}}}{t^{\text{fused}}_{\text{unblocked}} + t_{\text{blocked}}} = \frac{su + (1-u)}{su(1-rf) + (1-u)}$$

where r is the fraction of unblocked memory operations that are avoided via fusing, f is the fraction of unblocked floating-point computation that is associated with fusable operations, and u is the fraction of total floating-point operations performed within the unblocked algorithm. Note that approximations for r are given in the right-hand column of Figure 13 while f and u are estimated in the two right-most columns of Figure 14.

Figure 15 summarizes the expected asymptotic speedups due to fusing for all condensed form algorithms that contain fusable sets of operations.

The most obvious takeaway from Figures 13–15 is that while reduction to upper Hessenberg form and reduction to bidiagonal form appear well-suited for speedup, reduction to tridiagonal form presents fewer opportunities for fusing. In fact, the blocked lazy algorithm is only benefited through a lower-order term. Thus, we would not expect to see much improvement, if any, for this particular algorithm.

8 Performance results

We now report performance for implementations of various algorithms that is attained in practice.

to estimate performance improvement for relatively large problem sizes.

[7] Howell et al. implement fused operations as a sequence of level-2 BLAS operations. Rather than achieving speedup by reducing memory operations, this type of fusing uses blocking to interleave smaller fusable subproblems in an effort to promote increased data cache reuse.

Algorithm	floating-point operations				
	unblocked	fusable	blocked	$f = \frac{\text{fusable}}{\text{unblocked}}$	$u = \frac{\text{unblocked}}{\text{total}}$
Reduction to upper Hessenberg form					
Basic 2	$\frac{8}{3}n^3 + bn^2$	$\frac{8}{3}n^3$	$\frac{2}{3}n^3$	$\approx 99\%$	$\approx 80\%$
Rearranged	$\frac{4}{3}n^3 + \frac{15}{2}bn^2$	$\frac{4}{3}n^3 + \frac{7}{2}bn^2$	$2n^3$	$\approx 99\%$	$\approx 80\%$
Lazy	$\frac{4}{3}n^3 + \frac{15}{2}bn^2$	$\frac{4}{3}n^3 + \frac{7}{2}bn^2$	$2n^3$	$\approx 99\%$	$\approx 40\%$
Reduction to tridiagonal form					
Rearranged	$\frac{4}{3}n^3$	$\frac{4}{3}n^3$	N/A	$\approx 100\%$	$\approx 100\%$
Lazy	$\frac{2}{3}n^3 + 3bn^2$	$2bn^2$	$\frac{2}{3}n^3$	$\approx 1\%$	$\approx 51\%$
Reduction to bidiagonal form					
Basic	$4(mn^2 - \frac{1}{3}n^3)$	$4(mn^2 - \frac{1}{3}n^3)$	N/A	$\approx 100\%$	$\approx 100\%$
Rearranged	$4(mn^2 - \frac{1}{3}n^3)$	$4(mn^2 - \frac{1}{3}n^3)$	N/A	$\approx 100\%$	$\approx 100\%$
Lazy rearranged	$2(mn^2 - \frac{1}{3}n^3) + 8b(mn - n^2)$	$2(mn^2 - \frac{1}{3}n^3) + 4b(mn - n^2)$	$2(mn^2 - \frac{1}{3}n^3)$	$\approx 99\%$	$\approx 51\%$

Figure 14: A summary of the number of floating-point operations required by various algorithms for reducing a matrix to condensed form. The two right-most columns, combined with the right-hand column in Figure 13, may be used to estimate upper bounds for the speedup one would observe from fusing eligible subproblems within an operation's unblocked algorithm. These upper bounds are estimated in Figure 15.

Algorithm	memory operations	floating-point operations		speedup α	
	$r = \frac{\text{unfused}-\text{fused}}{\text{unfused}}$	$f = \frac{\text{fusable}}{\text{unblocked}}$	$u = \frac{\text{unblocked}}{\text{total}}$	$s = 4$	$s = 5$
Reduction to upper Hessenberg form					
Basic 2	$\approx 50\%$	$\approx 99\%$	$\approx 80\%$	1.87	1.89
Rearranged	$\approx 66\%$	$\approx 99\%$	$\approx 80\%$	2.60	2.65
Lazy	$\approx 50\%$	$\approx 99\%$	$\approx 40\%$	1.56	1.61
Reduction to tridiagonal form					
Rearranged	$\approx 33\%$	$\approx 100\%$	$\approx 100\%$	1.49	1.49
Lazy	$\approx 1\%$	$\approx 1\%$	$\approx 51\%$	1.00	1.00
Reduction to bidiagonal form					
Basic	$\approx 33\%$	$\approx 100\%$	$\approx 100\%$	1.49	1.49
Rearranged	$\approx 66\%$	$\approx 100\%$	$\approx 100\%$	2.94	2.94
Lazy rearranged	$\approx 50\%$	$\approx 99\%$	$\approx 51\%$	1.66	1.71

Figure 15: Estimated asymptotic speedup from fusing using a simple model that assumes: (1) that the level-3 computation in the blocked algorithm executes s times as fast as the level-1 and level-2 computation found in the corresponding unblocked algorithm; and (2) that memory operations (rather than floating-point operations) are the limiting factor to performance in the unblocked algorithm. We estimate speedup for $s = 4$ and $s = 5$.

8.1 Platform details

All experiments reported in this paper were performed on a single core of a Dell PowerEdge R900 server consisting of four Intel "Dunnington" six-core processors. Each core provides a peak performance of 10.64

GFLOPS. Performance experiments were gathered under the GNU/Linux 2.6.18 operating system. Source code was compiled by the GNU C compiler, version 4.1.2. All experiments were performed in double-precision real floating-point arithmetic.

All reduction to condensed form implementations were linked to the BLAS provided by GotoBLAS2 1.10. All LAPACK implementations were obtained via the netlib distribution of LAPACK version 3.3.1. For the reduction to bidiagonal form we also compare against an implementation by Howell et al. (Algorithm III), reported on in [15] and available from [14]. (This code was compiled by the GNU Fortran compiler, version 4.1.2.)

8.2 Fused operation implementations

Experiments were performed with both cache-level and register-level fused implementations. All implementations were coded in C. Operations fused at the cache-level were expressed in terms of level-1 BLAS. By contrast, operations fused at the register-level were coded using SSE2 and SSE3 vector intrinsics. The corresponding assembly code of each register-level fused kernel was carefully inspected to ensure that (1) the correct vector arithmetic instructions were emitted by the compiler and (2) the number of load/store instructions were kept to a minimum. We believe that the resulting fused implementations are, for the most part, comparable to what one would arrive at if the operation were assembly-coded by hand.

Some readers may wonder why we chose to implement our cache-level fused operations in terms of level-1 operations rather than the level-2 approach taken by [15]. It is true that if a cache-level fused implementation is based on level-2 operations, it can automatically benefit from certain optimizations that may be employed within the level-2 BLAS that are not available to level-1 operations. For example, one such optimization involves unrolling the outer loop of the GEMV operation L times (provided there are enough registers available to support the unrolling). This allows the implementation to reduce the number of memory accesses on either the input or output vector by a factor of L. However, the implementation is not *required* to do so, and the specific details concerning a BLAS library's implementation are oftentimes not available. Without knowing exactly how the level-2 operations are implemented, we cannot precisely quantify the number of memory operations avoided by using register-level fusing. Therefore, we implement cache-level fusing in terms of level-1 operations not because we think it yields the best possible performance, but because we can model its performance without making overly-specific assumptions about the implementation.

8.3 Implementations of the reduction algorithms

The blocked algorithms were implemented using the FLAME/C API [28, 4] which allows the implementations to closely mirror the algorithms presented in this paper. Since this API carries considerable overhead that affects performance, the unblocked algorithms were translated into lower-level implementations that use the BLAS-like Interface Subprograms (BLIS) interface [31]. This is a C interface that (1) resembles the BLAS interface but is more natural for C, and (2) fixes certain problems for the routines that compute with (single- and double-precision) complex datatypes. All these implementations are part of the standard `libflame` distribution so that others can experiment with further optimizations.

8.4 Tuning of block size

We performed experiments to determine the optimal block size for the blocked algorithms. A block size of 32, the default block size for the LAPACK implementation, appeared to be near-optimal and was used for all experiments.

8.5 Reduction to upper Hessenberg form

The table in Figure 15 indicates that there is considerable potential for speedup from fusing for all three fusable algorithms, particularly an algorithm based on the rearranged unblocked algorithm. Performance of the various implementations of reduction to upper Hessenberg form are given in Figure 16, with raw

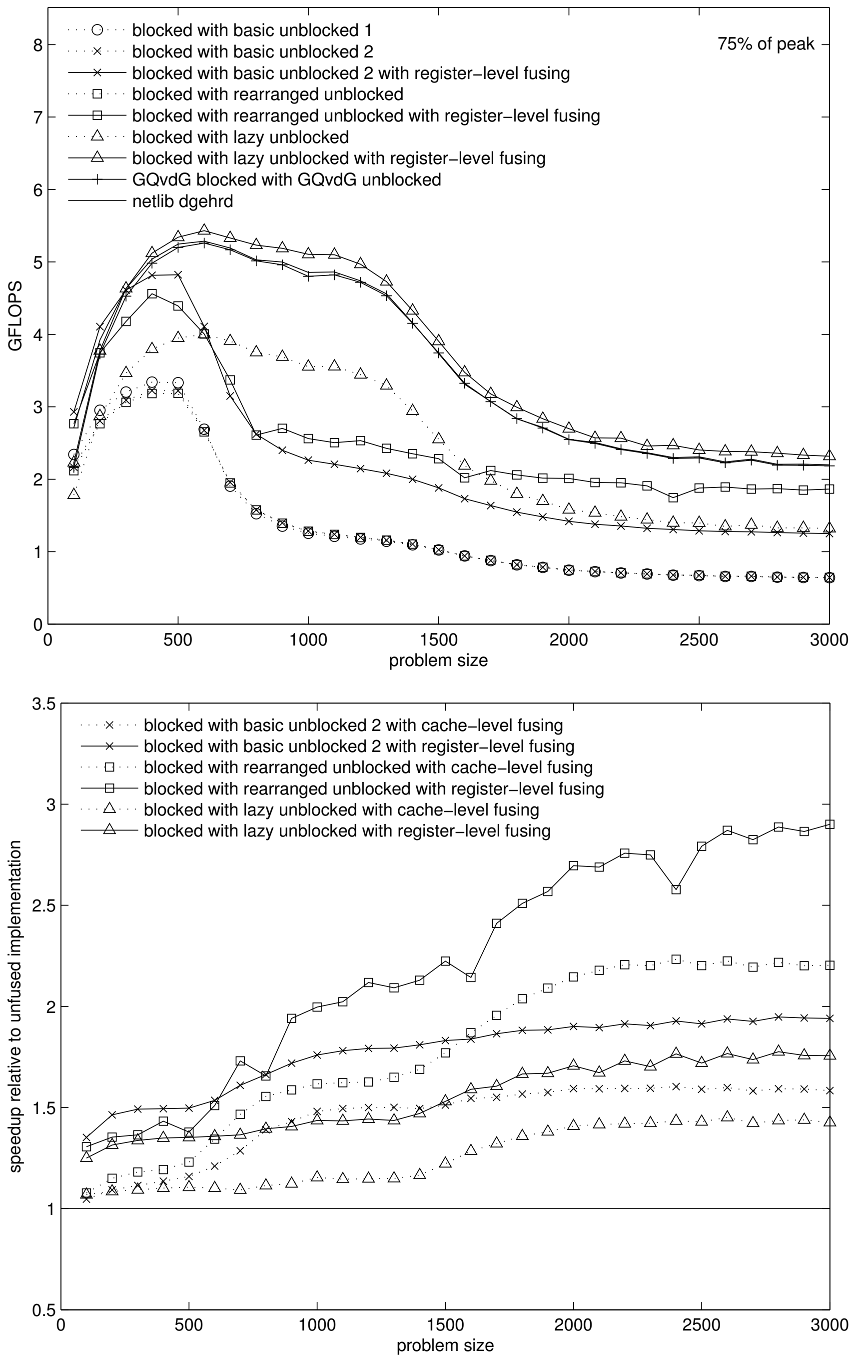


Figure 16: Performance of various implementations of reduction to upper Hessenberg form for problem sizes up to 3000 for double-precision real (top) and speedup of fusable algorithms relative to their unfused counterparts using cache-level and register-level fusing (bottom). Implementations of blocked algorithms use a block size of 32. Note that in the top graph, the performance curve for "netlib dgehrd" coincides mostly with the curve for "GQvdG blocked with GQvdG unblocked."

performance results in the top graph and speedup of fusable algorithms, using both cache-level and register-level fusing, shown in the bottom graph.

Not surprisingly, register-level fusing provides a significant improvement in performance over cache-level fusing. Remarkably, the speedups predicted by the model, as summarized in Figure 15, provide good estimates of the performance of algorithm implementations that use register-level fusing.

For larger matrices ($n \geq 300$), the blocked implementation that uses a lazy unblocked algorithm with register-level fusing (labeled "blocked with lazy unblocked with register-level fusing") outperforms all other implementations, even the netlib `dgehrd` and "GQvdG blocked with GQvdG unblocked" implementations. Note that netlib `dgehrd` uses the "GQvdG blocked with GQvdG unblocked" algorithm, with the minor modification that the algorithm switches to what is essentially our pure basic unblocked algorithm for the final 128×128 subproblem (when A_{BR} is 128×128).

8.6 Reduction to tridiagonal form

In contrast to reduction to upper Hessenberg form, Figure 15 suggests that there is much less room for improvement via fusing in the reduction to tridiagonal form algorithms, particularly for the lazy algorithm.

The reason for the negligible potential for speedup in the lazy algorithm can be traced back to the memory and flop count analysis in Figures 13 and 14. The reduction in memory operations that may be achieved via fusing within the unblocked lazy algorithm constitutes a lower-order term. Likewise, the floating-point operations in the fusable portions of this algorithm amount to a similar lower-order term. Thus, we would expect very little performance benefit from fusing for this algorithmic variant. By contrast, a simple rearranged unblocked algorithm should stand to benefit noticeably from fusing. However, with none of its computation expressible in terms of level-3 operations, such an algorithm is bound to asymptotically underperform its lazy counterpart.

Figure 17 (top) reports performance for various implementations of reduction to tridiagonal form, with corresponding speedups for the two fusable algorithms displayed in Figure 17 (bottom). The fused implementations perform mostly as expected.

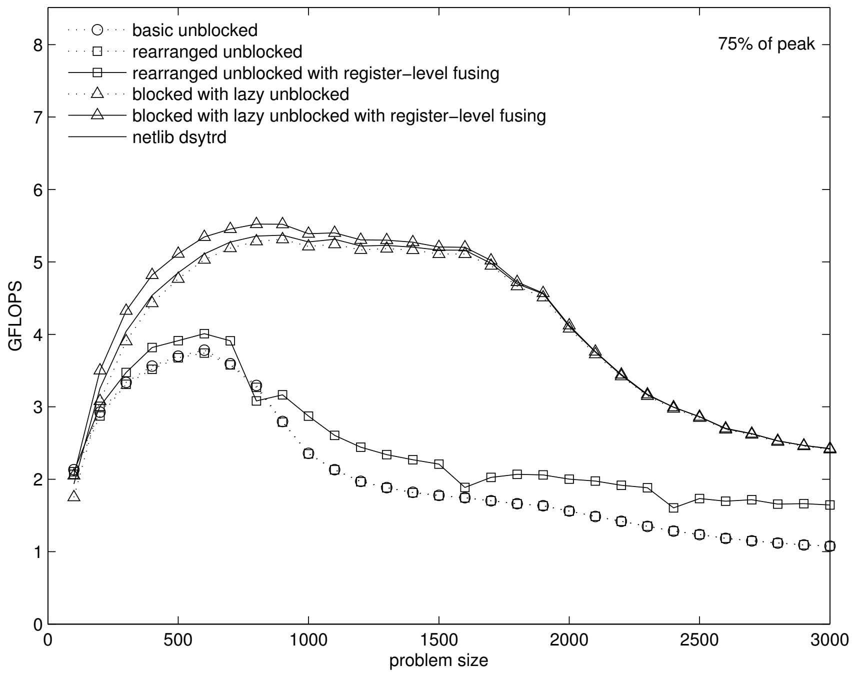
8.7 Reduction to bidiagonal form

According to Figure 15, reduction to bidiagonal form should receive significant benefit from fusing.

Figure 18 (top) reports performance for various implementations of reduction to bidiagonal form while Figure 18 (bottom) shows speedups for fusable algorithms. For this operation there is a clear advantage gained from rearranging the computations and fusing operations, particularly when register-level fusing is employed. With the exception of small problem sizes, the "blocked with lazy rearranged with register-level fusing" outperforms all others, including the implementation of Algorithm III reported on in [15]. Once again, our simple model provides good estimates of the asymptotic speedup for each fusable algorithm.

The performance results for "blocked with lazy rearranged unblocked with cache-level fusing", along with Howell's Algorithm III, clearly show that considerable improvement can be gained from cache-level fusing. However, as one might expect, accessing an element of data from cache is still more costly than avoiding the memory operation altogether, as the "blocked with lazy rearranged unblocked with register-level fusing" exhibits the highest performance, except for the smallest problem sizes.

Note that in Figure 18 (bottom) Howell's Algorithm III outperforms the "blocked with lazy rearranged unblocked with cache-level fusing" algorithm by a small margin. The two algorithm implementations are similar except that the former (1) fuses in terms of level-2 BLAS instead of level-1 BLAS, and (2) is coded entirely in Fortran-77 rather than C with higher-level FLAME abstractions. Given that both styles of cache-level fusing incur the same number of memory operations, we suspect the outperformance can be explained almost entirely by the latter point, as modern compilers tend to be able to more highly optimize pure Fortran-77 over C that contains some calls to the FLAME/C APIs. Thus, it may be possible to achieve marginal improvements in performance of all register-level fused implementations by removing all programming abstractions and coding entirely at low levels.



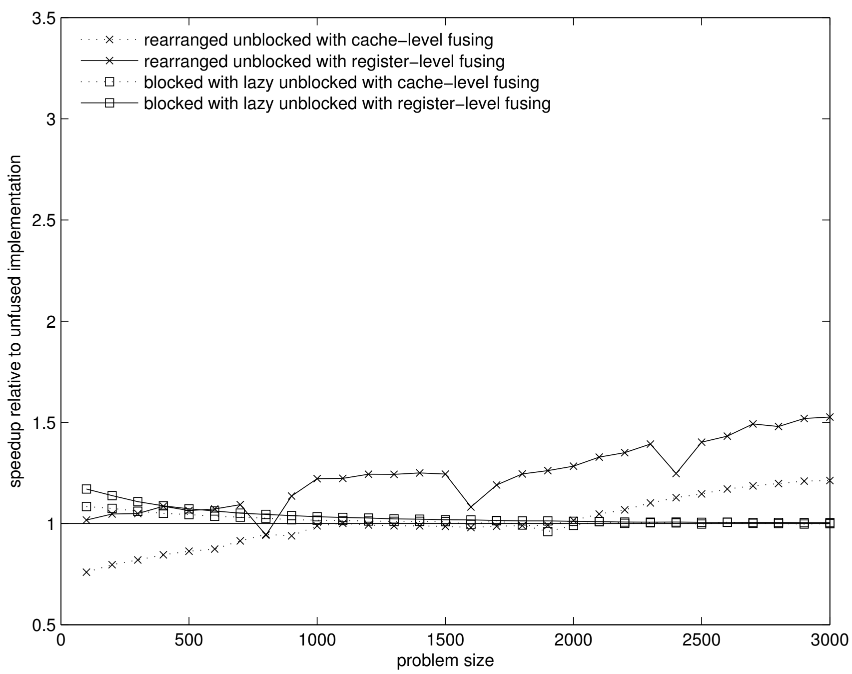


Figure 17: Performance of various implementations of reduction to tridiagonal form for problem sizes up to 3000 for double-precision real (top) and speedup of fusable algorithms relative to their unfused counterparts using cache-level and register-level fusing (bottom). Implementations of blocked algorithms use a block size of 32. Note that in the top graph, the performance curve for "netlib dsytrd" coincides mostly with the curve for "blocked with lazy unblocked with register-level fusing."

8.8 Hybrid algorithms

In Figure 18 (top) it can be observed that, for smaller problem sizes ($n \leq 500$), the "rearranged unblocked with register-level fusing" algorithm yields the best performance. This suggests that a library routine should switch algorithms as a function of problem size. Note that the netlib LAPACK implementations of all three condensed form operations tested in this paper employ hybrid approaches, albeit with different crossover points. The netlib routines for reduction to upper Hessenberg form (`dgehrd`) and reduction to bidiagonal form (`dgebrd`) switch to basic unblocked algorithms for the final 128×128 submatrix, while the routine for reduction to tridiagonal form (`dsytrd`) switches for the final 32×32 submatrix.

Hybrid algorithms for all three reduction to condensed form operations can be constructed in a straightforward manner, and thus we omit results for such implementations from this paper.

9 Conclusion

This paper presents what we believe to be the most complete analysis to date of algorithms for reducing matrices to condensed form. Numerous algorithms are summarized and opportunities for rearranging and fusing of operations are exposed. The benefit of cache-level fusing is confirmed, while more highly-optimized register-level fusing is shown, in theory and practice, to offer superior gain. These performance improvements based on register-level fused kernels conform reasonably well to the speedups predicted by a simple model.

Future work in this area will investigate the impact of fusing in multicore environments.

Acknowledgments This research was partially sponsored by NSF grants OCI-0850750 and NSF CCF-0917167, and a grant from Microsoft.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[3] Paolo Bientinesi, Francisco D. Igual, Daniel Kressner, Matthias Petschow, and Enrique S. Quintana-Ortí. Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. *Concurrency and Computation: Practice and Experience*, 23:694–707, May 2011.

[4] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[5] Christian Bischof, Bruno Lang, and Xiaobai Sun. Parallel tridiagonalization through two-step band reduction. In *In Proceedings of the Scalable High-Performance Computing Conference*, pages 23–27. IEEE Computer Society Press, 1994.

[6] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s2–s13, Jan. 1987.

[7] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. Algorithm 807: The sbr toolbox-software for successive band reduction. *ACM Trans. Math. Soft.*, 26:602–616, December 2000.

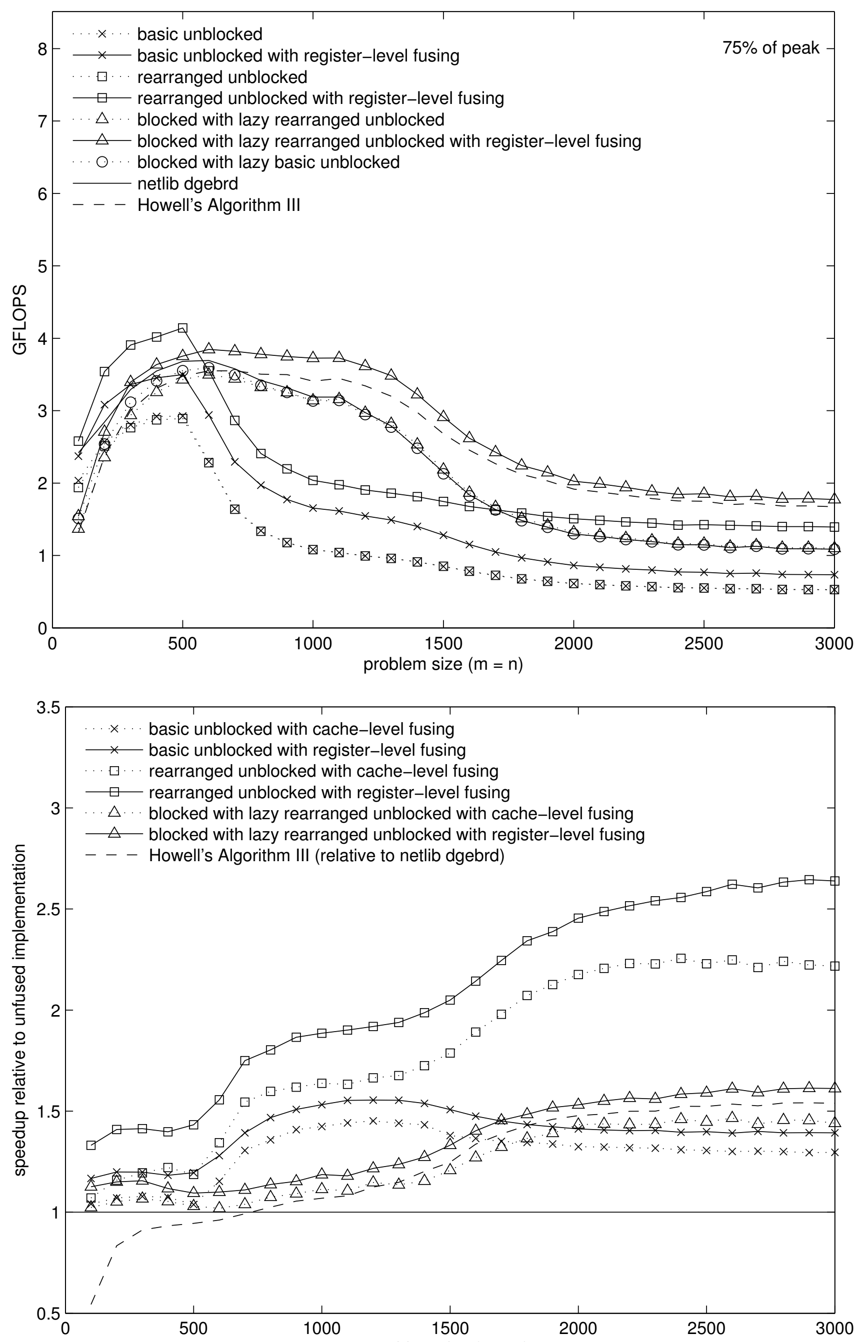


Figure 18: Performance of various implementations of reduction to bidiagonal form for problem sizes up to 3000 for double-precision real (top) and speedup of fusable algorithms relative to their unfused counterparts using cache-level and register-level fusing (bottom). Implementations of blocked algorithms use a block size of 32.

[8] Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 16(1), Spring 2002.

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

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

[11] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, PA, 1991.

[12] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27, 1989.

[13] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[14] Gary Howell. Fortran 77 codes for Householder bidiagonalization. `http://www.ncsu.edu/itd/hpc/Documents/Publications/gary_howell/030905.t%ar`, 2005.

[15] Gary W. Howell, James W. Demmel, Charles T. Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software*, 34(3):14:1–14:33, May 2008.

[16] Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field Van Zee. Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software*, 32(2):169–179, June 2006.

[17] Bruno Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Comput.*, 25:845–860, July 1999.

[18] Representation of orthogonal or unitary matrices. `http://www.netlib.org/lapack/lug/node128.html`.

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

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

[21] C. Puglisi. Modification of the Householder method based on the compact wy representation. *SIAM J. Sci. Stat. Comput.*, 13:723–726, 1992.

[22] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[23] Gregorio Quintana-Ortí and Robert van de Geijn. Improving the performance of reduction to Hessenberg form. *ACM Transactions on Mathematical Software*, 32(2):180–194, June 2006.

[24] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, Jan. 1989.

[25] G. W. Stewart. *Matrix algorithms Volume II: Eigensystems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[26] Xiaobai Sun. Aggregations of elementary transformations. Technical Report Technical report DUKE–TR–1996–03, Duke University, 1996.

[27] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations.* www.lulu.com, 2008.

[28] Field G. Van Zee. *libflame: The Complete Reference.* www.lulu.com, 2011a.

[29] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *Computing in Science and Engineering*, 11:56–63, 2009.

[30] Field G. Van Zee, Gregorio Quintana-Ortí, Robert van de Geijn, and G. Joseph Elizondo. Algorithms for reducing a matrix to condensed form. FLAME Working Note #53 TR-10-37, The University of Texas at Austin, Department of Computer Science, October 2010b. Submitted to ACM TOMS.

[31] Richard M. Veras, Jonathan S. Monette, Field G. Van Zee, Robert A. van de Geijn, and Enrique S. Quintana-Ortí. FLAMES2S: From abstraction to high performance. *ACM Trans. Math. Soft.* submitted. Available from http://z.cs.utexas.edu/wiki/flame.wiki/Publications/.

[32] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, 1988.

A Computing in the complex domain

For simplicity and clarity, the algorithms given thus far have assumed computation on real matrices. In this appendix, we briefly discuss how to formulate a few of these algorithms for complex matrices.

In order to capture more generalized algorithms which work in both the real and complex domains, we must first introduce a complex Householder transform.

Definition 2 *Let $u \in \mathbb{C}^n$, $\tau \in \mathbb{R}$. Then $H = H(u) = I - \tau^{-1} u u^H$, where $\tau = \frac{1}{2} u^H u$, is a complex Householder transformation.*

The complex Householder transform has properties similar to those of the real instantiation, namely: (1) $HH = I$; (2) $H = H^H$, and so $H^H H = H H^H = I$; and (3) if $H_0, \cdots, H_{k-1}$ are complex Householder transformations and $Q = H_0 H_1 \cdots H_{k-1}$, then $Q^H Q = Q Q^H = I$.

Let $x, v, u \in \mathbb{C}^n$,

$$x \rightarrow \left(\frac{\chi_1}{x_2} \right), v \rightarrow \left(\frac{\nu_1}{v_2} \right), u \rightarrow \left(\frac{\upsilon_1}{u_2} \right),$$

$v = x - \alpha e_0$, and $u = v/\nu_1$. We can re-express the complex Householder transform H as:

$$H \;=\; \left(I - \tau^{-1} \left(\frac{1}{u_2} \right) \left(\frac{1}{u_2} \right)^H \right)$$

It can be shown that the application of $H(u)$ to a vector x,

$$H \left(\frac{\chi_1}{x_2} \right) \;=\; \left(\frac{\alpha}{0} \right) \tag{5}$$

is satisfied for

$$\alpha \;=\; -\frac{\|x\|_2 \chi_1}{|\chi_1|}. \tag{6}$$

Notice that for $x, v, u \in \mathbb{R}^n$, this definition of α is equivalent to the definition given for real Householder transformations in Section 2.2, since $\chi_1/|\chi_1| = \text{sign}(\chi_1)$. By re-defining α this way, we allow τ to remain real, which allows the complex Householder transform to retain the property of being a reflector.

There is one drawback to this approach, however. Applying $H(u)$ to x results in α being a complex value. This causes problems for some applications. For example, in the case of reduction to tridiagonal form, the values of α generated by each transformation form the off-diagonal elements of the tridiagonal matrix. Typically, one wishes these off-diagonal elements to be real because it simplifies the arithmetic in subsequent computations. But it turns out there is a straightforward solution to this problem.

First, we leverage the fact that a complex off-diagonal element α_i from the ith row (or column) of a tridiagonal matrix T can be rotated into the real domain by computing a complex scalar $\rho_i = \bar{\alpha}_i/|\alpha_i|$ and then scaling the ith row of T by ρ_i and the ith column of T by $\bar{\rho}_i$. Notice that this has no effect on the ith diagonal element T_{ii} since $\rho_i \bar{\rho}_i = 1$. Also, this guarantees that $\rho_i \alpha_i$ is positive. Thus, one can easily compute and apply a diagonal matrix R that transforms a tridiagonal matrix T with complex off-diagonals to a real tridiagonal matrix T_R. In this case, the overall reduction to real tridiagonal form becomes:

$$\begin{aligned} A &= Q T Q^H \\ &= Q R^H R T R^H R Q^H \\ &= Q R^H T_R R Q^H \\ &= Q R^H T_R (Q R^H)^H \end{aligned}$$

And so the total cost of defining complex Householder transforms as reflectors amounts to: (1) computing R; (2) applying R to T from the left and R^H from the right; and (3) applying R^H to Q from the right. These operations are $\mathcal{O}(n)$, $\mathcal{O}(n)$, and $\mathcal{O}(n^2)$, respectively, and are typically lower-order terms within larger $\mathcal{O}(n^3)$ computations. This process is similar to the one described by Stewart in [25].

Other instances of the Householder transform, such as those found in LAPACK, restrict α to the real domain [18, 20]. In these situations, Eq. (5) is satisfiable only if $\tau \in \mathbb{C}$, which results in $HH \neq I$. This definition has the benefit of giving real α values without any additional scaling, but results in mathematics and implementation code that are somewhat more complicated (ie: one must keep track of whether it is appropriate to apply H or H^H).

Ultimately, we prefer our Householder transforms to remain reflectors in both the real and complex domains, and so we choose to define α as in Eq. (6).

Recall that Figures 1–11 illustrate algorithms for computing on real matrices. We will now review a few of the algorithms, as expressed in terms of the complex Householder transform.

A.1 Reduction to upper Hessenberg form

Since the complex Householder transform H is a reflector, the basic unblocked algorithm for reducing a complex matrix to upper Hessenberg is, at a high level, identical to the algorithm for real matrices:

- Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
- Let $[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$.
- Update

$$\left(\begin{array}{c|c} a_{01} & A_{02} \\ \hline \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & H \end{array}\right) \left(\begin{array}{c|c} a_{01} & A_{02} \\ \hline \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & H \end{array}\right) = \left(\begin{array}{c|c} a_{01} & A_{02}H \\ \hline \alpha_{11} & a_{12}^T H \\ \hline H a_{21} & H A_{22} H \end{array}\right)$$

  where $H = H(u_{21})$. Note that $a_{21} := Ha_{21}$ need not be executed since this update was performed by the instance of $\textsc{Housev}$ above.
- Continue this process with the updated A_{22}.

As before, Ha_{21} is computed by $\textsc{Housev}$.

The real and complex algorithms begin to differ with the updates of a_{12}^T and A_{02}:

$$\begin{array}{rcl} a_{12}^T & := & a_{12}^T H \\ & = & a_{12}^T - a_{12}^T u_{21} u_{21}^H / \tau \\ A_{02} & := & A_{02} H \\ & = & A_{02} - A_{02} u_{21} u_{21}^H / \tau \end{array}$$

Specifically, we can see that u_{21} is conjugate-transposed instead of simply transposed.

The remaining differences can be seen by inspecting the update of A_{22}:

$$\begin{array}{rcl} A_{22} & := & H A_{22} H \\ & = & (I - u_{21} u_{21}^H / \tau) A_{22} (I - u_{21} u_{21}^H / \tau) \\ & = & A_{22} - u_{21} (\underbrace{A_{22}^H u_{21}}_{v_{21}})^H / \tau - (\underbrace{A_{22} u_{21}}_{w_{21}}) u_{21}^H / \tau + (u_{21}^H \underbrace{A_{22} u_{21}}_{w_{21}}) u_{21} u_{21}^H / \tau^2 \\ & = & A_{22} - u_{21} v_{21}^H / \tau - w_{21} u_{21}^H / \tau + \underbrace{u_{21}^H w_{21}}_{2\beta} u_{21} u_{21}^H / \tau^2 \\ & = & A_{22} - u_{21} (v_{21}^H - \beta u_{21}^H / \tau) / \tau - ((w_{21} - \beta u_{21} / \tau) / \tau) u_{21}^H \\ & = & A_{22} - u_{21} \underbrace{((v_{21} - \bar{\beta} u_{21} / \tau) / \tau)}_{y_{21}}^H - \underbrace{((w_{21} - \beta u_{21} / \tau) / \tau)}_{z_{21}} u_{21}^H \\ & = & A_{22} - (u_{21} y_{21}^H + z_{21} u_{21}^H) \end{array}$$

This leads towards the basic and rearranged unblocked algorithms in Figure 19.

Algorithm: $[A] := \textsc{ComplexHessRed\_unb}(b, A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $u \rightarrow \left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right)$, $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, $z \rightarrow \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right)$

where A_{TL} is 0×0 and u_T, y_T, and z_T have 0 rows

while $m(A_{TL}) < b$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right) \rightarrow \left(\begin{array}{c} u_{01} \\ \hline \upsilon_{11} \\ \hline u_{21} \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_{01} \\ \hline \psi_{11} \\ \hline y_{21} \end{array}\right), \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \rightarrow \left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right)$$

where α_{11}, υ_{11}, ψ_{11}, ζ_{11} are scalars

Basic unblocked 1:

$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$

$A_{22} := (I - u_{21}u_{21}^H/\tau)A_{22} = A_{22} - u_{21}u_{21}^H A_{22}/\tau$

$$\left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} A_{02} \\ \hline a_{12}^T \\ \hline A_{22} \end{array}\right)(I - u_{21}u_{21}^H/\tau) = \left(\begin{array}{c} A_{02} - A_{02}u_{21}u_{21}^H/\tau \\ \hline a_{12}^T - a_{12}^T u_{21}u_{21}^H/\tau \\ \hline A_{22} - A_{22}u_{21}u_{21}^H/\tau \end{array}\right)$$

Basic unblocked 2:

$[u_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$

$y_{21} := A_{22}^H u_{21}$
$z_{21} := A_{22} u_{21}$

$\beta := u_{21}^H z_{21}/2$
$y_{21} := (y_{21} - \bar{\beta} u_{21}/\tau)/\tau$
$z_{21} := (z_{21} - \beta u_{21}/\tau)/\tau$
$A_{22} := A_{22} - u_{21}y_{21}^H - z_{21}u_{21}^H$
$a_{12}^T := a_{12}^T - a_{12}^T u_{21}u_{21}^H/\tau$
$A_{02} := A_{02} - A_{02}u_{21}u_{21}^H/\tau$

Rearranged unblocked:

$\alpha_{11} := \alpha_{11} - \upsilon_1\bar{\psi}_1 - \zeta_1\bar{\upsilon}_1$ $(\star)$
$a_{12}^T := a_{12}^T - \upsilon_1 y_{21}^H - \zeta_1 u_{21}^H$ $(\star)$
$a_{21} := a_{21} - u_{21}\bar{\psi}_1 - z_{21}\bar{\upsilon}_1$ $(\star)$
$[x_{21}, \tau, a_{21}] := \textsc{Housev}(a_{21})$
$A_{22} := A_{22} - u_{21}y_{21}^H - z_{21}u_{21}^H$ $(\star)$
$v_{21} := A_{22}^H x_{21}$
$w_{21} := A_{22} x_{21}$
$u_{21} := x_{21}$; $y_{21} := v_{21}$
$z_{21} := w_{21}$
$\beta := u_{21}^H z_{21}/2$
$y_{21} := (y_{21} - \bar{\beta} u_{21}/\tau)/\tau$
$z_{21} := (z_{21} - \beta u_{21}/\tau)/\tau$

$a_{12}^T := a_{12}^T - a_{12}^T u_{21}u_{21}^H/\tau$
$A_{02} := A_{02} - A_{02}u_{21}u_{21}^H/\tau$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} u_T \\ \hline u_B \end{array}\right) \leftarrow \left(\begin{array}{c} u_{01} \\ \hline \upsilon_{11} \\ \hline u_{21} \end{array}\right), \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_{01} \\ \hline \psi_{11} \\ \hline y_{21} \end{array}\right), \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \leftarrow \left(\begin{array}{c} z_{01} \\ \hline \zeta_{11} \\ \hline z_{21} \end{array}\right)$$

endwhile

Figure 19: Unblocked reduction to upper Hessenberg form using a complex Householder transform. Left: basic algorithm. Right: rearranged algorithm so that operations can be fused. Operations marked with $(\star)$ are not executed during the first iteration.

Algorithm: $[A] := \text{ComplexTriRed\_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

for $x \in \{u, y\}$

where A_{TL} is 0×0 and u_T, y_T have 0 rows

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

for $(x, \chi) \in \{(u, \upsilon), (y, \psi)\}$

where α_{11}, υ_{11}, and ψ_{11} are scalars

Basic unblocked:	Rearranged unblocked:	
	$\alpha_{11} := \alpha_{11} - \upsilon_{11}\bar{\psi}_{11} - \upsilon_{11}\bar{\psi}_{11}$	$(\star)$
	$a_{21} := a_{21} - (u_{21}\bar{\psi}_{11} + y_{21}\bar{\upsilon}_{11})$	$(\star)$
$[u_{21}, \tau, a_{21}] := \text{Housev}(a_{21})$	$[x_{21}, \tau, a_{21}] := \text{Housev}(a_{21})$	
	$A_{22} := A_{22} - u_{21}y_{21}^H - y_{21}u_{21}^H$	$(\star)$
$y_{21} := A_{22}u_{21}$	$v_{21} := A_{22}x_{21}$	
	$u_{21} := x_{21}$; $y_{21} := v_{21}$	
$\beta := u_{21}^H y_{21}/2$	$\beta := u_{21}^H y_{21}/2$	
$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$	$y_{21} := (y_{21} - \beta u_{21}/\tau)/\tau$	
$A_{22} := A_{22} - u_{21}y_{21}^H - y_{21}u_{21}^H$		

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

for $(x, \chi) \in \{(u, \upsilon), (y, \psi)\}$

endwhile

Figure 20: Unblocked reduction to tridiagonal form using a complex Householder transformation. Left: basic algorithm. Right: rearranged to allow fusing of operations. Operations marked with $(\star)$ are not executed during the first iteration.

A.2 Reduction to tridiagonal form

Let $A \in \mathbb{C}^{n\times n}$ be Hermitian. If $A \rightarrow QBQ^H$ where B is upper Hessenberg and Q is unitary, then B is Hermitian and therefore tridiagonal. We may take advantage of the Hermitian structure of A just as we did with symmetry in Section 4. Let us assume that only the lower triangular part of A is stored and read, and that only the lower triangular part is overwritten by B.

When matrix A is Hermitian, and only the lower triangular part is referenced, the unblocked algorithms for reducing A to upper Hessenberg form can be changed by noting that $v_{21} = w_{21}^H$ and $y_{21} = z_{21}^H$. This results in the basic and rearranged unblocked algorithms shown in Figure 20.

A.3 Reduction to bidiagonal form

The basic algorithm for reducing a complex matrix to bidiagonal form can be explained as follows:

- Partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$.
- Let $\left[\left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ \hline 0 \end{array}\right)\right] := \text{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)\right)$.

Algorithm: $[A] := \text{ComplexBiRed\_unb}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$

for $x \in \{u, v, y, z\}$

where A_{TL} is 0×0, u_T, v_T, y_T, z_T have 0 elements

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

for $(x, \chi) \in \{(u, \upsilon), (v, \nu), (y, \psi), (z, \zeta)\}$

where α_{11}, υ_{11}, ν_{11}, ψ_{11}, and ζ_{11} are scalars

Basic unblocked:

$$\left[\left(\begin{array}{c} 1 \\ u_{21} \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ 0 \end{array}\right)\right] := \text{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ a_{21} \end{array}\right)\right)$$

$y_{21} := \bar{a}_{12} + A_{22}^H u_{21}$
$a_{12}^T := a_{12}^T - y_{21}^H / \tau_L$

$[v_{21}, \tau_R, a_{12}] := \text{Housev}\,(a_{12})$

$\beta := y_{21}^H v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (A_{22} v_{21} - \beta u_{21} / \tau_L) / \tau_R$

$A_{22} := A_{22} - u_{21} y_{21}^H - z_{21} v_{21}^H$

Rearranged unblocked:

$\alpha_{11} := \alpha_{11} - \upsilon_{11} \bar{\psi}_{11} - \zeta_{11} \bar{\nu}_{11}$ $(\star)$
$a_{21} := a_{21} - u_{21} \bar{\psi}_{11} - z_{21} \bar{\nu}_{11}$ $(\star)$
$a_{12}^T := a_{12}^T - \upsilon_{11} y_{21}^H - \zeta_{11} v_{21}^H$ $(\star)$

$$\left[\left(\begin{array}{c} 1 \\ u_{21}^+ \end{array}\right), \tau_L, \left(\begin{array}{c} \alpha_{11} \\ 0 \end{array}\right)\right] := \text{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ a_{21} \end{array}\right)\right)$$

$a_{12}^+ := a_{12} - a_{12} / \tau_L$
$A_{22} := A_{22} - u_{21} y_{21}^H - z_{21} v_{21}^H$ $(\star)$
$y_{21} := A_{22}^H u_{21}^+$
$a_{12}^+ := a_{12}^+ - \bar{y}_{21} / \tau_L$
$w_{21} := A_{22} \bar{a}_{12}^+$
$y_{21} := y_{21} + \bar{a}_{12}$
$[\psi_{11} - \alpha_{12}, \tau_R, \alpha_{12}] := \text{Houses}(a_{12}^+)$
$v_{21} := (\bar{a}_{12}^+ - \bar{\alpha}_{12} e_0) / (\bar{\psi}_{11} - \bar{\alpha}_{12})$
$a_{12}^T := \alpha_{12} e_0^T$
$u_{21} := u_{21}^+$
$\beta := y_{21}^H v_{21}$
$y_{21} := y_{21} / \tau_L$
$z_{21} := (w_{21} - \bar{\alpha}_{12} A_{22} e_0) / (\bar{\psi}_{11} - \bar{\alpha}_{12})$
$z_{21} := z_{21} - \beta u_{21} / \tau_L$
$z_{21} := z_{21} / \tau_R$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_{01} \\ \hline \chi_{11} \\ \hline x_{21} \end{array}\right)$$

for $(x, \chi) \in \{(u, \upsilon), (v, \nu), (y, \psi), (z, \zeta)\}$

endwhile

Figure 21: Unblocked reduction to bidiagonal form using a complex Householder transformation. Left: basic algorithm. Right: rearranged to allow fusing of operations. Operations marked with $(\star)$ are not executed during the first iteration.

- Update

$$\left(\begin{array}{c|c}\alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22}\end{array}\right) := \left(I - \left(\begin{array}{c}1 \\ \hline u_{21}\end{array}\right)\left(\begin{array}{c}1 \\ \hline u_{21}\end{array}\right)^H / \tau_L\right)\left(\begin{array}{c|c}\alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22}\end{array}\right)$$
$$= \left(\begin{array}{c|c}\alpha - \psi_{11}/\tau_L & a_{12}^T - y_{21}^T/\tau_L \\ \hline 0 & A_{22} - u_{21}y_{21}^T/\tau_L\end{array}\right),$$

  where $\psi_{11} = \alpha_{11} + u_{21}^H a_{21}$ and $y_{21}^T = a_{12}^T + u_{21}^H A_{22}$. Note that $\alpha_{11} := \alpha - \psi_{11}/\tau_L$ need not be executed since this update was performed by the instance of HOUSEV above.

- Let $[v_{21}, \tau_R, a_{12}] := \text{HOUSEV}\,(a_{12})$.

- Update $A_{22} := A_{22}(I - v_{21}v_{21}^T/\tau_R) = A_{22} - z_{21}v_{21}^T/\tau_R$, where $z_{21} = A_{22}v_{21}$.

- Continue this process with the so updated A_{22}.

The resulting unblocked algorithm and a rearranged variant that allows fusing are given in Figure 21.

A.4 Blocked algorithms

Blocked algorithms may be constructed for reduction to upper Hessenberg form by making the following minor changes to the algorithms shown in Figure 4:

- For Algorithms 1–4, update A_{TR} by applying the complex block Householder transform, $(I - U_B T^{-1} U_B^H)$, instead of $(I - U_B T^{-1} U_B^T)$.

- For Algorithm 3, update A_{22} as $A_{22} = A_{22} - U_2 Y_2^H - Z_2 U_2^H$.

- Compute T as $T = \frac{1}{2}D + S$ where $U_B^H U_B = S^H + D + S$.

Blocked algorithms for reduction to tridiagonal form and bidiagonal form can be constructed in a similar fashion.