Restructuring the QR Algorithm for High-Performance Application of Givens Rotations

FLAME Working Note #60

Field G. Van Zee
Robert A. van de Geijn
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712

Gregorio Quintana-Ortí
Departamento de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071, Castellón, Spain

October 18, 2011

Abstract

We show how the QR algorithm can be restructured so that it becomes rich in operations that can achieve near-peak performance on a modern processor. The key is a novel, cache-friendly algorithm for applying multiple sets of Givens rotations, which is then implemented with optimizations that (1) leverage vector instruction units to increase floating-point throughput, and (2) fuse multiple rotations to decrease the total number of memory operations. We demonstrate the merits of this new QR algorithm for computing the Hermitian (symmetric) eigenvalue decomposition and singular value decomposition of dense matrices when all eigenvectors/singular vectors are computed. The approach yields vastly improved performance relative to the traditional QR algorithm and is competitive with two commonly used alternatives—Cuppen's Divide and Conquer algorithm and the Method of Multiple Relatively Robust Representations—while inheriting the more modest $\mathcal{O}(n)$ workspace requirements of the original QR algorithm. Since the computations performed by the restructured algorithm remain essentially identical to those performed by the original method, robust numerical properties are preserved.

1 Introduction

The QR algorithm is taught in a typical graduate-level numerical linear algebra course, and despite being among the most accurate methods for solving eigenvalue and singular value problems, it is not used much in practice because its performance is not competitive [44, 19, 40, 12]. The reason for this is twofold: First, classic QR algorithm implementations, such as those in LAPACK, cast most of their computation (the application of Givens rotations) in terms of a routine that is absent from the BLAS, and thus is typically not available in optimized form. Second, even if such an optimized implementation existed, it would not matter because the QR algorithm is currently structured to apply Givens rotations via repeated instances of $\mathcal{O}(n^2)$ computation on $\mathcal{O}(n^2)$ data, effectively making it rich in level-2 BLAS-like operations which inherently cannot achieve high performance because there is minimal opportunity for reuse of cached data. Many in the numerical linear algebra community have long speculated that the QR algorithm's performance could be improved by saving up many sets of Givens rotations before applying them to the matrix in which eigenvectors or singular vectors are being accumulated. In this paper, we show that, when computing all eigenvectors

of a dense Hermitian matrix or all singular vectors of a dense general matrix, dramatic improvements in performance can indeed be achieved.

This work makes a number of contributions to this subject:

- It exposes a hidden $\mathcal{O}(n^3)$ cost associated with the back-transformation that follows after the method of Multiple Relatively Robust Representations (MRRR) [13, 14] and Cuppen's Divide-and-Conquer method (D&C) [8].
- It describes how the traditional QR algorithm can be restructured so that computation is cast in terms of an operation that applies many sets of Givens rotations to the matrix in which the eigen-/singular vectors are accumulated.
- It proposes an algorithm for applying many sets of Givens rotations that, in theory, exhibits greatly improved reuse of data in the cache. It then shows that an implementation of this algorithm can achieve near-peak performance by (1) utilizing vector instruction units to increase floating-point operation throughput, and (2) fusing multiple rotations so that data can be reused in-register, which decreases costly memory operations.
- It demonstrates performance of EVD via the QR algorithm that is competitive with that of D&C- and MRRR-based EVD, and QR-based SVD performance that is comparable to D&C-based SVD.
- It argues that the resulting approach requires only linear ($\mathcal{O}(n)$) workspace, while the other methods discussed require larger $\mathcal{O}(n^2)$ workspaces. (Though an optional optimization that requires $\mathcal{O}(n^2)$ workspace is also discussed and tested.)
- It makes the resulting implementations available as part of the open source `libflame` library.

The paper primarily focuses on the complex case since the mathematics trivially extends to the real case.

We consider these results to be significant in part because we place a premium on simplicity and reliability. The restructured QR algorithm presented in this paper is simple, gives performance that is almost as good as that of more complex algorithms (such as D&C and MRRR) and does so using only $\mathcal{O}(n)$ workspace, and without the worry of what might happen to numerical accuracy in pathological cases such as tightly clustered eigen-/singular values.

Before moving on, we now briefly review the notational conventions used in this paper. Scalars, vectors, and matrices are typically represented by lower case Greek, lower case Roman, and upper case Roman letters, respectively. Certain lowercase letters, such as m, n, i, j, k, and b, are used to represent integer indices into and/or dimensions of matrices and vectors. We try to coordinate the naming of scalars and vectors that are referenced from within matrices. For example, $\alpha_{i,j}$ denotes the (i, j) entry in matrix A and a_j its jth column.

2 Computing the Spectral Decomposition of a Hermitian Matrix

Given a Hermitian matrix $A \in \mathbb{C}^{n\times n}$, its eigenvalue decomposition (EVD) is given by $A = QDQ^H$, where $Q \in \mathbb{C}^{n\times n}$ is unitary ($Q^HQ = I$) and $D \in \mathbb{R}^{n\times n}$ is diagonal. The eigenvalues of matrix A can then be found on the diagonal of D while the corresponding eigenvectors are the columns of Q. The standard approach to computing the EVD proceeds in three steps [40]: Reduce matrix A to real tridiagonal form T via unitary similarity transformations: $A = Q_A T Q_A^H$; Compute the EVD of T: $T = Q_T D Q_T^T$; and back-transform the eigenvectors of T: $Q = Q_A Q_T$ so that $A = QDQ^H$. Let us discuss these in more detail.

2.1 Reduction to real tridiagonal form

The reduction to tridiagonal form proceeds as the computation and application of a sequence of Householder transformations. When the transformations are defined as reflectors [19, 42, 40], the tridiagonal reduction takes the form $H_{n-2} \cdots H_1 H_0 A H_0 H_1 \cdots H_{n-2} = Q_A^H A Q_A = T$, a real-valued, tridiagonal matrix.[1]

[1] Actually, this produces a matrix with complex off-diagonal elements, which are then "fixed" to become real, as described in [42, 40], an unimportant detail for the present discussion. Note that LAPACK does not need to fix the off-diagonal elements

The cost of the reduction to tridiagonal form is $\frac{4}{3}n^3$ floating-point operations (flops) if A is real and $4 \times \frac{4}{3}n^3$ flops if it is complex-valued. About half of those computations are in symmetric (or Hermitian) matrix-vector multiplications (a level-2 BLAS operation [16]), which are inherently slow since they perform $\mathcal{O}(n^2)$ computations on $\mathcal{O}(n^2)$ data. Most of the remaining flops are in Hermitian rank-k updates (a level-3 BLAS operation) which can attain near-peak performance on modern architectures. If a flop that is executed as part of a level-2 BLAS operation is K_2 times slower than one executed as part of a level-3 BLAS operation, we will give the effective cost of this operation as $(K_2+1)\frac{2}{3}n^3$, where $K_2 \geq 1$ and typically $K_2 \gg 1$. Reduction to tridiagonal form requires approximately bn workspace elements, where b is the algorithmic blocksize used in the blocked algorithm [42, 43].

2.2 Spectral decomposition of T

LAPACK [1] provides four algorithms for computing the Hermitian EVD, which we now discuss. We will use the general term "workspace" to refer to any significant space needed beyond the $n \times n$ space that holds the input matrix A and the n-length space that holds output eigenvalues (ie: the diagonal of D).

The QR algorithm (QR) The cost of the QR algorithm is approximated by $3kn^3$, where k equals the average number of Francis steps before deflation when a trailing eigenvalue has been found. The rule of thumb is that $1 \leq k \leq 2$. For complex matrices A, the cost becomes $2 \times 3kn^3$ because updating a complex eigenvector matrix Q_A with real Givens rotations requires twice as many flops.

The fundamental problem with traditional QR algorithm implementations, such as the one found in LAPACK, is that this $\mathcal{O}(n^3)$ computation is cast in terms of level-1 BLAS-like operations [31] which inherently cannot achieve high performance. If we assume that such operations are K_1 slower than those performed as part of a level-3 BLAS operation, the effective cost becomes $3K_1kn^3$. Generally, $K_1 \geq K_2 \gg 1$. The main contribution of this paper is the restructuring of the computation so that the cost approaches $(2\times)3kn^3$.

The QR algorithm typically requires only $2(n-1)$ real-valued elements of workspace, which constitutes the maximum needed to hold the scalars which define the Givens rotations of any single Francis step. The eigenvectors overwrite the original matrix A.

Cuppen's Divide and Conquer (D&C) An implementation of D&C, similar to the one in LAPACK, costs approximately $\frac{4}{3}\alpha^2 n^3(1-4^{-t})$ flops, where $t = \log_2 n - 1$ and $\alpha = 1 - \delta$ [37]. Here δ represents a typical value for the fraction of eigenvalues at any given level of recursion that deflate. Notice that $(1-4^{-t})$ quickly approaches 1 as n grows larger (thus, we drop this term in our analysis). It has been shown that the D&C algorithm is, in practice, quite fast when operating on matrices that engender a high number of deflations [8, 37, 22]. This is captured in the α parameter, which attenuates quadratically as the number of deflations increase. A small value for α can significantly lessen the impact of the n^3 term. However, for some types of matrices, the number of deflations is relatively low, in which case the algorithm behaves more like a typical $\mathcal{O}(n^3)$ algorithm.

The LAPACK implementation of D&C requires n^2 complex-valued elements and $2n^2 + 4n$ real-valued elements of workspace to form and compute with Q_T, in additional to $5n$ additional integer elements.

Bisection and Inverse Iteration (BII) BII requires $\mathcal{O}(n^2)$ operations to compute eigenvalues and, in the best case, another $\mathcal{O}(n^2)$ operations to compute the corresponding eigenvectors [10]. For matrices with clustered eigenvalues, the latter cost rises to $\mathcal{O}(n^3)$ as the algorithm must also perform modified Gram-Schmidt orthogonalization in an attempt to produce orthogonal eigenvectors [10, 11]. This method of reorthogonalization, however, has been shown to cause BII to fail in both theory and practice [11]. Thus, its use in the community has waned.

The implementation of BII in LAPACK requires n^2, $5n$, and $5n$ complex-, real-, and integer-valued elements of workspace, respectively.

---

because the library defines Householder transformations in such a way that T is reduced directly to real tridiagonal form. However, the cost of this minor optimization is that the transformations do not retain the property of being reflectors (ie: $HH \neq I$), and thus they must sometimes be conjugate-transposed depending on the direction from which they are applied.

EVD Algorithm	Effective higher-order floating-point operation cost		
	$A \rightarrow Q_A T Q_A^H$	$T \rightarrow Q_T D Q_T^T$	Form/Apply Q_A
original QR	$4 \times (K_2 + 1)\frac{2}{3}n^3$	$2 \times K_1 3kn^3$	$4 \times \frac{4}{3}n^3$
restructured QR	$4 \times (K_2 + 1)\frac{2}{3}n^3$	$2 \times 3kn^3$	$4 \times \frac{4}{3}n^3$
D&C	$4 \times (K_2 + 1)\frac{2}{3}n^3$	$\frac{4}{3}\alpha^2 n^3$	$4 \times 2n^3$
BII	$4 \times (K_2 + 1)\frac{2}{3}n^3$	$\mathcal{O}(n^2) \sim \mathcal{O}(n^3)$	$4 \times 2n^3$
MRRR	$4 \times (K_2 + 1)\frac{2}{3}n^3$	$\mathcal{O}(n^2)$	$4 \times 2n^3$

Figure 1: A summary of approximate floating-point operation cost for suboperations of various algorithms for computing a Hermitian eigenvalue decomposition. Integer scalars at the front of the expressions (e.g. "$4 \times \ldots$") indicate the floating-point multiple due to performing arithmetic with complex values. In the cost estimates for the QR algorithms, k represents the typical number of Francis steps needed for an eigenvalue to converge. For Cuppen's Divide-and-conquer algorithm, α represents the number of non-deflations in each recursive subproblem.

Multiple Relatively Robust Representations (MRRR) MRRR is a more recent algorithm that guarantees $\mathcal{O}(n^2)$ computation even in the worst case: when the matrix contains a cluster of eigenvalues, where each eigenvalue in the cluster is identical to d digits of precision, the algorithm must perform work proportional to dn^2 [10].

The MRRR algorithm is widely reported to require only linear $\mathcal{O}(n)$ workspace [1, 14, 10]. But in the context of a dense Hermitian eigenproblem, the workspace required is actually n^2 real-valued elements for the same reason that the D&C and BII algorithms require $\mathcal{O}(n^2)$ workspace—namely, the algorithm inherently must explicitly compute and store the eigenvectors of T before applying Q_A towards the formation of Q. The latest implementation of MRRR in LAPACK as of this writing (version 3.3.1) requires $n^2 + 22n$ and $10n$ real- and integer-valued elements of workspace, respectively.

2.3 Back-transformation

D&C, BII, and MRRR all share the property that they compute and store Q_T explicitly as a dense matrix. The matrix Q_A is stored implicitly as Householder transformations, which are applied to Q_T, yielding Q. This step is known as the back-transformation. The application of Householder transforms can be formulated as a blocked algorithm (either with the traditional compact WY-transform [3, 38] or with the UT-transform used by `libflame` [27, 42]) at a cost of approximately $2n^3$ flops (multiplied by four for complex). This casts most computation in terms of level-3 BLAS.

When the QR algorithm is employed, Q_A is formed before the QR algorithm is called. Forming Q_A is done "in place," overwriting matrix A at a cost of approximately $\frac{4}{3}n^3$ flops (times four for complex), cast mostly in terms of level-3 BLAS. The Givens rotations are subsequently applied directly to Q_A. This analysis exposes a "hidden" $\frac{2}{3}n^3$ flops (times four for complex) in the back-transformation of D&C and MRRR, relative to the corresponding stage in the QR algorithm.

2.4 Numerical accuracy

A recent paper evaluated the numerical accuracy of the four tridiagonal eigenvalue algorithms available in LAPACK [10]. The authors observed that, in practice, the QR and D&C algorithms were the most accurate, exhibiting $\mathcal{O}(\sqrt{n}\varepsilon)$ accuracy, where n is the dimension of the matrix and ε is the machine precision. By contrast, the accuracy of BII and MRRR was observed to be approximately $\mathcal{O}(n\varepsilon)$. Another paper compared the numerical performance of the netlib MRRR (as of LAPACK 3.3.2), netlib D&C, and an improved MRRR algorithm, and found that both of the former begin to break down for at least some of the matrix test cases [46]. The authors assert that the "best method for computing all eigenvalues" is a derivative of the QR

EVD Algorithm	Workspace required for optimal performance		
	$A \rightarrow Q_A T Q_A^H$	$T \rightarrow Q_T D Q_T^T$	Form/Apply Q_A
original QR	$\mathbb{C}$: $(b+1)n$ $\mathbb{R}$: $2n-1$	$\mathbb{R}$: $2b(n-1)$	$\mathbb{C}$: $b(n-1)$
restructured QR	$\mathbb{C}$: $(2b+1)n$ $\mathbb{R}$: $2n-1$	$\mathbb{R}$: $2b(n-1)$	$\mathbb{C}$: $b(n-1)$
D&C	$\mathbb{C}$: $(b+1)n$ $\mathbb{R}$: $2n-1$	$\mathbb{C}$: n^2 $\mathbb{R}$: $2n^2+4n$ $\mathbb{Z}$: $5n$	$\mathbb{C}$: bn
BII	$\mathbb{C}$: $(b+1)n$ $\mathbb{R}$: $2n-1$	$\mathbb{C}$: n^2 $\mathbb{R}$: $5n$ $\mathbb{Z}$: $5n$	$\mathbb{C}$: bn
MRRR	$\mathbb{C}$: $(b+1)n$ $\mathbb{R}$: $2n-1$	$\mathbb{R}$: n^2+22n $\mathbb{Z}$: $10n$	$\mathbb{C}$: bn

Figure 2: A summary of workspace required to attain optimal performance for each suboperation of various algorithms employed in computing a Hermitian eigenvalue decomposition. Expressions denoted by $\mathbb{R}$, $\mathbb{C}$, and $\mathbb{Z}$ refer to the amount of real, complex, and integer workspace required. Some suboperations, such as forming or applying Q, may be executed with less workspace than indicated, but at a cost of significantly lower performance.

algorithm, known as the *dqd* algorithm [18, 35], but suggest that it is avoided only because it is, in practice, found to be "very slow."[2]

It goes without saying that some applications are dependent upon very high accuracy, even if these algorithms are slower for many classes of matrices. Therefore, realizing a QR algorithm that is restructured for high-performance becomes even more consequential.

2.5 Cost comparison

The cost and workspace requirements for each step of the EVD are summarized in Figures 1 and 2. We use these estimates to predict the benefits of a hypothetical restructured QR algorithm that applies Givens rotations with level-3 BLAS performance, shown in Figure 3. For that figure we make the following assumptions: First, we assume that $K_1 \approx K_2$; Furthermore, since microprocessor cores are increasing in speed at a faster rate than memory architectures, and since level-1 and level-2 BLAS operations are memory-bound, K_1 and K_2 for future architectures will likely increase; The average number of Francis steps before deflation is taken as $k = 1.5$ and $\alpha = 0.5$ for D&C ; And we entirely ignore the $\mathcal{O}(n^2)$ cost of $T \rightarrow Q_T D Q_T^T$ in the MRRR-based algorithm. We omit BII from our results because (1) in the best case, its performance is similar to that of MRRR, and (2) because it has been shown to be numerically unreliable [11]. We can conclude that a high-performance QR algorithm can become competitive with EVD via D&C or MRRR, especially when K_1 and K_2 are large, and especially for the complex-valued case. The question now becomes how to achieve such high-performance.

3 Tridiagonal QR Algorithm Basics

We quickly review the basic ideas behind the (tridiagonal) QR algorithm for computing the EVD.

[2]Note that if the matrix generated by the application is already tridiagonal, then D&C and MRRR would be the methods of choice. If, additionally, only a few eigenvectors are desired and the application can accept the possibility of somewhat less accurate results, then MRRR may be preferred.

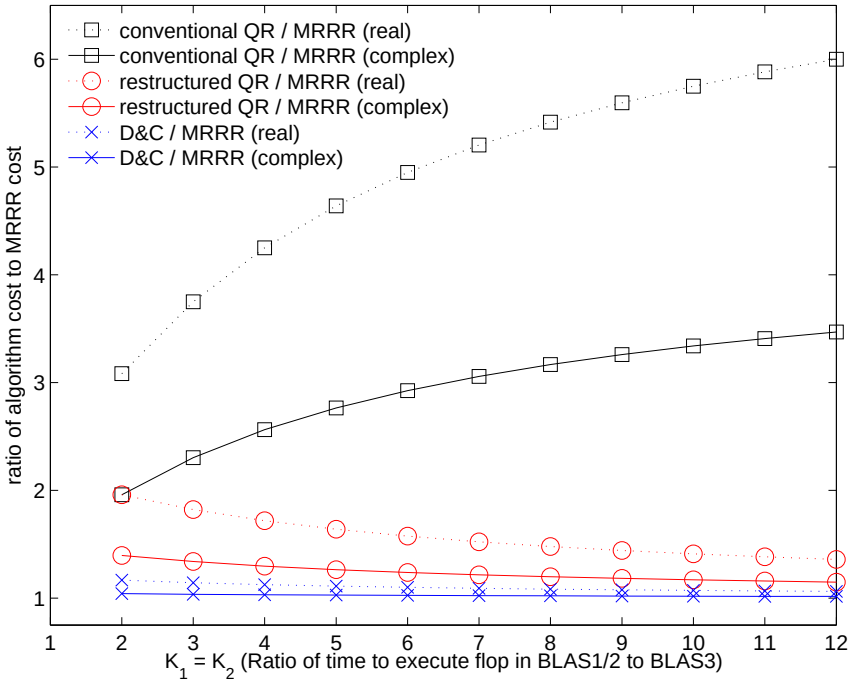

Figure 3: Predicted ratios of floating-point operation cost for various eigenvalue algorithms relative to the cost of Hermitian eigenvalue decomposition via MRRR.

3.1 Shifted tridiagonal QR algorithm

Given a tridiagonal matrix T, the shifted QR algorithm is given by

$$
\begin{array}{l}
V := I \\
\textbf{for } i = 0 : \text{convergence} \\
\quad \kappa_i := (\text{some shift}) \\
\quad T - \kappa_i I \rightarrow QR \\
\quad T^{\text{next}} := RQ + \kappa_i I \\
\quad V := VQ \\
\quad T := T^{\text{next}} \\
\textbf{endfor}
\end{array}
$$

Here, $T \rightarrow QR$ denotes a QR factorization. Under mild conditions this converges to a matrix of structure $\left(\begin{array}{c|c} T_B & 0 \\ \hline 0 & \lambda \end{array}\right)$, where λ is an eigenvalue of T. The process can then continue by *deflating* the problem, which means continuing the iteration with the tridiagonal matrix T_B. Eventually, T becomes a diagonal matrix, with the eigenvalues along its diagonal and V converges to a matrix with the corresponding eigenvectors (of the original matrix) as its columns.

Convergence occurs when one of the off-diagonal elements $\tau_{i+1,i}$ becomes negligible. Following Stewart [40], $\tau_{i+1,i}$ is considered to be negligible when $|\tau_{i+1,i}| \leq \varepsilon \sqrt{|\tau_{i,i}\ \tau_{i+1,i+1}|}$. Most often, deflation will occur in element $\tau_{n-1,n-2}$.

Two choices for shift κ_i are commonly used: (1) The Rayleigh Quotient shift — This method consists of choosing $\kappa_i = \tau_{n-1,n-1}$ (i.e., the bottom-right-most element of the current tridiagonal matrix), yields cubic convergence [40], but has not been shown to be globally convergent; and (2) The Wilkinson shift — This

shift chooses κ_i as the eigenvalue of $\begin{pmatrix} \tau_{n-2,n-2} & \tau_{n-1,n-2} \\ \tau_{n-1,n-2} & \tau_{n-1,n-1} \end{pmatrix}$ that is closest to $\tau_{n-1,n-1}$, converges at least quadratically, and is globally convergent [40]. We will use the Wilkinson shift, which in practice appears to yield faster results.

3.2 The Francis Implicit Q Step

One of the most remarkable discoveries in numerical linear algebra is the Francis Implicit Q Step, which provides a clever way of implementing a single iteration of the QR algorithm with a tridiagonal matrix [44]. Let $T - \kappa_i I$ equal

$$T = \left(\begin{array}{cc|ccccc} \tau_{0,0}-\kappa_i & \tau_{1,0} & 0 & 0 & \cdots & 0 \\ \tau_{1,0} & \tau_{1,1}-\kappa_i & \tau_{1,2} & 0 & \cdots & 0 \\ \hline 0 & \tau_{2,1} & \tau_{2,2}-\kappa_i & \tau_{3,2} & \ddots & \vdots \\ 0 & 0 & \tau_{3,2} & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \tau_{n-2,n-2}-\kappa_i & \tau_{n-1,n-2} \\ 0 & 0 & \cdots & 0 & \tau_{n-1,n-2} & \tau_{n-1,n-1}-\kappa_i \end{array}\right).$$

As part of the QR factorization of $T - \kappa_i I$, the first step is to annihilate $\tau_{1,0}$. For this purpose, we compute a *Givens rotation* $G_0^T = \begin{pmatrix} \gamma_0 & \sigma_0 \\ -\sigma_0 & \gamma_0 \end{pmatrix}$ that, when applied to the first two rows, annihilates $\tau_{1,0}$. (We label the rotations such that G_i^T is the rotation that affects rows i and $i+1$.) But rather than applying G_0^T to $T - \kappa_i I$, we instead apply it to the original T, affecting the first two rows, after which we apply G_0 from the right to the first two columns. This yields a matrix of the form

$$T = \left(\begin{array}{cc|ccccc} \tau_{0,0}^{\text{next}} & \star & \star & 0 & \cdots & 0 \\ \star & \star & \star & 0 & \cdots & 0 \\ \hline \star & \star & \tau_{2,2} & \tau_{3,2} & \ddots & \vdots \\ 0 & 0 & \tau_{3,2} & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \tau_{n-2,n-2} & \tau_{n-1,n-2} \\ 0 & 0 & \cdots & 0 & \tau_{n-1,n-2} & \tau_{n-1,n-1} \end{array}\right).$$

where the "$\star$"s represent non-zeroes. This step of computing G_0^T from $T - \kappa_i I$ and applying it to T is said to "introduce the bulge" because it causes a non-zero entry to appear below the first subdiagonal (and, because of symmetry, above the first superdiagonal), which we have highlighted above. Next, we compute a rotation G_1 that will annihilate element $(2,0)$ of T and apply it to the second two rows of the updated matrix, and its transpose to the second two columns. This begins the process of "chasing the bulge" and causes the non-zero entry to reappear at element $(3,1)$:

$$T = \left(\begin{array}{cc|ccccc} \tau_{0,0}^{\text{next}} & \tau_{1,0}^{\text{next}} & 0 & 0 & \cdots & 0 \\ \tau_{1,0}^{\text{next}} & \tau_{1,1}^{\text{next}} & \star & \star & \cdots & 0 \\ \hline 0 & \star & \star & \star & \ddots & \vdots \\ 0 & \star & \star & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \tau_{n-2,n-2} & \tau_{n-1,n-2} \\ 0 & 0 & \cdots & 0 & \tau_{n-1,n-2} & \tau_{n-1,n-1} \end{array}\right).$$

This process continues, computing and applying rotations $G_0, G_1, G_2, \ldots, G_{n-2}$ in this manner (transposed from the left, without transpose from the right) until the matrix is again tridiagonal. Remarkably, the T that results from this process is exactly the same as the T^{next} in the original algorithm.

This process is known as a Francis Implicit Q Step (or, more concisely, a Francis step). Neither Q nor R nor T is ever explicitly formed.

3.3 Accumulating the eigenvectors

If we initially let $V = I$, then Q_T can be computed as follows. First, we partition V into its columns: $V = \begin{pmatrix} v_0 & v_1 & \cdots & v_{n-1} \end{pmatrix}$. For each Francis step, for $j = 0, \ldots, n-2$, apply the rotations G_j to columns v_j and v_{j+1}, in the order indicated by

$$\begin{array}{ccccccccccccccccc} & & G_0 & \rightarrow & G_1 & \rightarrow & G_2 & \rightarrow & G_3 & \rightarrow & G_4 & \rightarrow & G_5 & \rightarrow & G_6 & \rightarrow & G_7 & \rightarrow & G_8 \\ v_0 & & & v_1 & & v_2 & & v_3 & & v_4 & & v_5 & & v_6 & & v_7 & & v_8 & & v_9 \end{array}$$

Here, the arrows indicate the order in which the Givens rotations must be applied while the two columns situated below each rotation represent the columns v_j and v_{j+1} to which that rotation is applied. We refer to $\{G_0, \ldots, G_{n-2}\}$ as a single Francis *set* of rotations. Following this process, after the QR iteration is complete and T has converged to D (i.e., the eigenvalues of T), $V = Q_T$. That is, column v_j contains the eigenvector of T that corresponds to the eigenvalue $\delta_{j,j}$.

If we initialize $V := Q_A$ and the rotations computed from each Francis step are applied to V as previously described, then upon completion of the QR iteration $V = Q$, where column v_j contains the eigenvector of A associated with the eigenvalue $\delta_{j,j}$.

The application of a Givens rotation to a pair of columns of Q requires approximately $6n$ flops ($2 \times 6n$ flops for complex V). Thus, applying $n-1$ rotations to n columns costs approximately $(2\times)6n^2$ flops. Since performing the Francis step without applying to Q takes $\mathcal{O}(n)$ computation, it is clearly the accumulation of eigenvectors that dominates the cost of the QR algorithm.

3.4 LAPACK routines {sdcz}steqr

The tridiagonal QR algorithm is implemented in LAPACK in the form of the subroutines `ssteqr` and `dsteqr` (for single- and double- precision real matrices, respectively) and `csteqr` and `zsteqr` (for single- and double-precision complex matrices, respectively). These routines implement the entire algorithm described in Sections 3.1–3.3 in a single routine[3]. In these implementations, Givens rotations from a single Francis step are saved in temporary storage until the step is complete. These rotations, a Francis set, are then applied to Q_A before moving on to the next Francis step. The application of the Givens rotations to the columns of Q_A is implemented via calls to {sdcz}lasr, which provide flexible interfaces to routines that are very similar to the level-1 BLAS routines {sd}rot. This, inherently, means that LAPACK's {sdcz}steqr routines do not benefit from the fast cache memories that have been part of modern microprocessors since the mid-1980s. Thus, accumulating eigenvectors is costly not only because of total *number* of flops, but also because those flops are performed at a very slow *rate*.

4 Applying Givens Rotations

We now analyze the application of Givens rotations to a matrix.

4.1 The basic idea

Recall from Section 3.3 that applying one Francis set of $(n-1)$ rotations to the $n \times n$ matrix V in which eigenvectors are being accumulated requires approximately $6n^2$ flops (or $2 \times 6n^2$ for complex V). This requires that $(2\times)n^2$ floating-point numbers be loaded from and stored back to main memory at least once each, for an absolute minimum number of $(2\times)2n^2$ memory operations.[4] Thus, the ratio of flops to memops

[3] These routines, coded in Fortran-77, are likely direct translations from EISPACK, which dates back to the late 1960's. Evidence of this exists in the number of GO TO statements in the code.

[4] Henceforth, we will call the reading or writing of a floating-point number a memory operation, or memop.

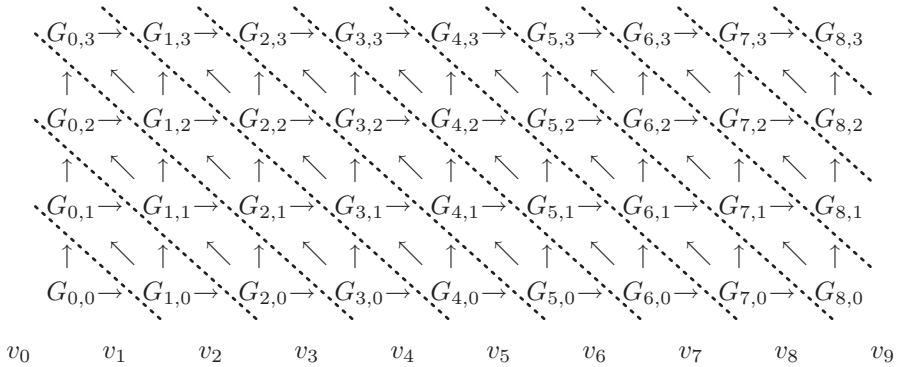

Figure 4: Applying Givens rotations from four Francis steps to a matrix V consisting of 10 columns. A rotation $G_{j,i}$ is generated during the ith Francis step and is applied to columns v_j and v_{j+1}. For any given rotation G, inbound arrows indicate dependencies that must be satisfied before the rotation may be performed. Temporal and spatial locality may be increased by applying rotations in diagonal "waves" from left to right (where the rotations in each wave are separated by dotted lines), and from bottom-right to top-left within each wave, according to the dependencies shown.

is at best 3. In practice, this ratio is even lower. For instance, the LAPACK routines for performing this operation, {sdcz}lasr perform (2×)6 flops for every (2×)4 memops, leaving them with a flop-to-memop ratio of just 1.5. This is a problem given that a memop typically requires much more time to execute than a flop, and the ratio between them is getting worse as microprocessor speeds outpace memory technology. We conclude that updating the matrix one Francis set at a time will not be efficient.

The simple insight behind this paper is that if one instead applies k Francis sets, this ratio's theoretical maximum rises to $3k$ flops for every memop. Evidence that this opportunity exists is presented in Figure 4. In this example, there are $k = 4$ Francis sets of rotations. The arrows indicate dependencies between rotations. The semantic meaning expressed by these arrows is simple: a rotation must have all dependencies (inbound arrows) satisfied before it may be performed. Notice that the drawing is a directed-acyclic graph (DAG) and hence a partial order. As long as the partial order is obeyed, the rotations are applied to columns in the *exact same order* as the traditional QR algorithm.

Furthermore, the dotted lines in Figure 4 expose "waves" of rotations that may be applied. The idea here is that waves of rotations are applied, from left to right, with rotations applied from bottom-right to top-left within each wave. (Notice that this pattern of applying rotations obeys the partial order.) It is easy to see that this wave-based approach increases temporal and spatial data locality. In fact, once the wavefront is fully formed, only one new column of V is brought into cache for each wave, and all the other columns needed for that wave are *already residing in the cache.*

It should be pointed out that we are not the first to highlight the need for optimized routines for applying Givens rotations. As part of the BLAST forum [4] there were requests for a routine that would apply multiple sets of rotations to a matrix. However, a standard interface never materialized, and so library maintainers were never sufficiently motivated to include the operation in their BLAS libraries.

4.2 Details of cache behavior

Let us perform a more detailed (but still simple) analysis of the cache behavior of applying Givens rotations, as described in the previous section. Let us assume we have a fully-associative cache that uses the least-recently used replacement policy, and that the number of elements in our cache, n_C, is significantly less than n^2 but is greater than kn, where $k \ll n$. Also, let us ignore the fact that accessing data of unit stride often produces cache hits due to entire cache lines being loaded from memory. We justify ignoring this effect by pointing out that the scalar-level hit and miss pattern would be identical for all column-oriented algorithms and thus would not affect our analysis. The idea is that we wish to abstract away to hits and misses of

entire *columns* of $V = Q_A$, given that applying Givens rotations inherently means touching entire columns of data. Thus, from now on, in the context of analyzing the application of Givens rotations, we will only count *column*-accesses, which refer to the memory operations needed to load or store n consecutive elements.

Let us begin by considering how we apply the first of k Francis sets of Givens rotations using the single-step approach employed by the traditional QR algorithm. Applying the first set of rotations accesses most of the n columns of V twice, for a total of $2(n-1)$ column-accesses. Of these accesses, n are cache misses and $n-2$ are cache hits. Now, we assume that since $n_C \ll n^2$, upon finishing applying first Francis set and beginning the next, columns $v_0, v_1, v_2, \ldots$ have long since been evicted from the cache. Thus, there is no data reuse *between* sets of Givens rotations using the single-step approach found in LAPACK, resulting in kn cache misses and $k(n-2)$ cache hits, for a hit rate of approximately 50%.

We now highlight a simple observation. *If multiple sets of rotations are accumulated, the order in which they are applied can be modified to improve the reuse of data in the caches.*

Consider the approach exemplified by Figure 4. Upon inspection, we can identify three distinct "stages" of the wavefront algorithm. For future reference, let us define these phases as follows:

- The *startup phase* refers to the initial $k-1$ waves of rotation applications where the "wavelength" (the number of rotations within the wave) is initially 1 and gradually increases.
- The *pipeline phase* refers to the middle $n-k$ waves of rotation applications where each wave is of length k.
- The *shutdown phase* refers to the final $k-1$ waves, where the wavelength becomes less than k and gradually decreases to 1.

Let us now more closely analyze the cache behavior of this algorithm. The wavefront algorithm illustrated in Figure 4 produces $k \times 2(n-1) = 2k(n-1)$ column-accesses. By straightforward reasoning, we find that, of these accesses, there are $2 + (k-2) = k$ cache misses in the startup phase, $(n-1) - (k-1) = n-k$ cache misses in the pipeline phase, and no cache misses in the shutdown phase, for a total of n cache misses. This means the ratio of cache misses to total accesses is approximately $\dfrac{n}{2k(n-1)} \approx \dfrac{1}{2k}$, and the fraction of cache hits is $1 - \dfrac{1}{2k}$. For larger values of k, this cache hit ratio is quite close to 1, and as a result, the wavefront algorithm should exhibit very favorable data access patterns.

4.3 A wavefront algorithm

In this section we will give several algorithms for applying Givens rotations, leading up to an algorithm for wavefront application of k sets of rotations to an $m \times n$ matrix. To our knowledge, this wavefront algorithm is previously unpublished.

First, let us define a helper algorithm ApplyGmx2, shown in Figure 5, which applies a single rotation $G = \begin{pmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{pmatrix}$ from the right to a pair of vectors x and y of length m. The rotation is never explicitly formed. Rather, it is represented and passed in by the pair of Givens scalars $\{\gamma, \sigma\}$. Also, notice that ApplyGmx2 avoids any computation if the rotation is equal to identity.

Let us now formulate, for reference, an algorithm ApplyGsingle that, given an $(n-1) \times k$ matrix G of Givens scalar pairs, applies the sets of Givens rotations in single-step fashion to an $m \times n$ matrix V. We show this algorithm in Figure 6.

Figure 7 shows a wavefront algorithm that applies the Givens rotations stored in an $(n-1) \times k$ matrix G of Givens scalars to an $m \times n$ matrix V. This algorithm implements the exact sequence of rotation application captured by the drawing in Figure 4. Each outer loop iterates through a series of waves, while the inner loops iterate through the rotations within each wave. Note that we simply call ApplyGsingle if there are not enough columns in V to at least fully execute the startup and shutdown phases (ie: if $n < k$).

Algorithm: $[\ x,\ y\ ]$:=ApplyGmx2(m, $\{\gamma, \sigma\}$, x, y)

if($\gamma = 1$) **return fi**
for($i := 0$; $i < m$; $++i$)

$$\begin{pmatrix} \chi_i & \psi_i \end{pmatrix} := \begin{pmatrix} \chi_i & \psi_i \end{pmatrix} \begin{pmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{pmatrix}$$

endfor

Figure 5: An algorithm for applying a single Givens rotation, defined by a pair of Givens scalars $\{\gamma, \sigma\}$, to two length m columns x and y from the right.

Algorithm: $[\ V\ ]$:=ApplyGsingle(k, m, n, G, V)

for($h := 0$; $h < k$; $++h$)
  for($j := 0$; $j < n - 1$; $++j$)
    ApplyGmx2(m, $G_{j,h}$, v_j, v_{j+1})
  endfor
endfor

Figure 6: An algorithm for applying the Givens rotations contained within an $(n-1) \times k$ matrix G of Givens scalar pairs, in single-step fashion, to an $m \times n$ matrix V.

Algorithm: $[\ V\ ]$:=ApplyGwave(k, m, n, G, V)

if($n < k$ OR $k = 1$)
  V = ApplyGsingle(k, m, n, G, V)
  return
endif
for($j := 0$; $j < k - 1$; $++j$) *// Startup phase*
  for($i := 0$, $g := j$; $i < j + 1$; $++i$, $--g$)
    $[v_g, v_{g+1}]$:= ApplyGmx2(m, $G_{g,i}$, v_g, v_{g+1})
  endfor
endfor
for($j := k - 1$; $j < n - 1$; $++j$) *// Pipeline phase*
  for($i := 0$, $g := j$; $i < k$; $++i$, $--g$)
    $[v_g, v_{g+1}]$:= ApplyGmx2(m, $G_{g,i}$, v_g, v_{g+1})
  endfor
endfor
for($j := n - k$; $j < n - 1$; $++j$) *// Shutdown phase*
  for($i := 1$, $g := n - 2$; $i < k$; $++i$, $--g$)
    $[v_g, v_{g+1}]$:= ApplyGmx2(m, $G_{g,i}$, v_g, v_{g+1})
  endfor
endfor

Figure 7: An algorithm for applying the Givens rotations contained within an $(n-1) \times k$ matrix G of Givens scalar pairs, in waves, to an $m \times n$ matrix V.

4.4 Blocking through V to control cache footprint

If applied to a matrix V with n rows, the wavefront algorithm requires approximately k columns to simultaneously fit in cache. Clearly, if k such columns do not fit, matrix V can be partitioned into blocks of rows and the k sets of rotations can be applied to one such block at a time.

Algorithm: [T, G] = TriQRAlgKSteps(i, k, n, T)

if($n \leq 1$ OR $i = k - 1$)
  return
else if($n = 2$)
  $[T, G_{0,i}]$:= SymEvd2x2(T)
  return
else
  if no deflation is possible
    Perform QR Francis step, computing $G_{0:n-2,i}$
    $[T, G_{0:n-2,i+1:k-1}]$:=TriQRAlgKSteps($i+1$, k, n, T)
  else
    $$T \rightarrow \begin{pmatrix} T_0 & 0 \\ 0 & T_1 \end{pmatrix} \text{ where } T_0 \text{ is } n_0 \times n_0 \text{ and } T_1 \text{ is } n_1 \times n_1$$
    $[T_0, G_{0:n_0-2,i:k-1}]$:= TriQRAlgKSteps(i, k, n_0, T_0)
    $[T_1, G_{n_0:n-2,i:k-1}]$:= TriQRAlgKSteps(i, k, n_1, T_1)
  endif
  return
endif

Figure 8: A recursive algorithm for computing k Francis steps, saving the rotations in G. It is assumed that G was initialized with identity rotations upon entry.

4.5 Why we avoid (for now) a hybrid QR/QL algorithm

There exist certain kinds of graded matrices for which the QR algorithm does not perform well. In particular, if a matrix is graded so that $|\tau_{n-1,n-1}| \gg |\tau_{0,0}|$, then the shifting of the matrix (computing $\tau_{0,0} - \tau_{n-1,n-1}$ wipes out the accuracy in $\tau_{0,0}$ and causes excessive roundoff error. For this reason, an implementation may choose to use a QL algorithm instead, in which $T - \kappa_i I \rightarrow QL$ and $T^{\text{next}} := LQ + \kappa_i I$. Indeed, depending on which of $\tau_{0,0}$ and $\tau_{n-1,n-1}$ is greater in magnitude, QR or QL may be used on an iteration-by-iteration basis.

Alternating between QR and QL steps disrupts our ability to accumulate and apply many rotations. Key to applying the wavefront algorithm is that all Francis steps move in the same direction through matrix T: in the case of the QR algorithm, from top-left to bottom-right. Clearly, if we *only* used the QL algorithm, we could formulate a similarly optimized algorithm with waves moving in the opposite direction. It would also be possible to periodically switch between multiple QR steps and multiple QL steps. However, frequent mixing of directions would limit our ability to reuse data in cache. It is for this reason that we don't alternate between QR and QL steps. As a result, this is the one part of the traditional implementation that we do not yet directly incorporate in our restructured algorithm.

5 A Restructured Tridiagonal QR Algorithm

In the last section, we discussed how the application of multiple Francis sets of rotations can, in principle, yield a performance boost. The problem is that the QR algorithm, as implemented in LAPACK, is not structured to produce multiple sets of rotations. In this section, we discuss the necessary restructuring.

5.1 The basic idea

The fundamental problem is that applying multiple sets of rotations is most effective if there are a lot of rotations to be applied. Now, take a possible state of T after some Francis steps have been performed:

$$T = \begin{pmatrix} T_0 & 0 & 0 \\ 0 & T_1 & 0 \\ 0 & 0 & T_2 \end{pmatrix}.$$

Algorithm: $[\ T,\ V\ ]$:=RESTRUCTUREDTRIQR(b, k, n, T, V)

do

Partition $T \rightarrow \begin{pmatrix} T_0 & 0 & \cdots & 0 & 0 \\ 0 & T_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & T_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & D \end{pmatrix}$, $G \rightarrow \begin{pmatrix} G_0 \\ g_0 \\ \vdots \\ G_{N-1} \\ g_{N-1} \\ G_N \end{pmatrix}$

where

T_i is $n_i \times n_i$ and unreducible,
D is $n_N \times n_N$ and diagonal,
n_N is as large as possible,
G_i is $(n_i - 1) \times k$, and
g_i is $1 \times k$.

if($n_N = n$) **break**

for($i = 0;\ i < N;\ ++i$)

$[T_i, G_i]$:= TRIQRALGKSTEPS(0, k, n_i, T_i)

endfor

$[V]$:= APPLYGWAVEBLK(b, k, n, n, G, V)

od

return

Figure 9: Our restructured QR algorithm for the tridiagonal EVD.

What we capture here is the fact that deflation may happen before k Francis steps have completed. What a typical QR algorithm would do is continue separately with each of the diagonal blocks, but that would disrupt the generation of all rotations for all k Francis steps with the original matrix T. We still want to take advantage of deflation. Essentially, if deflation can occur and the number of Francis steps completed so far is $i < k$, then for each block on the diagonal we perform the remaining $k - i$ Francis steps, while continuing to accumulate rotations. After the k rotation sets are accumulated for all deflated subproblems along the diagonal of T, the rotations are applied to V. A recursive algorithm that achieves this is given in Figure 8. What is left is an outer loop around this routine, as given in Figure 9.

5.2 Restructured QR algorithm details

Now that we have given a brief, high-level description for the restructured tridiagonal QR algorithm, let us walk through the algorithm in further detail.

We start with the RESTRUCTUREDTRIQR algorithm in Figure 9. The first time through the outer loop, it is assumed that T has no negligible (or zero) off-diagonal entries. Thus, $N = 1$, $n_0 = n$, $n_N = 0$, and as a result T is partitioned such that T_0 contains all of T and D is empty. Likewise, G is partitioned into just the $(n-1) \times k$ subpartition G_0, which contains all of G. (Note that, once deflation occurs, the reason we separate G_i from G_{i+1} by a single row vector g_i is because this vector corresponds to the rotations that would have been applied to the 2×2 submatrix starting at element $(n_i - 1, n_i - 1)$ along the diagonal of T if deflation had not already occurred there.) Now, since $N = 1$, the inner loop executes just once, calling TRIQRALGKSTEPS with $T_0 = T$ and passing in $i = 0$.

Within TRIQRALGKSTEPS, we return if T is empty or a singleton. (We also return if $i = k - 1$, which happens when we have accumulated the Givens scalars for k Francis sets.) If $n = 2$, we compute the EVD of T via SYMEVD2X2, which returns the eigenvalues λ_1 and λ_2 along the diagonal of T and a pair $\{\gamma, \sigma\}$ (stored to G) such that $T = \begin{pmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{pmatrix}^T$. Otherwise, if $n > 2$, we search the off-diagonal of T for negligible elements. If we find a negligible element, then we may deflate T into top-left

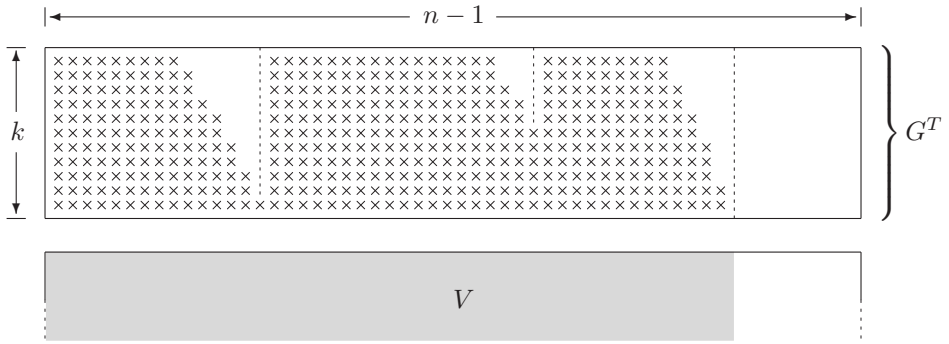

Figure 10: An illustration of the possible contents of $(n-1) \times k$ matrix G at the end of a sweep, just before its rotations are applied to matrix V. (G is shown transposed to better visualize the portion of V (highlighted) that is affected by the application.) Givens scalar pairs, which encode Givens rotations, are marked by "$\times$" symbols while unmarked locations represent identity rotations (which are skipped). This picture shows what G might look like after executing a sweep that encountered two instances of internal deflation: one in the first Francis step, which occurred while iterating over the entire non-deflated portion of T, and one in the sixth Francis step, which occurred within the lower-right recursive subproblem that began as a result of the first deflation.

and bottom-right partitions T_0 and T_1, which are $n_0 \times n_0$ and $n_1 \times n_1$, respectively, where $n_0 + n_1 = n$, and then call TriQRAlgKSteps recursively on these subpartitions, saving the resulting Givens scalar pairs to the corresponding rows of columns i through $k-1$. If no negligible off-diagonal elements are found, then we perform a Francis step on T, saving the Givens scalar pairs to the ith column of matrix G. We then proceed by recursively calling TriQRAlgKSteps with an incremented value of i.

At some point, we will fill matrix G with Givens scalar pairs, at which time control returns to RestructuredTriQR and ApplyGWaveBlk applies the rotations encoded in G to matrix V.

After the application, the outer loop repeats and, upon re-entering the inner loop, another "sweep" of T begins where up to k Francis steps are performed on each submatrix T_i before once again applying the rotations to V. This process repeats until the entire off-diagonal of T becomes negligible, at which time $n_N = n$, and control breaks out of the outer loop, signaling that the QR algorithm's execution is complete.

5.3 Impact of identity rotations

Notice that we initialize G to contain identity rotations ($\gamma = 1$ and $\sigma = 0$) prior to each entry into the inner loop of RestructuredTriQR. This allows Francis steps executed by TriQRAlgKSteps to simply overwrite elements of G with scalar pairs representing Givens rotations as they are computed, without having to worry about which elements are not needed due to deflation. Then, ApplyGWave simply detects these identity rotations and skips them so as to save computation.

Depending on how much deflation occurs, this method of storing rotations could have an impact on performance. We would expect that the wavefront algorithm for applying Givens rotations would execute most favorably when deflation occurs only (or mostly) at or near the bottom-right corner of T. This is because each column of V would be accessed and re-used a maximum number of times.

However, Figure 10 shows how internal deflation early on in a set of Francis steps can create "pockets" of identity rotations. Since identity rotations are skipped entirely, no computation is wasted. However, this would effectively cause temporary bursts of cache misses while the columns associated with subsequent non-identity rotations are touched and loaded into cache for the first time. The worst case arises when deflation occurs (1) somewhere in the middle of T, corresponding to the columns that would have been updated as part of full waves, and/or (2) very early on in the Francis set, as it gives the pocket of identity rotations the most time to grow due to subsequent tail deflation in the upper-left subproblem.

6 Optimizing the Application of Givens Rotations

In Section 4, we developed a wavefront algorithm for applying Givens rotations that is, in theory, capable of accessing data in a very cache-efficient manner. We also introduced a supplementary blocked algorithm to partition the eigenvector matrix V into smaller panels to further control the amount of data from V the algorithm tries to keep in cache without compromising the expected hit rate for column-accesses. This effectively induces a blocked algorithm for the operation, which constitutes a substantial algorithm-level improvement over a naive approach that applies one Francis set of rotations at a time to the entire matrix V. Even with these advances, further optimizations are possible.

6.1 Instruction-level optimizations

Most modern general-purpose microprocessors contain at least one SIMD instruction unit that may be used to execute floating-point instructions at a faster rate than the processor's general floating-point unit (FPU). The most commonly available instruction set for this purpose exists on Intel-compatible x86 microprocessors and is known as the Streaming SIMD Extensions, or SSE. The instruction set has gone through multiple revisions since its original inception in the late 1990's, including SSE2 in late 2000 which added support for double-precision floating-point arithmetic. It is typically the case that, in order to get the best performance and make the best use of memory loads and stores, one must program with a vector instruction set such as SSE. Unfortunately, the most direct way to use SSE is typically by programming in assembly language, which most in the scientific computing community avoid if at all possible since assembly code is exceedingly difficult for humans to read.

Thankfully, there is a way to make use of vector instruction sets from a higher level language such as C. Both the Intel and GNU C compilers support low-level vector APIs known as "vector intrinsics." These basic programming interfaces allow the software developer to precisely specify when to load and store data. Furthermore, in the case of SSE, the standard floating-point operators such as addition, subtraction, multiplication, and division, are overloaded to operate on special datatypes representing vectors of floating-point data as stored in registers. This allows the programmer to achieve high performance with relatively minor source code changes, which are often isolated to an application's inner-most computational kernels.

To illustrate the use of SSE vector intrinsics, we give two example codes in Figure 11 which implement the ApplyGmx2 algorithm shown in Figure 5. The `applygmx2` function on the left shows how a programmer might code the ApplyGmx2 operation using only basic C constructs and operations. Note that pointer arithmetic is used instead of array indexing notation.

The `applygmx2_vec` function implements a similar function, except that it has been modified to employ SSE vector intrinsics. We briefly discuss the changes:

- **Header support.** Note that the header file `pmmintrin.h` is included. This pulls in various type definitions and intrinsic prototypes for SSE support, up to the SSE3 revision.
- **Vector type declarations.** The function `applygmx2_vec` declares five vector objects, `gv`, `sv`, `xv`, `yv`, and `tv`, each of which is capable of storing two double-precision floating-point values. These are the objects through which we will interface with the SSE instruction set.
- **Modified vector increments and loop bound.** Since we will be invoking SSE instructions that operate on two `double` values at a time, we will need to increase the stride through `x` and `y` by a factor of two. Similarly, we need to adjust the loop bound since the number of iterations will be (roughly) halved. We must now handle the case where m is odd with a special code block after the main loop.
- **Vector intrinsic usage.** Prior to entering the loop, we load the values `gamma` and `sigma` and duplicate each to both locations within `gv` and `sv`, respectively. We do this because SSE math operations typically operate on an element-wise basis. Within the loop, we load the current 2×1 vectors of elements at `xp` and `yp` into `xv` and `yv`, respectively. We then specify computation with the vectors, as we might in a higher-level language. Finally, we indicate that the values held in the `xv` and `yv` vectors must be stored back to the consecutive memory elements at `xp` and `yp`, respectively.

```
void applygmx2( int     m,
                double  gamma,
                double  sigma,
                double* x,
                double* y )
{
   double* restrict xp = x;
   double* restrict yp = y;
   double           tmp;
   int              i;

   if ( gamma == 1.0 ) return;

   for( i = 0; i < m; ++i )
   {
      tmp = *xp;

      *xp = gamma *  tmp  + sigma * (*yp);
      *yp = gamma * (*yp) - sigma *  tmp;

      xp += 1; yp += 1;
   }
}
```

```
#include "pmmintrin.h"

void applygmx2_vec( int     m,
                    double  gamma,
                    double  sigma,
                    double* x,
                    double* y )
{
   double* restrict xp = x;
   double* restrict yp = y;
   double           tmp;
   int              i;
   int              m_iter = m / 2;
   int              m_left = m % 2;
   __m128d          gv, sv, xv, yv, tv;

   if ( gamma == 1.0 ) return;

   gv = _mm_loaddup_pd( &gamma );
   sv = _mm_loaddup_pd( &sigma );

   for ( i = 0; i < m_iter; ++i )
   {
      xv = _mm_load_pd( xp );
      yv = _mm_load_pd( yp );

      tv = xv;
      xv = gv * tv + sv * yv;
      yv = gv * yv - sv * tv;

      _mm_store_pd( xp, xv );
      _mm_store_pd( yp, yv );

      xp += 2; yp += 2;
   }

   if ( m_left == 1 )
   {
      tmp = *xp;

      *xp = gamma *  tmp  + sigma * (*yp);
      *yp = gamma * (*yp) - sigma *  tmp;
   }
}
```

Figure 11: Two C implementations for computing an ApplyGmx2 operation: one that uses regular C constructs (left), and one that uses SSE vector instrinsics to precisely specify the loading and storing of values to and from main memory (right).

As straightforward as it may appear, there are some issues that must be carefully addressed. For example, the particular load and store SSE instructions used by `applygmx2_vec` require that the memory locations being loaded from (or stored to) be aligned to a 16-byte boundary. A practical implementation must either guarantee that all data begins at an aligned memory address or handle the mis-aligned data with conditional "edge" cases. (Notice that we do *not* handle mis-aligned data in our example.)

Despite these low-level issues, vector intrinsics provide a relatively accessible mechanism for taking advantage of the vector floating-point units in modern processors. Using just basic load/store intrinsics, we can coerce the compiler into emitting code that exhibits very high performance relative to code that does not use streaming instructions. For operations that are rich in computation relative to the size of the data on which they operate, such as the application of Givens rotations, these vector intrinsics should allow one to achieve performance that approaches that of highly-optimized level-3 BLAS operations, such as general matrix-matrix multiplication (GEMM).

Next we will show how precisely controlling the loading and storing of data to and from main memory opens the door to further algorithmic-level optimizations within the application of Givens rotations.

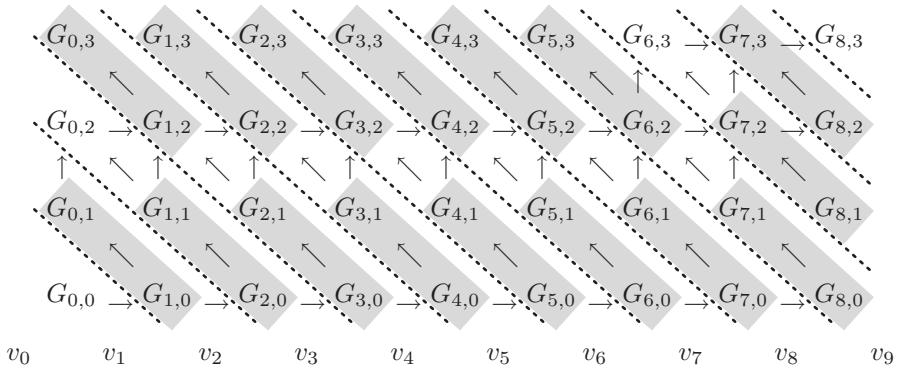

Figure 12: By fusing pairs of successive rotations within the same wave, the number of memory operations performed on each element of V, per rotation applied, may be reduced from approximately four to three. Here, fused rotations are highlighted, with arrows redrawn to show dependencies within a fused pair as well as dependencies between pairs. Rotation pairs may be applied in waves, similar to the approach illustrated by Figure 4. Note that the updated dependencies satisfy the original dependency graph.

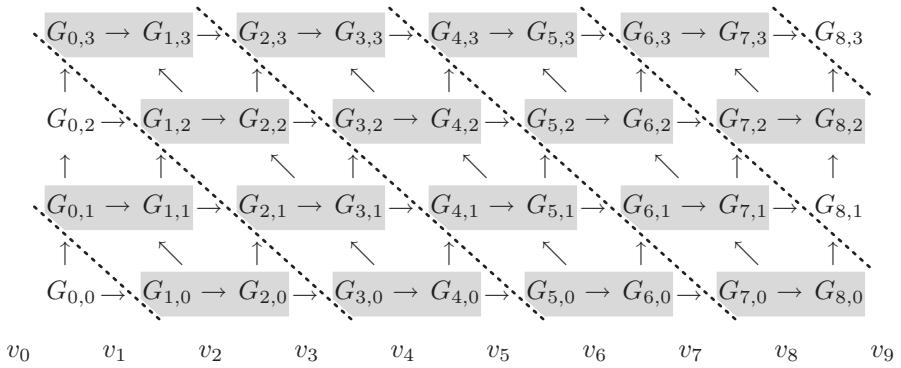

Figure 13: Instead of fusing successive rotations within the same wave, as shown in Figure 12, we may also fuse pairs of successive rotations from the same Francis step. This approach similarly reduces the number of memory operations performed on each element of V per rotation from four to three.

6.2 Fusing rotations

In [43] the authors showed how certain level-2 BLAS operations could be "fused" to reduce the total number of memory accesses, which tend to be rather slow and expensive relative to floating-point computation. They further showed that fusing two operations at the register-level—that is, re-using the data while it is still loaded in the register set—is superior to a cache-level approach whereby we simply assume that recently touched data in the cache will be readily available. The bottom line: all else being equal, reducing the number of memory operations, *regardless of whether the data is already in cache*, will typically allow shorter execution times and thus higher performance.

Consider Figure 12. Here, we have made the observation that successive Givens rotations within a single wave may be fused to reduce the number of times the rotations must access their corresponding columns of V. (Dependency arrows have been redrawn to indicate dependencies within pairs of rotations as well as dependencies between pairs.) Let us consider one such fusable pair, $G_{1,0}$ and $G_{0,1}$, in detail. When no fusing is employed, $G_{1,0}$ will load and store each element of v_1 and v_2, for a total of four column-accesses. Immediately following, $G_{0,1}$ will load and store v_0 and v_1, for another four column accesses. However, *any given element* of v_1 does not change in between the time that $G_{1,0}$ stores it and $G_{0,1}$ loads it. Thus, we could fuse the application of rotations $G_{1,0}$ and $G_{0,1}$. This would result in the ith element of v_1 and the ith

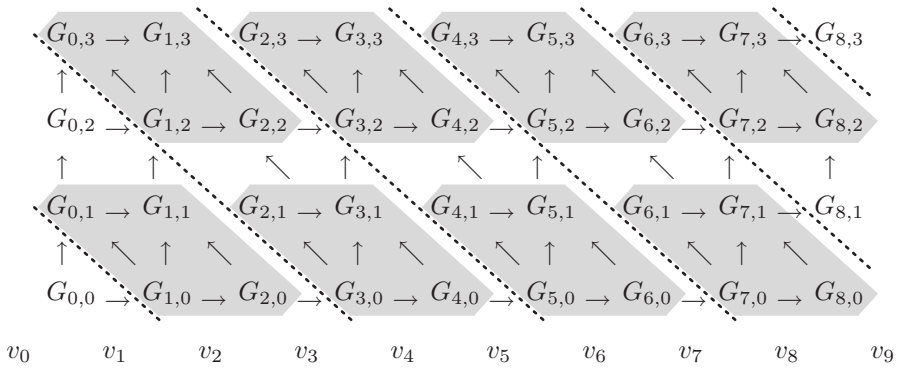

Figure 14: Combining the approaches from Figures 12 and 13, we can fuse in both the k and n dimensions. This creates parallelogram-shaped groups of fusings. This two-dimensional fusing reduces the average number of column-access memory operations per rotation from four, in an algorithm that uses no fusing, to just two.

element of v_2 first being loaded (by $G_{1,0}$). After computing the updated values, *only* element i of v_2 is stored while the ith element of v_1 is retained in a register. At this point, $G_{0,1}$ need only load the corresponding ith element from v_0 in order to execute. After its computation, $G_{0,1}$ stores the ith elements of both v_0 and v_1. This reduces the total number of column-accesses needed to apply two rotations from eight to six. Put another way, this reduces the number of column-accesses needed per rotation from four to three—a 25 percent reduction in memory operations. Likewise, this increases the ratio of total flops to total memops from 3, as established in Section 4.1, to 4.

We can express the number of memory accesses in terms of the level of fusing, as follows. If we fuse p rotations in a single wave where $p \ll k$, then the approximate number of total column-accesses incurred is given by $2kn\left(1+\frac{1}{p}\right)$, and thus with $k(n-1)$ total rotations to apply, the number of column-accesses needed per rotation is approximated by $2\left(1+\frac{1}{p}\right)$. Notice that aggressively increasing p results in diminishing improvement in the number of memory accesses.

If we turn our attention back to Figure 4, we can find another method of fusing rotation pairs. This method is illustrated in Figure 13. Instead of fusing successive rotations within the same wave, we can instead fuse successive rotations from the same Francis step. This approach, like the previous one, is similarly generalizable to fuse p rotations, and similarly reduces the number of column memory operations incurred per rotation to $2\left(1+\frac{1}{p}\right)$.

Now, let us revisit both Figures 12 and 13. Notice that the dependency arrows allow us to fuse rotations along both the k dimension (across Francis sets) and the n dimension (within a Francis set) of matrix G, as illustrated by Figure 14, to create parallelogram-shaped groups of fused rotations. By inspection, we find that the average number of column-accesses per rotation, n_{CAPR}, incurred by this more generalized method of fusing can be expressed as

$$n_{CAPR} \approx \frac{2(p_k + p_n)}{p_k p_n}$$

where p_k and p_n are the number of rotations fused in the k and n dimensions, respectively. Notice that for $p_k = p_n = 2$, the situation captured by Figure 14, this two-dimension fusing reduces the average number of memory operations required per rotation to just two column-accesses. This places the ratio of total flops to total memops at 6. Also notice that the optimal fusing shape is that of a "square" parallelogram (ie: $p_k = p_n$) because it corresponds to the classic problem of maximizing the area of a rectangle (the number of rotations) relative to its perimeter (the number of column-accesses).

In the limit, when $p_k = k$ and $p_n = n$, the ratio of total flops to memops rises to our theoretical maximum, $3k$. However, values of p_k and p_n higher than 2 are harder to implement because it implies that either (1) an increasing number of Givens scalars are being kept in registers, or (2) an increasing number of Givens scalars' load times are being overlapped with computation (a technique sometimes called "pre-fetching").

Since the processor's register set is typically quite small, either of these situations may be sustained only to a limited extent.

6.3 Pitfalls to fusing

In our experience, we have encountered one major caveat to fusing: if any of the rotations within a fused group is the identity rotation, then extra conditional logic must be introduced into the algorithm so as to not unnecessarily apply that rotation to V. This extra logic may add some overhead to the algorithm. In addition, the larger the fused group of rotations, the greater the chance that any given fused group will contain an identity rotation, which reduces opportunities for cache reuse during the execution of that group.

6.4 An alternate method of forming Q for complex matrices

Until now, we have only discussed Hermitian EVD whereby Givens rotations computed during the QR iteration are applied directly to $V = Q_A$ where $Q_A \in \mathbb{C}^{n \times n}$. However, there is another option that holds potential as long as the input matrix A remains complex.

Consider an algorithm for EVD in which we initialize $V_R = I$, where $V_R \in \mathbb{R}^{n \times n}$. Rotations may then be accumulated into V_R as before, which results in $V_R = Q_T$. Subsequently, Q may be formed by computing $Q := Q_A V_R$, which may be done with a general matrix-matrix multiplication where the left input matrix is complex and the right input matrix is real. While this kind of `dgemm` is not implemented in the BLAS, it is equivalent to multiplying V_R by the real and imaginary components of Q_A separately. If all matrices are stored in column-major order, this can be done all at once by using `dgemm` and specifying the m and leading dimensions of Q_A as twice their actual values, since the floating-point representation of complex numbers consists of two real-valued components. This requires that Q_A be copied to a temporary matrix $\hat{Q}_A$ so that `dgemm` can overwrite $Q_A := \hat{Q}_A V_R$. This approach has advantages and at least one major drawback.

The first advantage of this approach is that part of the computation is now shifted to `dgemm`, which is capable of near-peak performance on many modern architectures [20] (whereas applying Givens rotations is inherently limited to 75% of peak on some architectures, as discussed later in Section 9.2).

The second advantage is that this method incurs fewer floating-point operations when the number of Francis steps performed per deflation, k, is above a certain threshold. As discussed earlier in this paper, applying rotations directly to $V = Q_A$ leads to approximately $2 \times 3kn^3$ flops. However, accumulating the rotations to a real matrix V_R results in only $3kn^3$ flops. The subsequent invocation of `dgemm` incurs an additional $2(2n)n^2 = 4n^3$ flops. Thus, this approach incurs fewer flops when $3kn^3 + 4n^3 < 2 \times 3kn^3$, or, when $k > \frac{4}{3}$. However, additional flops may be saved during the accumulation of V_R. Notice that after applying the first Francis set of rotations to $V_R = I$, the matrix is filled in down to the first subdiagonal with the remaining entries containing zeros. Applying the second Francis set results in one additional subdiagonal of fill-in. By exploiting this gradual fill-in, we can save approximately n^3 flops, reducing the total number of Givens rotation flops from $3kn^3$ to approximately $(3k - 1)n^3$. With these flop savings, the method now incurs fewer flops when $k > 1$, which is almost always the case.

The most obvious disadvantage of this approach is that it requires $\mathcal{O}(n^2)$ workspace. Specifically, one would need to allocate a real $n \times n$ matrix V_R to store the accumulated rotations. In addition, it would require a second complex $n \times n$ matrix to hold $\hat{Q}_A$. (Actually, one could get by with allocating a smaller $b \times n$ matrix for $\hat{Q}_A$, where $b \ll n$ but still large enough to allow `dgemm` to operate at high performance, and use a blocked algorithm to repeatedly perform several smaller panel-matrix multiplications.)

The idea of using matrix-matrix multiplication to accelerate rotation-based algorithms has been studied before [29, 36]. Most recently, in [28], it has been shown that it is possible to accumulate $2k \times 2k$ blocks of rotations and then use `dgemm` and `dtrmm` to multiply these blocks into $V = Q_A$. This method uses only $\mathcal{O}(n)$ workspace. However, this method incurs roughly 50% more flops than our plain restructured QR algorithm due to overlap between the accumulated blocks. This method would become advantageous when the ratio of observed performance between ApplyGWaveBlk and `dgemm` (and/or `dtrmm`) is less than $\frac{2}{3}$. Otherwise, we would expect this method to exhibit longer run times than that of applying rotations directly to $V = Q_A$ via the blocked wavefront algorithm.

7 Numerical Accuracy

Ordinarily, this kind of paper would include extensive experiments to measure the accuracy of the new approach and compare it to that of other algorithms. However, because the computations performed by our restructured QR algorithm, and the order in which they are performed, are virtually identical to those of the original implementation in LAPACK (modulo the case of certain highly graded matrices), it follows that the algorithm's numerical properties are preserved, and thus the residuals $\|Av_j - \lambda_j v_j\|$ are small. Furthermore, the accuracy and convergence properties of the tridiagonal QR algorithm relative to other algorithms are well-studied subjects [45, 40, 10].

It is worth pointing out that even in the case of graded matrices, there are certain cases where the QR algorithm performs well. Specifically, if the matrix grading is downward along the diagonals, the QR algorithm typically yields eigenvalues and eigenvectors to high relative accuracy [40].

8 Extending to Singular Value Decomposition

Virtually all of the insights presented in this paper thus far may be easily applied to the SVD of a dense matrix.

Given a general matrix $A \in \mathbb{C}^{m \times n}$, its singular value decomposition is given by $A = U\Sigma V^H$, where $U \in \mathbb{C}^{m \times m}$, $V \in \mathbb{C}^{n \times n}$, both U and V are unitary, and $\Sigma \in \mathbb{R}^{m \times n}$ is positive and diagonal. The singular values of A can then be found on the diagonal of Σ while the corresponding left and right singular vectors reside in U and V, respectively.

Computing the SVD is similar to the three stages of dense Hermitian EVD [40]: Reduce matrix A to real bidiagonal form B via unitary similarity transformations: $A = U_A B V_A^H$; Compute the SVD of B: $B = U_B \Sigma V_B^T$; and back-transform the left and right singular vectors of B: $U = U_A U_B$ and $V = V_A V_B$ so that $A = U\Sigma V^H$.

If m is larger than n to a significant degree, it is often more efficient to first perform a QR factorization $A \rightarrow QR$ and then reduce the upper triangular factor R to real bidiagonal form [40], as $R = U_R B V_R^H$. The process then unfolds as described above, except that after the bidiagonal SVD (and back-transformation, if applicable), the matrix $U_R U_B$ must eventually be combined with the unitary matrix Q to form the left singular vectors of A.

Let us briefly discuss each stage.

Reduction to real bidiagonal form One can reduce a matrix to bidiagonal form in a manner very similar to that of tridiagonal form, except that here separate Householder transformations are computed for and applied to the left and right sides of A. In the (more common) case of $m \geq n$, where the matrix is reduced to *upper* bidiagonal form, these transformations annihilate the strictly lower triangle and all elements above the first superdiagonal. Families of algorithms for reducing a matrix to bidiagonal form are given in [42].

Computing the SVD of a real bidiagonal matrix Computing the SVD of a real bidiagonal matrix can be performed via Francis steps similar to those discussed in Section 3.2, except that: (1) the shift is typically computed from the smallest singular value of the trailing 2×2 submatrix along the diagonal [40]; (2) the bulge in the Francis step is chased by computing and applying separate Givens rotations from the left *and* the right; and (3) the left and right Givens rotations are similarly applied to left and right matrices which, upon completion, contain the left and right singular vectors of the bidiagonal matrix B. Also, the test to determine whether a superdiagonal element is negligible is somewhat different than the test used in the tridiagonal QR algorithm. Details concerning the negligibility test, along with other topics related to computing singular values, may be found in [9]. Others have successfully developed Divide and Conquer [21] and MRRR [47] algorithms for computing the singular value decomposition of a bidiagonal matrix. The most recent release of LAPACK (version 3.3.1 as of this writing) includes an implementation of dense matrix SVD based on the D&C algorithm; however, an MRRR implementation has not yet been published in the library. Many algorithmic variants of the SVD exist, including several based on the QR algorithm [7], but for practical purposes we limit our consideration to those algorithms implemented in LAPACK.

SVD Algorithm	Effective higher-order floating-point operation cost		
	$A \rightarrow U_A B V_A^H$	$B \rightarrow U_B \Sigma V_B^T$	Form/Apply U, V
orig. QR	$4 \times (K_2 + 1)$ $(2mn^2 - \frac{2}{3}n^3)$	$2 \times K_1 3kn^2(m+n)$	$4 \times 4n(m^2 - mn + \frac{2}{3}n^2)$
rest. QR	$4 \times (K_2 + 1)$ $(2mn^2 - \frac{2}{3}n^3)$	$2 \times 3kn^2(m+n)$	$4 \times 4n(m^2 - mn + \frac{2}{3}n^2)$
D&C	$4 \times (K_2 + 1)$ $(2mn^2 - \frac{2}{3}n^3)$	$\mathcal{O}(mn + n^2)$	$4 \times 4n(m^2 - \frac{1}{2}mn + \frac{1}{2}n^2)$

Figure 15: A summary of approximate floating-point operation cost for suboperations of various algorithms for computing a general singular value decomposition. Integer scalars at the front of the expressions (e.g. "$4 \times \ldots$") indicate the floating-point multiple due to performing arithmetic with complex values. In the cost estimates for the QR algorithms, k represents the typical number of Francis steps needed for an eigenvalue to converge. For simplicity, it is assumed that $m \geq n$, but also that it is *not* the case that $m >> n$ and thus an initial QR factorization of A is *not* employed.

SVD Algorithm	Workspace required for optimal performance		
	$A \rightarrow U_A B V_A^H$	$B \rightarrow U_B \Sigma V_B^T$	Form/Apply U, V
original QR	$\mathbb{C}$: $2n + b(m+n)$ $\mathbb{R}$: $2n - 1$	$\mathbb{R}$: $4(n-1)$	$\mathbb{C}$: $b(m-1)$
restructured QR	$\mathbb{C}$: $2n + b(m+3n)$ $\mathbb{R}$: $2n - 1$	$\mathbb{R}$: $4b(n-1)$	$\mathbb{C}$: $b(m-1)$
D&C	$\mathbb{C}$: $2n + b(m+n)$ $\mathbb{R}$: $2n - 1$	$\mathbb{R}$: $3n^2 + 4n$ $\mathbb{Z}$: $8n$	$\mathbb{C}$: bm

Figure 16: A summary of workspace required to attain optimal performance for each suboperation of various algorithms employed in computing a general singular value decomposition. For simplicity, it is assumed that $m \geq n$, but also that it is *not* the case that $m \gg n$ and thus an initial QR factorization of A is *not* employed.

Back-transformation of U and V The back-transformation of U and V take place in a manner similar to that of Q in the EVD, except that when $m > n$ only the first n columns of U_A are updated by U_B, and when $m < n$ only the first m columns of V_A are updated by V_B.

Figures 15 and 16 show floating-point operation counts and workspace requires, respectively, for the dense SVD problem. When counting workspace for SVD, we do not count either of the $m \times m$ or $n \times n$ output singular vector matrices as workspace. Here, we stray from our convention of only exempting the input matrices since it is not possible to output all singular vectors by overwriting the input matrix.

As with reduction to tridiagonal form, our reduction to bidiagonal form retains the triangular factors of the block Householder transformations, except here the operation requires an additional $2bn$ complex storage elements since we must retain the triangular factors for both U_A and V_A.

Note that the back-transformation of D&C has a "hidden" $\mathcal{O}(n^3)$ cost relative to the QR algorithm that is very similar to the hidden cost found in the D&C and MRRR algorithms for computing the Hermitian EVD. Actually, when $m = n$, the hidden cost is $\frac{4}{3}n^3$, which is twice as large as that of the EVD because the SVD must back-transform both U and V. The original and restructured QR algorithms can achieve high performance with approximately $\mathcal{O}(b(m+n))$ workspace while the D&C-based method requires an additional $3n^2$ workspace elements.

9 Performance

We have implemented the restructured QR algorithm, along with several algorithms for applying multiple sets of Givens rotations, including several optimized variants of the wavefront algorithm presented in Section 4. In this section, we will discuss observed performance of these implementations.

9.1 Platform and implementation 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. All experiments were performed in double-precision floating-point arithmetic on complex Hermitian (for EVD) and complex general (for SVD) input matrices. Generally speaking, we test three types of implementations:

- **netlib LAPACK.** Our core set of comparisons are against Hermitian EVD and SVD implementations found in the netlib distribution of the LAPACK library, version 3.3.1 [1, 30]. When building executables for these netlib codes, we link to OpenBLAS 0.1-alpha-2.4 [34]. The OpenBLAS project provides highly-optimized BLAS based on the GotoBLAS2 library, which was released under an open source license subsequent to version 1.13.
- **MKL.** We also compare against the Math Kernel Library (MKL), version 10.2.2, which is a highly-optimized commercial product offered by Intel that provides a wide range of scientific computing tools, including an implementation of LAPACK as well as a built-in BLAS library.
- **libflame.** The restructured QR algorithm discussed in this paper was implemented in the C programming language and integrated into the `libflame` library for dense matrix computations, which is available to the scientific computing community under an open source license [41, 32]. When linking these implementations, we use the same OpenBLAS library used to link netlib codes.

We compile the netlib and `libflame` codes with the GNU Fortran and C compilers, respectively, included in the GNU Compiler Collection version 4.1.2.

9.2 Applying Givens rotations

Before presenting dense EVD and SVD results, we will first review performance results for various implementations for applying k sets of Givens rotations to an $n \times n$ matrix. For these tests, random (non-identity) rotations were applied to a random matrix V. The results for these implementations are shown in Figure 17.

Let us begin our discussion with the results for "ApplyGsingle (no SSE)," which most closely represent the performance of the LAPACK routine `zlasr`, which applies a single set of Givens rotations to a double-precision complex matrix. Asymptotically, the performance for this implementation is very low—on par with un-optimized level-1 BLAS routines. As with many so-called "unblocked" algorithms, this one performs somewhat better for smaller problem sizes, when the entire matrix V can fit in the L2 cache. Looking at the results for "ApplyGsingle (with SSE)," we see the benefit to using SSE instructions (implemented via C compiler intrinsics). However, performance beyond small problems is still quite low due to the algorithm's relative cache-inefficiency, as discussed in Section 4. By switching to a wavefront algorithm with blocking (while still using SSE vector intrinsics), as reflected by "ApplyGwaveBlk (no fusing)," performance is significantly improved and quite consistent as problem sizes grow larger. We further optimize the blocked wavefront algorithm by employing three kinds of fusing (labeled as $p_k \times p_n$), corresponding to the illustrations in Figures 12, 13, and 14. Remarkably, fusing with $p_k = p_n = 2$ yields performance that is over 60% higher than a similar blocked wavefront implementation without fusing. We include dense `dgemm` performance for reference, where all three matrices are square and equal to the problem size. We also mark the maximum attainable peak performance for applying Givens rotations.[5]

[5]Matrix-matrix multiplication typically incurs equal numbers of floating-point additions and multiplications, which is well-suited for modern architectures which can execute one add and one multiply simultaneously. But if an operation does not have

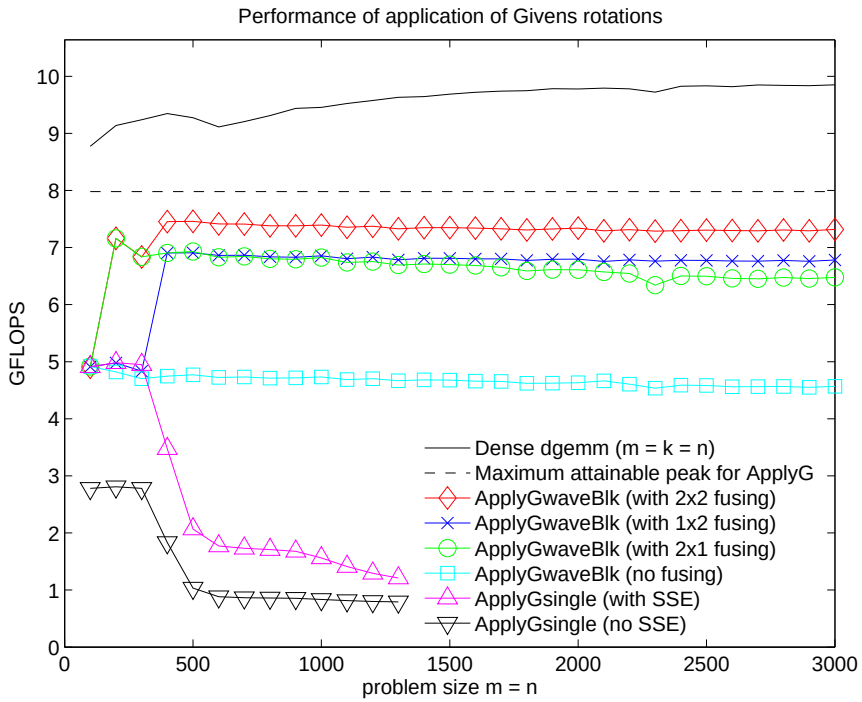

Figure 17: Performance of various implementations for applying $k = 192$ sets of Givens rotations. Those implementations based on blocked algorithms use an algorithmic blocksize $b = 256$. The maximum attainable peak for applying Givens rotations, which is 25% lower than that of `dgemm` because of a 1:2 ratio of floating-point additions to multiplcations, is also shown. Performance results for `dgemm`, where all matrix dimensions are equal, are included for reference.

9.3 Hermitian eigenvalue decomposition

Figure 18 shows the performance of seven implementations for Hermitian EVD, expressed in terms of seconds of run time, on matrices with linearly distributed eigenvalues. (We report results for five other distributions of eigenvalues in Appendix A, along with descriptions of how the distributions were created.) The implementations tested are described as follows:

- **MKL EVD via QR** refers to MKL `zheev`.
- **MKL EVD via D&C** refers to MKL `zheevd`.
- **MKL EVD via MRRR** refers to MKL `zheevr`.
- **netlib EVD via QR** refers to netlib `zheev` linked to OpenBLAS.
- **netlib EVD via D&C** refers to netlib `zheevd`[6] linked to OpenBLAS.
- **netlib EVD via MRRR** refers to netlib `zheevr` linked to OpenBLAS.

a one-to-one ratio of additions to multiplications, some instruction cycles are wasted, resulting in a lower peak performance. The application of Givens rotations is one such operation, generating one addition for every two multiplications. Furthermore, we confirmed that our test system does indeed suffer from this limitation in floating-point execution. Thus, our maximum attainable peak is 7.98 GFLOPS, which is 25% lower than the single core peak of 10.64 GFLOPS.

[6]During our tests, we discovered that the netlib implementation of `zheevd`, when queried for the optimal amount of workspace, returns a value that is approximately $b(n-1)$ less than is needed for the back-transformation to run as a blocked algorithm, which unnecessarily limits performance for most problem sizes. When running our tests, we always provided `zheevd` with additional workspace so that the routine could run at full speed.

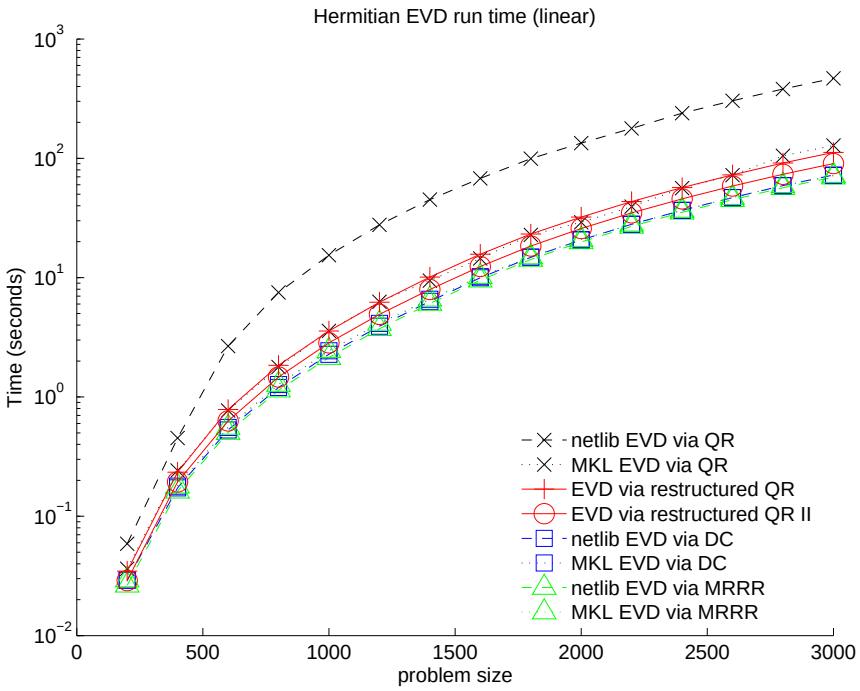

Figure 18: Run times for various implementations of Hermitian EVD. Experiments were run with complex matrices generated with linear distributions of eigenvalues.

- **EVD via restructured QR** refers to an EVD based on our restructured tridiagonal QR algorithm linked to OpenBLAS.
- **EVD via restructured QR II** refers to a variant of EVD via restructured QR that uses the alternative method of forming Q described in Section 6.4.

Note that the three EVD implementations found in MKL are widely known to be highly-optimized by experts at Intel. Also, the two EVDs based on our restructured QR algorithm use an implementation of reduction to tridiagonal form that exhibits a performance signature very similar to that of netlib LAPACK's `zhetrd`.

Default block sizes for netlib LAPACK were used for all component stages of netlib EVD. For each of these suboperations, all of these block sizes happen to be equal to 32. This block size of 32 was also used for the corresponding stages of the EVD via restructured QR. (MKL block sizes are, of course, not known and most likely immutable to the end-user.) For the implementation of ApplyGWaveBlk, the values $k = 32$ and $b = 512$ were used. (These parameter values appeared to give somewhat better results than the values that were used when collecting standalone performance results for ApplyGWaveBlk in Figure 17.)

The results in Figure 18 reveal that using our restructured QR algorithm allows Hermitian EVD performance that is relatively competitive with that of D&C and MRRR. The variant which uses the alternative method of forming Q performs even more closely with that of D&C and MRRR.

While compelling, this data reports only overall performance; it does not show how each stage of EVD contributes to the speedup. As expected, we found that the time spent in the reduction to tridiagonal form was approximately equal for all netlib implementations and our restructured QR algorithm.[7] Virtually all of the differences in overall performance were found in the tridiagonal EVD and the back-transformation (or formation of Q_A).

Figure 19 shows, for problem sizes of 2000×2000 and 3000×3000, the breakdown of time spent in the reduction to tridiagonal form, the tridiagonal EVD, and the formation of Q_A (for the netlib and restructured

[7]Note that we could not confirm this for MKL since we have no way to know that the top-level routines—`zheev`, `zheevd`, and `zheevr`—are actually implemented in terms of the three lower-level operations called by the netlib implementation.

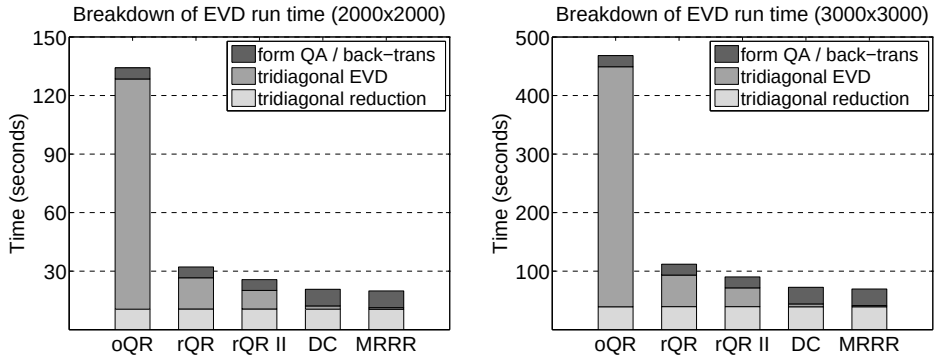

Figure 19: Breakdown of time spent in the three stages of Hermitian EVD: tridiagonal reduction, tridiagonal EVD, and the formation of Q_A (for QR-based algorithms) or the back-transformation (netlib D&C and MRRR) for problem sizes of 2000×2000 (left) and 3000×3000 (right). Here, "oQR" refers to EVD based on the original netlib QR algorithm while "rQR" refers to our restructured algorithm. This shows how the run time advantage of the tridiagonal D&C and MRRR algorithms over restructured QR is partially negated by increased time spent in the former methods' back-transformations.

QR-based algorithms) or the back-transformation (for netlib D&C and MRRR). It is easy to see why methods based on the original QR algorithm are not commonly used: the tridiagonal EVD is many times more expensive than either of the other stages. This can be attributed to the fact that netlib's `zheev` applies one set of Givens rotations at a time, and does so via a routine that typically performs on-par with un-optimized level-1 BLAS. These graphs also illustrate why the restructured QR algorithm is relatively competitive with D&C and MRRR: the run time advantage of using these faster $\mathcal{O}(n^2)$ algorithms is partially offset by an increase in time spent in their back-transformation stages.

9.4 General singular value decomposition

We now turn to performance of the SVD of a dense matrix. Figure 20 shows run times for several implementations. (Once again, we present results for five other singular value distributions in Appendix A.) The implementations tested are:

- **MKL SVD via QR** refers to MKL `zgesvd`.
- **MKL SVD via D&C** refers to MKL `zgesdd`.
- **netlib SVD via QR** refers to netlib `zgesvd` linked to OpenBLAS.
- **netlib SVD via D&C** refers to netlib `zgesdd` linked to OpenBLAS.
- **SVD via restructured QR** refers to an SVD based on our restructured bidiagonal QR algorithm linked to OpenBLAS.
- **SVD via restructured QR II** refers to a variant of SVD via restructured QR that uses an alternative method of forming U and V very similar to that of forming Q in EVD, as described in Section 6.4.

Note that we do not compare against an MRRR implementation of SVD since no such implementation currently exists within LAPACK. Also, we use an implementation of reduction to bidiagonal form that performs very similarly to that of netlib LAPACK's `zgebrd`.

As with the case of EVD, the default block sizes were equal to 32 and also used within the corresponding stages of SVD via restructured QR. Similarly, for ApplyGWaveBlk, the values $k = 32$ and $b = 512$ were used.

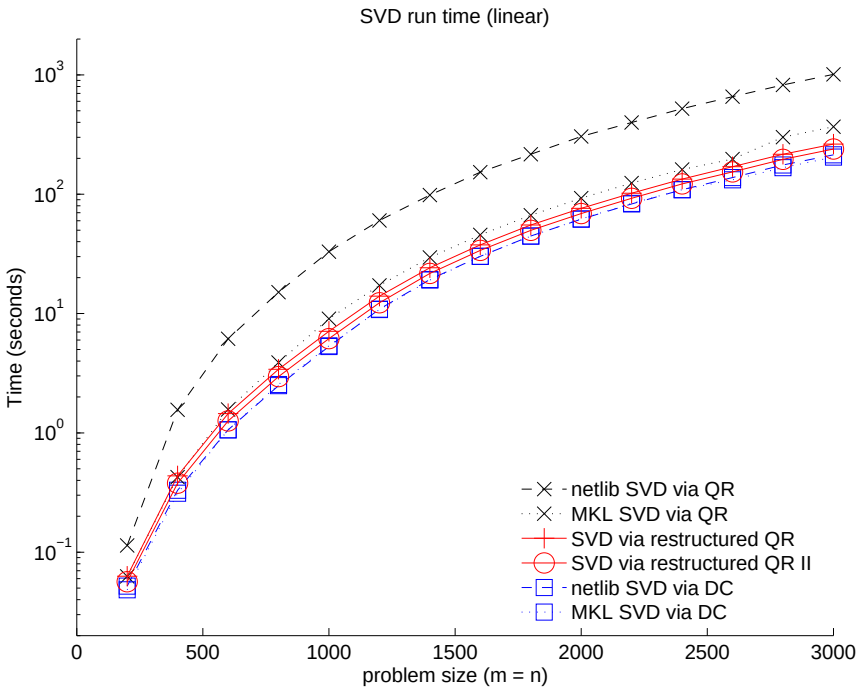

Figure 20: Run times for various implementations of SVD. Experiments were run with complex matrices generated with linear distributions of singular values. Here, the problem sizes listed are $m = n$.

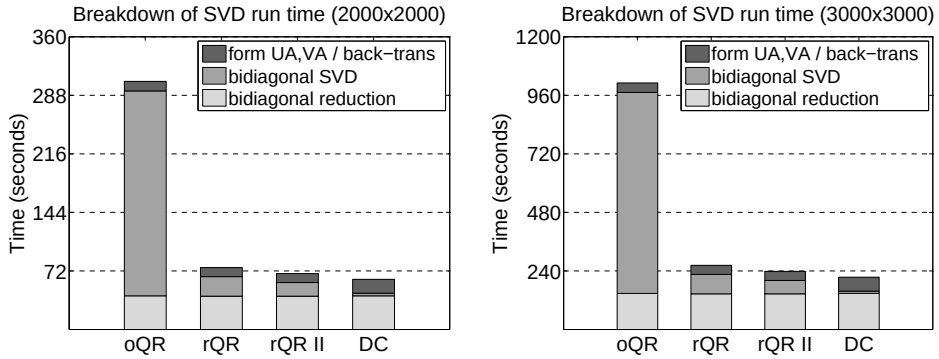

Figure 21: Breakdown of time spent in the three stages of SVD: bidiagonal reduction, bidiagonal SVD, and the formation of U_A and V_A (for QR-based algorithms) or the back-transformation (netlib D&C) for problem sizes of 2000×2000 (left) and 3000×3000 (right). Here, "oQR" refers to EVD based on the original netlib QR algorithm while "rQR" refers to our restructured algorithm. This shows how the run time advantage of the bidiagonal D&C algorithm over restructured QR is partially negated by increased time spent in the former method's back-transformations.

Looking at the performance results, we once again find that the method based on a restructured QR algorithm performs very well against a D&C-based implementation, with the alternate variant performing even better.

Figure 21 shows a time breakdown for SVD where $m = n$. The data tell a story similar to that of Figure 19: we find that the benefit to using the D&C algorithm, when compared to restructured QR, is partially negated by its more costly back-transformation.

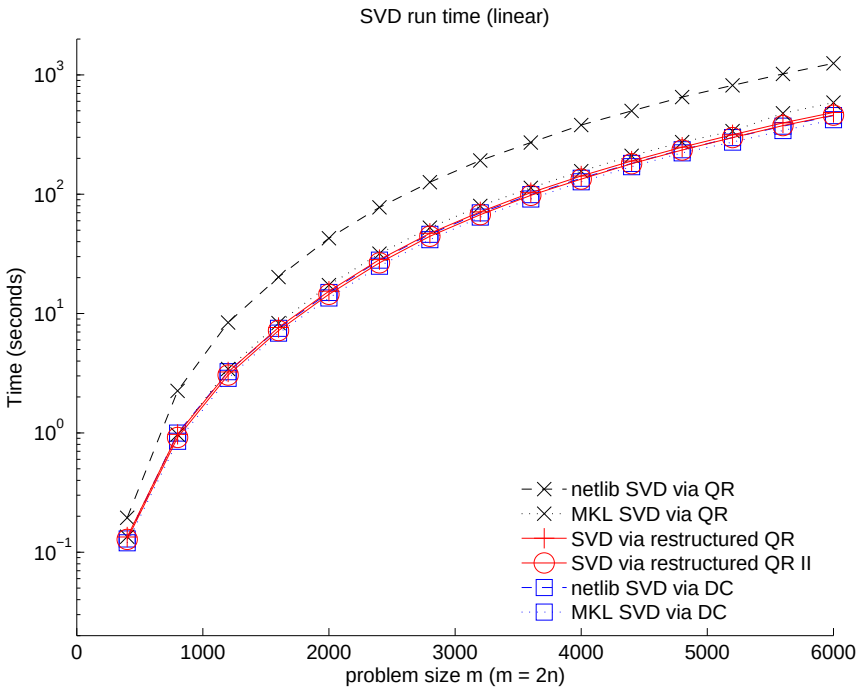

Figure 22: Run times for various implementations of SVD. Experiments were run with complex matrices generated with linear distributions of singular values. Here, the problem sizes listed are m, where $m = 2n$.

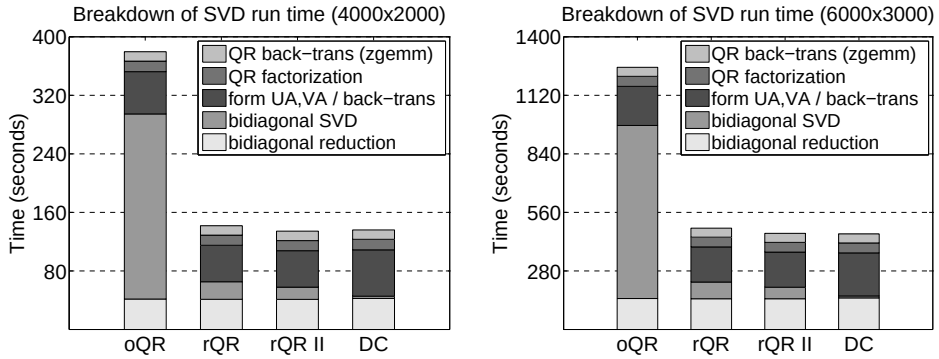

Figure 23: Breakdown of time spent in the three stages of SVD: bidiagonal reduction, bidiagonal SVD, and the formation of U_A and V_A (for QR-based algorithms) or the back-transformation (netlib D&C) for problem sizes of 4000×2000 (left) and 6000×3000 (right). Here, "oQR" refers to EVD based on the original netlib QR algorithm while "rQR" refers to our restructured algorithm. In contrast to Figure 21, these graphs reflect the additional time spent in the QR factorization of A and the corresponding backtransformation necessary to combine Q with $U_R U_B$, which is implemented via `zgemm`.

Figures 22 and 23 show run time and time breakdowns for SVD when the problem size is equal to n, where $m = 2n$. These matrices are sufficiently rectangular to trigger the use of a QR factorization, as described in Section 8. (We know the netlib codes, as well as our implementation using restructured QR, use the QR factorization for these problem shapes, though we are unsure of MKL since it is a proprietary product.) Aside from the additional step of performing the QR factorization, it also requires a corresponding additional back-transformation, performed via `zgemm`, to multiply the unitary matrix Q by the the intermediate product

$U_R U_B$. However, the time spent in both the QR factorization and Q back-transformations is relatively small, and very consistent across methods.

The results show that this $m = 2n$ problem size configuration actually allows the restructured QR-based SVD to match D&C's performance. Figure 23 reveals why: the extra time spent forming Q, along with back-transforming U_A with U_B (and V_A with V_B) virtually erases the time saved by using D&C for the bidiagonal SVD.

Now that there exist high-performance implementations of both the QR and D&C algorithms (and their back-transformations or equivalent stages) the last remaining performance bottleneck is the reduction to bidiagonal form. However, there is potential for speedup via this operation, too. The authors of [43], building on the efforts of [24], report on an implementation of reduction to bidiagonal form that is 60% faster, asymptotically, than the reference implementation provided by netlib LAPACK. For cases where $m = n$, we found the bidiagonal reduction to constitute anywhere from 40 to 60% of the total SVD run time when using the restructured QR algorithm. Thus, combining the results of that paper with this one would further accelerate the overall SVD. (The impact of this faster reduction on the overall SVD can be seen in the curves labeled "libflame SVD (fast BiRed + QR II)" in Figures 30–35 of Appendix A.) While, admittedly, this bidiagonal reduction could be used to speed up any code based on bidiagonal SVD, the `libflame` library is the only dense linear algebra library we know of to currently offer such an implementation.

9.5 Parallelization

Parallelization issues, while interesting and deserving of attention, lie beyond the intended scope of this paper. We plan to study this topic in future work.

10 Conclusions

Ever since the inception of the level-3 BLAS, some have questioned whether a routine for applying multiple sets of Givens rotations should have been part of the BLAS. We believe that the present paper provides compelling evidence that the answer is in the affirmative.

One could render quick judgement on this work by pointing out that the restructured QR algorithm facilitates performance that is merely competitive with, not faster than, that of a number algorithms that were developed in the 1980s and 1990s. But one could argue that there is undervalued virtue in simplicity and reliability (and the QR algorithm exhibits both), especially if it comes at the expense of only minor performance degradation. And to some, the option of $\mathcal{O}(n)$ workspace is important. The D&C and MRRR algorithms took many years (decades) to tweak and perfect (mostly for accuracy reasons) and require an expert to maintain and extend. By contrast, the QR algorithm as we present it is essentially identical to the one developed for EISPACK [39] in the 1960s, which can be understood and mastered by a typical first-year graduate student in numerical linear algebra. Perhaps most importantly, the QR algorithm's numerical robustness is undisputed, and now that its performance (in the case of dense matrices) is competitive, we would argue that it should be preferred over other alternatives because it can reliably solve a wide range of problems.

The results presented in this paper present several avenues for future research. We believe that the application of Givens rotations within the restructured QR algorithm may be parallelized for multicore systems with relatively minor changes. Acceleration with GPUs should also be possible. In some sense, limiting our discussion in this paper to only a single core favors other methods: on multiple cores, the reduction to tridiagonal form will, relatively speaking, become even more of an expense since it is largely limited by bandwidth to main memory. Recent re-emergence of successive band reduction tries to overcome this [33, 2, 23]. Now that the QR algorithm is competitive with more recent algorithms, it may be worth attempting to improve the shifting mechanism. For example, in [26] it is shown that a hybrid method that uses both Rayleigh Quotient shifts and Wilkinson shifts is cubically (and globally) convergent. This may reduce the number of Francis steps for many matrix types, which would decrease the overall runtime of the QR algorithm. Alternatively, in [15] it is suggested that, if eigenvalues are computed first, a "perfect shift" can be incorporated in the QR algorithm that causes deflation after only one iteration, though the same

document uncovers certain challenges associated with this approach. "Aggressive early deflation" has been developed to accelerate the QR algorithm for upper Hessenberg matrices [6]. It may be possible to develop a similar technique for symmetric tridiagonal and/or bidiagonal matrices. Previous efforts have succeeded in achieving level-3 BLAS performance in the aforementioned Hessenberg QR algorithm by applying multiple shifts and chasing multiple bulges simultaneously [5]. It may be possible to further improve those methods by restructuring the computation so that the wavefront algorithm for applying Givens rotations may be employed. Periodically, the Jacobi iteration (for the EVD and SVD) receives renewed attention [25, 17]. Now that the excellent performance of a routine that applies sets of Givens rotations has been demonstrated, an interesting question is how to incorporate this into a blocked Jacobi iteration. Thus, our work lays the foundation for further research.

Acknowledgments

We kindly thank G. W. (Pete) Stewart, Paolo Bientinesi, and Jack Poulson for offering valuable advice and expertise.

This research was partially sponsored by the UTAustin-Portugal Colab program, a grant from Microsoft, and grants from the National Science Foundation (Awards OCI-0850750 and CCF-0917167).

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, Francisco D. Igual, Daniel Kressner, Matthias Petschow, and Enrique S. Quintana-Ortí. Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. *Concurr. Comput. : Pract. Exper.*, 23(7):694–707, May 2011.

[3] Christian Bischof and Charles van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):2–13, January 1987.

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

[5] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. part I: Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23:929–947, April 2001.

[6] Karen Braman, Ralph Byers, and Roy Mathias. The multishift QR algorithm. part II: Aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23:948–973, April 2001.

[7] Alan K. Cline and Inderjit S. Dhillon. *Handbook of Linear Algebra*, chapter Computation of the Singular Value Decomposition, pages 45–1–45–13. CRC Press, 2006.

[8] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36:177–195, 1980.

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

[10] James W. Demmel, Osni A. Marques, and Beresford N. Parlett. Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers. LAPACK Working Note 183, April 2007.

[11] Inderjit S. Dhillon. Current inverse iteration software can fail. *BIT Numer. Math.*, 38:685–704, 1998.

[12] Inderjit S. Dhillon and Beresford N. Parlett. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, March 2003.

[13] Inderjit S. Dhillon and Beresford N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Lin. Alg. Appl.*, 387:1–28, 2004.

[14] Inderjit S. Dhillon, Beresford N. Parlett, and Christof Vömel. The design and implementation of the mrrr algorithm. *ACM Trans. Math. Soft.*, 32:533–560, December 2006.

[15] Inderjit Singh Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, EECS Department, University of California, Berkeley, Oct 1997.

[16] 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.

[17] Zlatko Drmač. A global convergence proof for cyclic Jacobi methods with block rotations. *SIAM J. Matrix Anal. Appl.*, 31(3):1329–1350, November 2009.

[18] K. V. Fernando and B. N. Parlett. Accurate singular values and differential qd algorithms. *Numer. Math.*, 67:191–229, 1994.

[19] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[20] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3), 2008.

[21] Ming Gu and Stanley C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Anal. Appl.*, 16(1):79–92, January 1995.

[22] Ming Gu and Stanley C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16(1):172–191, January 1995.

[23] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. LAPACK Working Note 254, August 2011.

[24] 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.

[25] C.G.J. Jacobi. Über ein leichtes verfahren, die in der theorie der säkularstörungen vorkommenden gleichungen numerisch aufzulösen. *Crelle's Journal*, 30:51–94, 1846.

[26] Erxiong Jiang and Zhenyue Zhang. A new shift of the QL algorithm for irreducible symmetric tridiagonal matrices. *Linear Algebra and its Applications*, 65(0):261–272, 1985.

[27] 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.

[28] Bo Kågström, Daniel Kressner, E. S. Quintana-Ortí, and G. Quintana-Ortí. Blocked algorithms for the reduction to Hessenberg-triangular form revisited. *BIT Numer. Math.*, 48(3):563–584, 2008.

[29] Bruno Lang. Using level 3 BLAS in rotation-based algorithms. *SIAM J. Sci. Comput.*, 19(2):626–634, March 1998.

[30] LAPACK. `http://www.netlib.org/lapack/`, 2011.

[31] 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.

[32] libflame. `http://www.cs.utexas.edu/users/flame/libflame/`, 2011.

[33] Piotr Luszczek, Hatem Ltaief, and Jack Dongarra. Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. LAPACK Working Note 244, April 2011.

[34] OpenBLAS. `https://github.com/xianyi/OpenBLAS/`, 2011.

[35] B. N. Parlett and O. A. Marques. An implementation of the dqds algorithm (positive case). *Lin. Alg. Appl.*, 309:217–259, 1999.

[36] Gregorio Quintana-Ortí and Antonio M. Vidal. Parallel algorithms for QR factorization on shared memory multiprocessors. In *European Workshop on Parallel Computing'92, Parallel Computing: From Theory to Sound Practice*, pages 72–75. IOS Press, 1992.

[37] Jeffery D Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Technical Report CSD-94-799, Computer Science Division, University of California at Berkeley, February 1994.

[38] 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, January 1989.

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

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

[41] Field G. Van Zee. *libflame: The Complete Reference*. `http://www.lulu.com/`, 2011.

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

[43] Field G. Van Zee, Robert van de Geijn, Gregorio Quintana-Ortí, and G. Joseph Elizondo. Families of algorithms for reducing a matrix to condensed form. *ACM Trans. Math. Soft.*, 2011. submitted. Available from `http://z.cs.utexas.edu/wiki/flame.wiki/Publications/`.

[44] David S. Watkins. Understanding the QR algorithm. *SIAM Review*, 24(4):427–440, 1982.

[45] J. H. Wilkinson. Global convergence of tridiagonal QR algorithm with origin shifts. *Linear Algebra and its Applications*, 1:409–420, 1968.

[46] Paul R. Willems and Bruno Lang. A framework for the MR^3 algorithm: Theory and implementation. Technical report, Bergische Universität Wuppertal, April 2011.

[47] Paul R. Willems, Bruno Lang, and Christof Vömel. Computing the bidiagonal SVD using multiple relatively robust representations. *SIAM J. Matrix Anal. Appl.*, 28(4):907–926, 2006.

A Additional Performance Results

In this section, we present complete results for six eigen-/singular value distributions for Hermitian EVD as well as SVD where $m = n$. The distribution types for eigen-/singular values λ_i, for $i \in 0, 1, 2, \ldots, n-1$, are a follows:

- **Geometric** distributions are computed as $\lambda_i = \alpha(1-\alpha)^{i+1}$. For our experiments, we used $\alpha = \dfrac{1}{n}$.
- **Inverse** distributions are computed as $\lambda_i = \dfrac{\alpha}{i+1}$. For our experiments, we used $\alpha = 1$.
- **Linear** distributions are computed as $\lambda_i = \beta + \alpha(i+1)$. For our experiments, we used $\alpha = 1$ and $\beta = 0$.
- **Logarithmic** distributions are computed as $\lambda_i = \dfrac{\alpha^{i+1}}{\alpha^n}$. For our experiments, we used $\alpha = 1.2$.
- **Random** distributions are computed randomly over an interval $[\sigma, \sigma + \omega]$. For our experiments, we used $\sigma = 0$ and $\omega = n$.
- **Cluster** distributions are generated containing n_c clusters, roughly equally-sized, where half of the overall spectrum lies in one of the n_c clusters while the remaining half resides in the (relatively) sparsely populated region between clusters. More specifically, this spectrum is generated by alternating between linear and random distributions. The first segment of the distribution is linear and roughly $\dfrac{n}{2n_c}$ values are generated (with β initially zero, and $\alpha = 1$ which remains constant), up to some λ_j. This value λ_j then becomes σ in a random distribution, where $0 < \omega \ll 1$. The last value in the random interval serves as the β for the next linear segment, and so forth until $2n_c$ separate regions have been created. For our experiments, we used $n_c = 10$ and $\omega = 10^{-9}$.

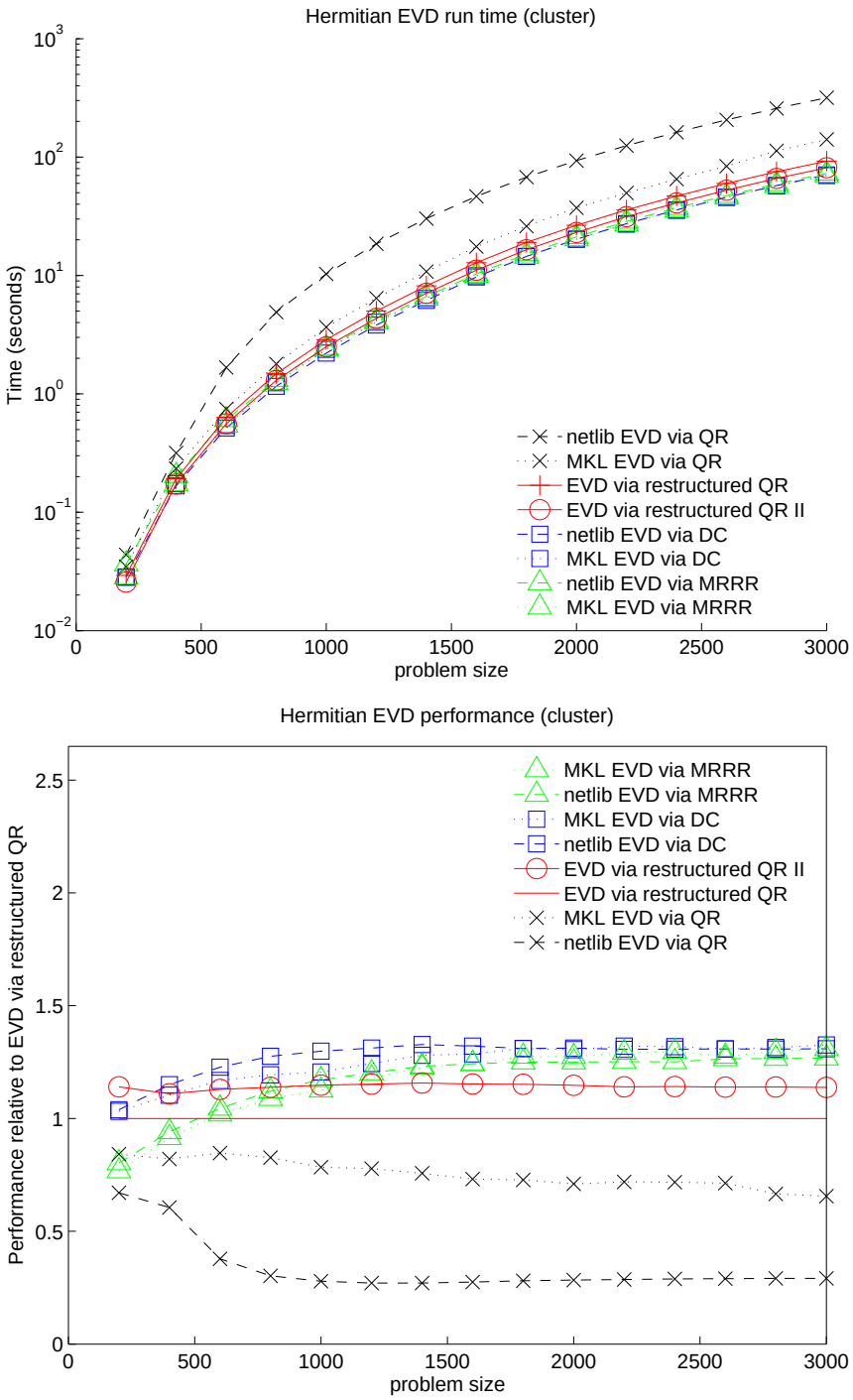

Figure 24: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with clustered distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

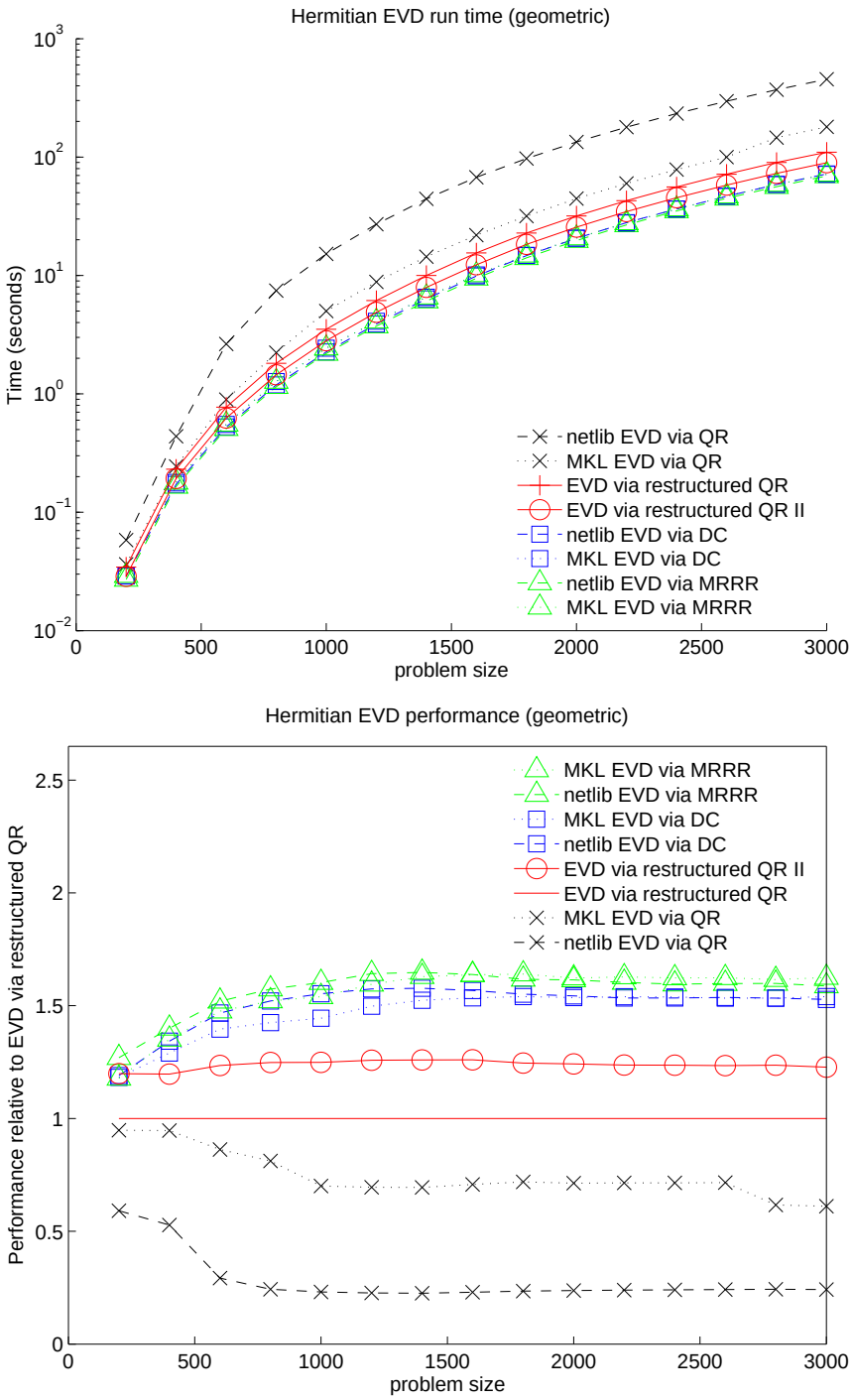

Figure 25: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with geometric distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

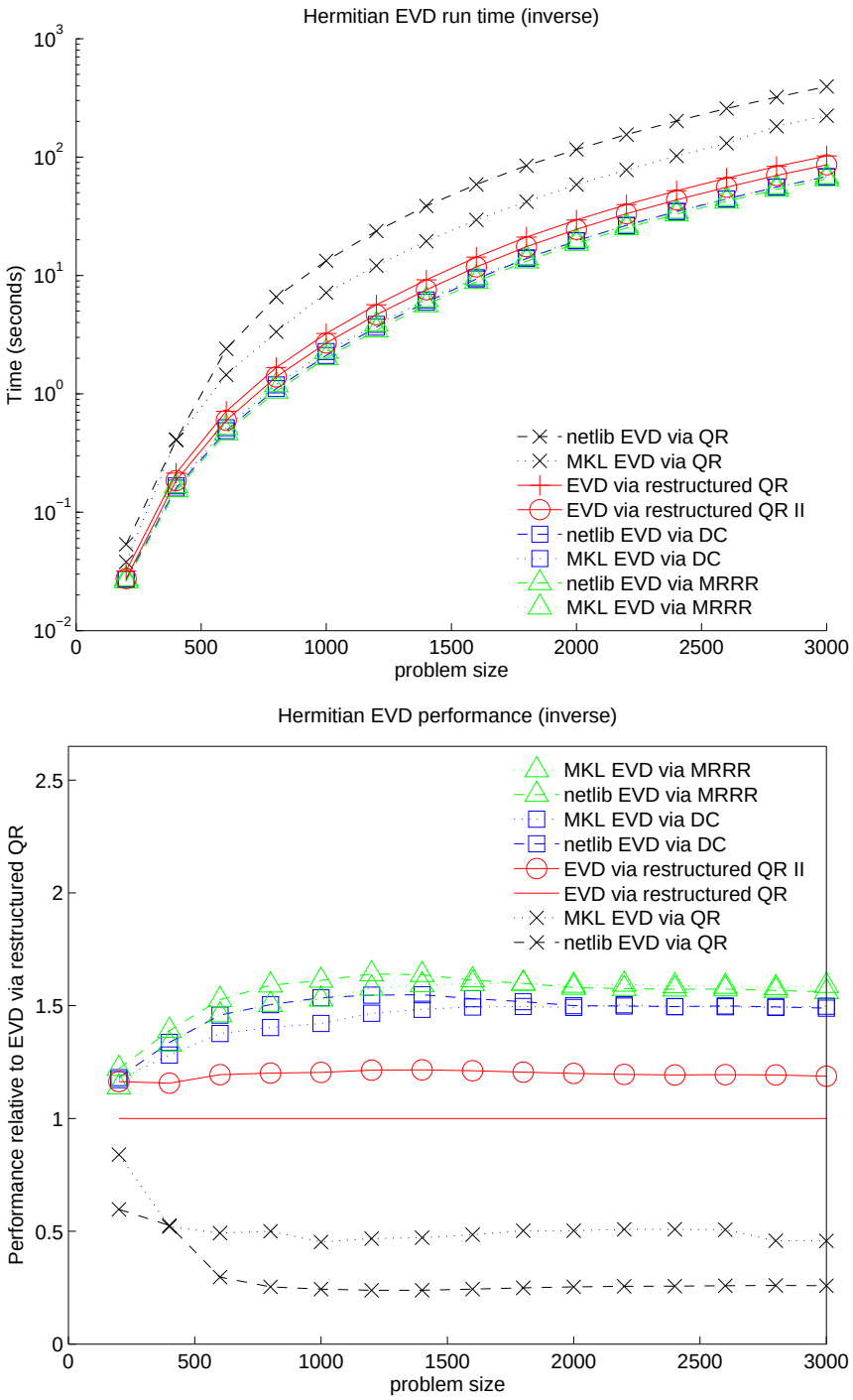

Figure 26: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with inverse distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

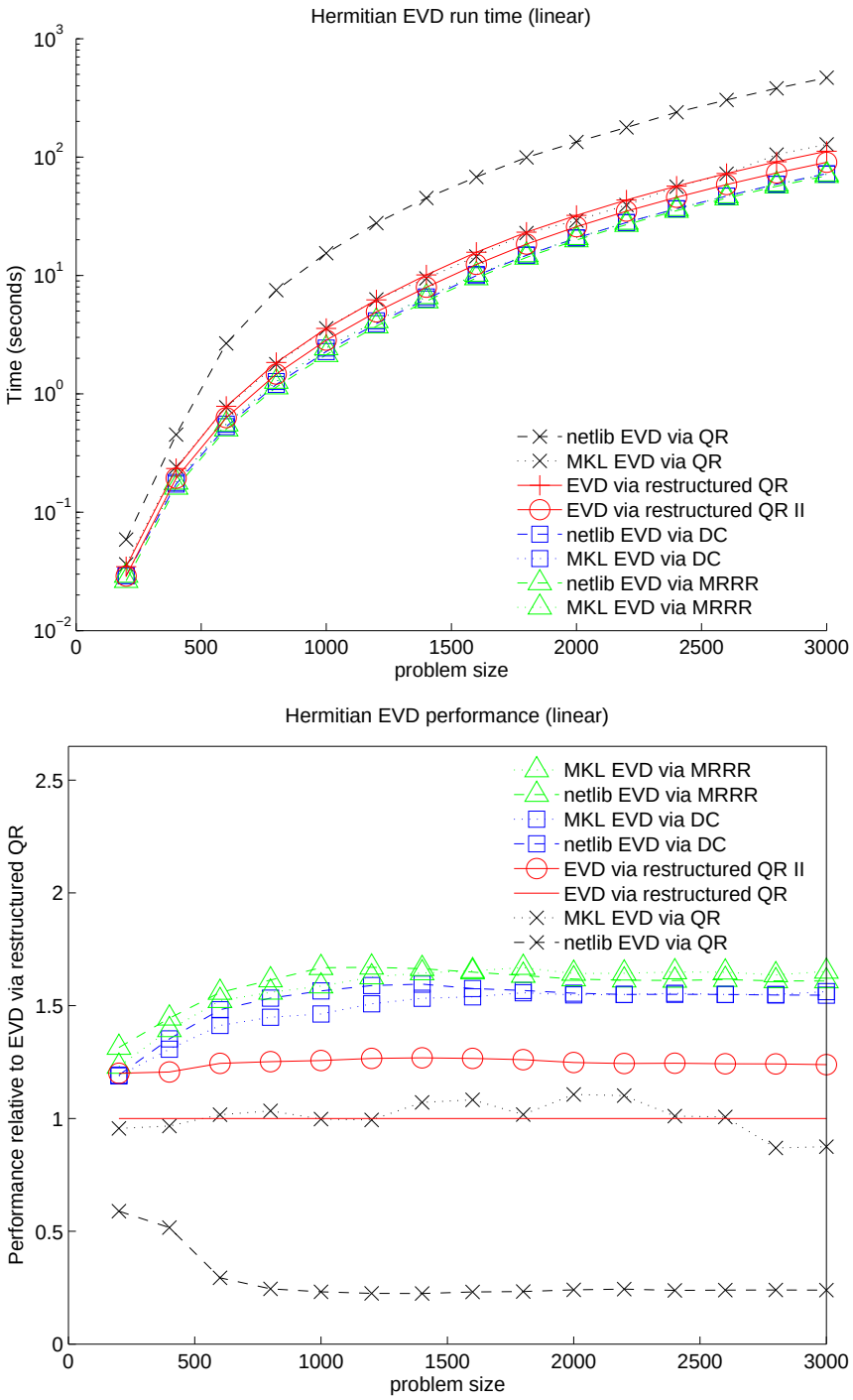

Figure 27: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with linear distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

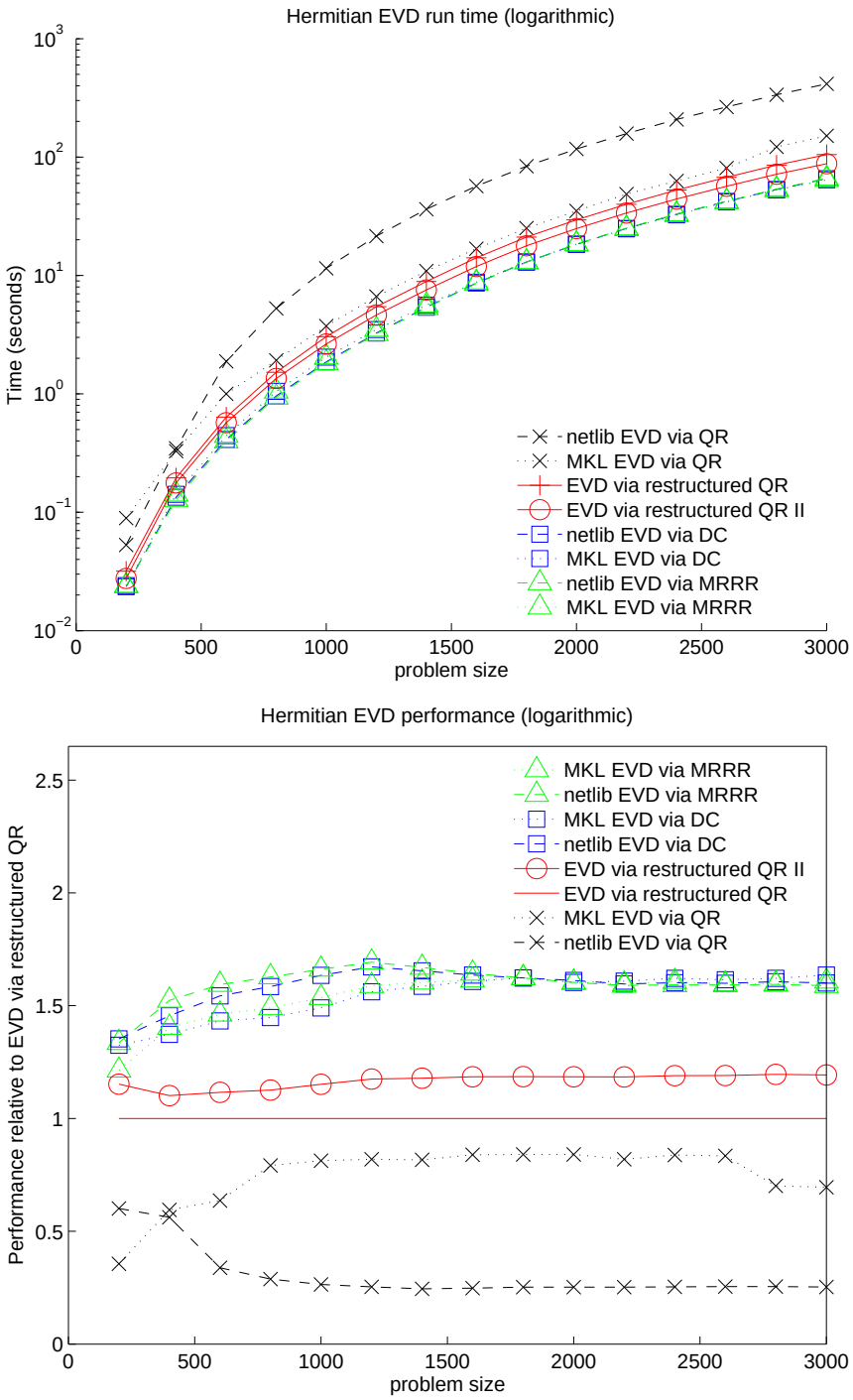

Figure 28: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with logarithmic distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

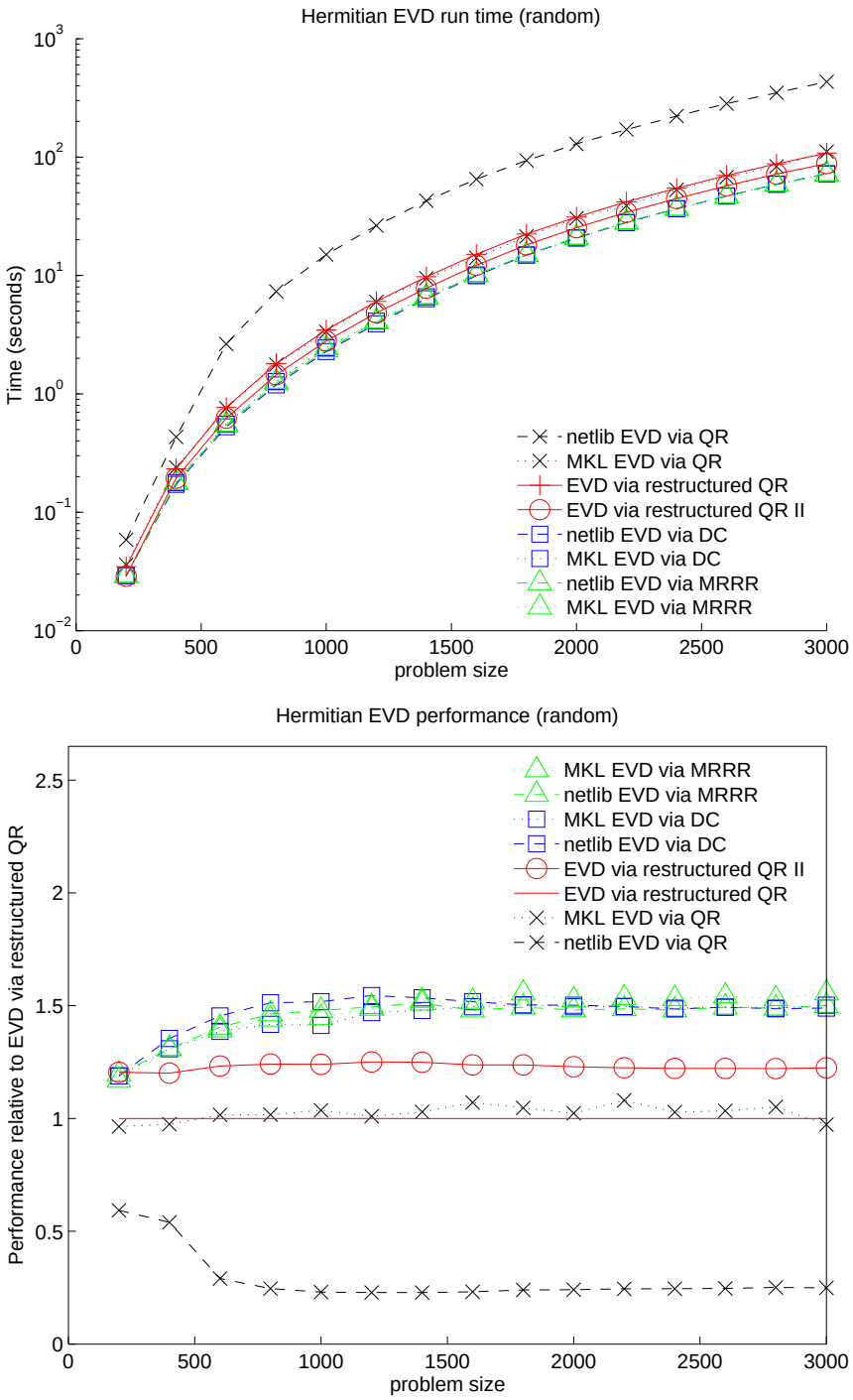

Figure 29: Top: Run times for various implementations of Hermitian EVD. Bottom: Performance of implementations relative to EVD via restructured QR. Experiments were run with complex matrices generated with random distributions of eigenvalues. In the case of EVD via restructured QR, matrices were reduced to tridiagonal form using a conventional implementation similar to that of `zhetrd`.

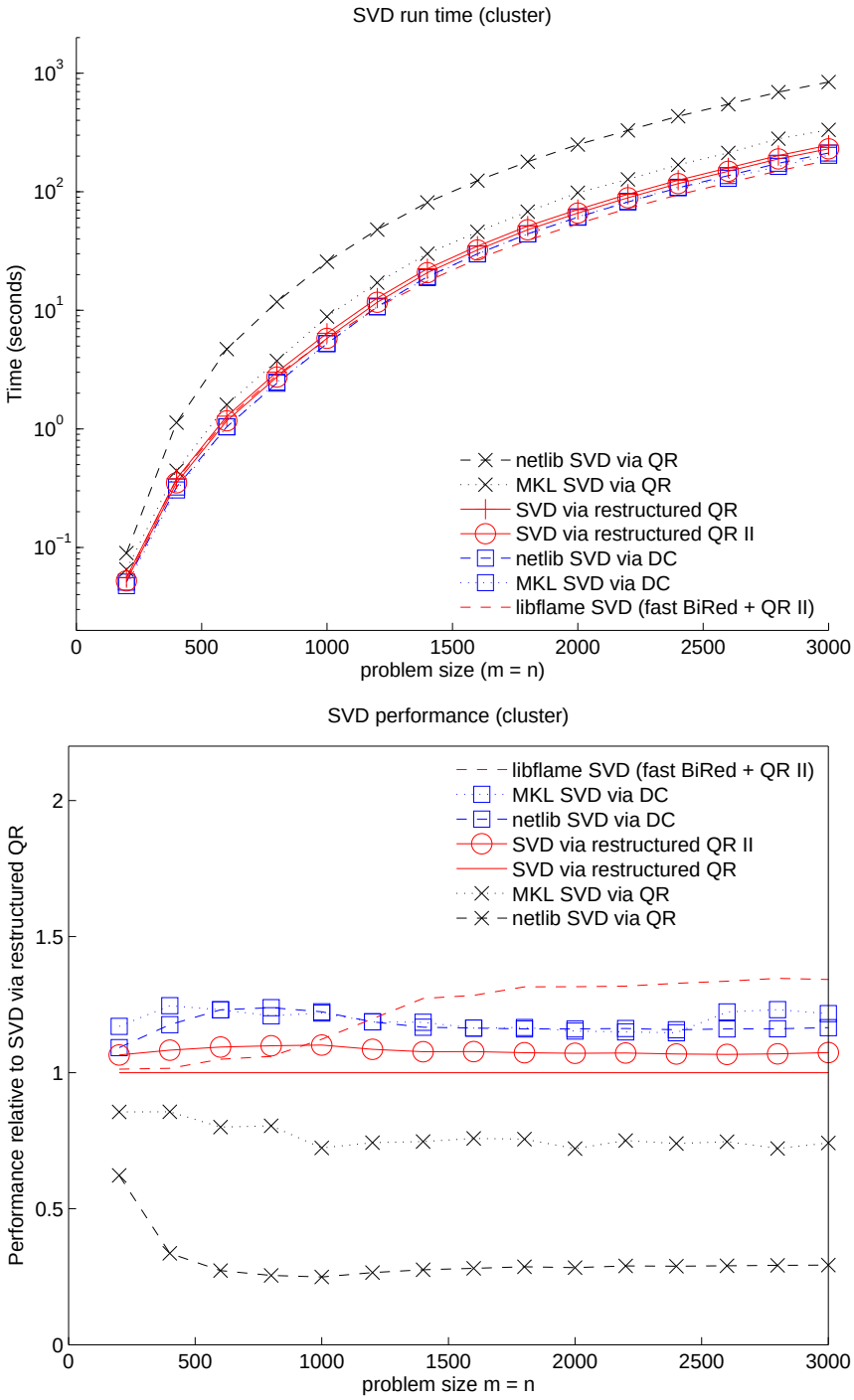

Figure 30: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with clustered distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.

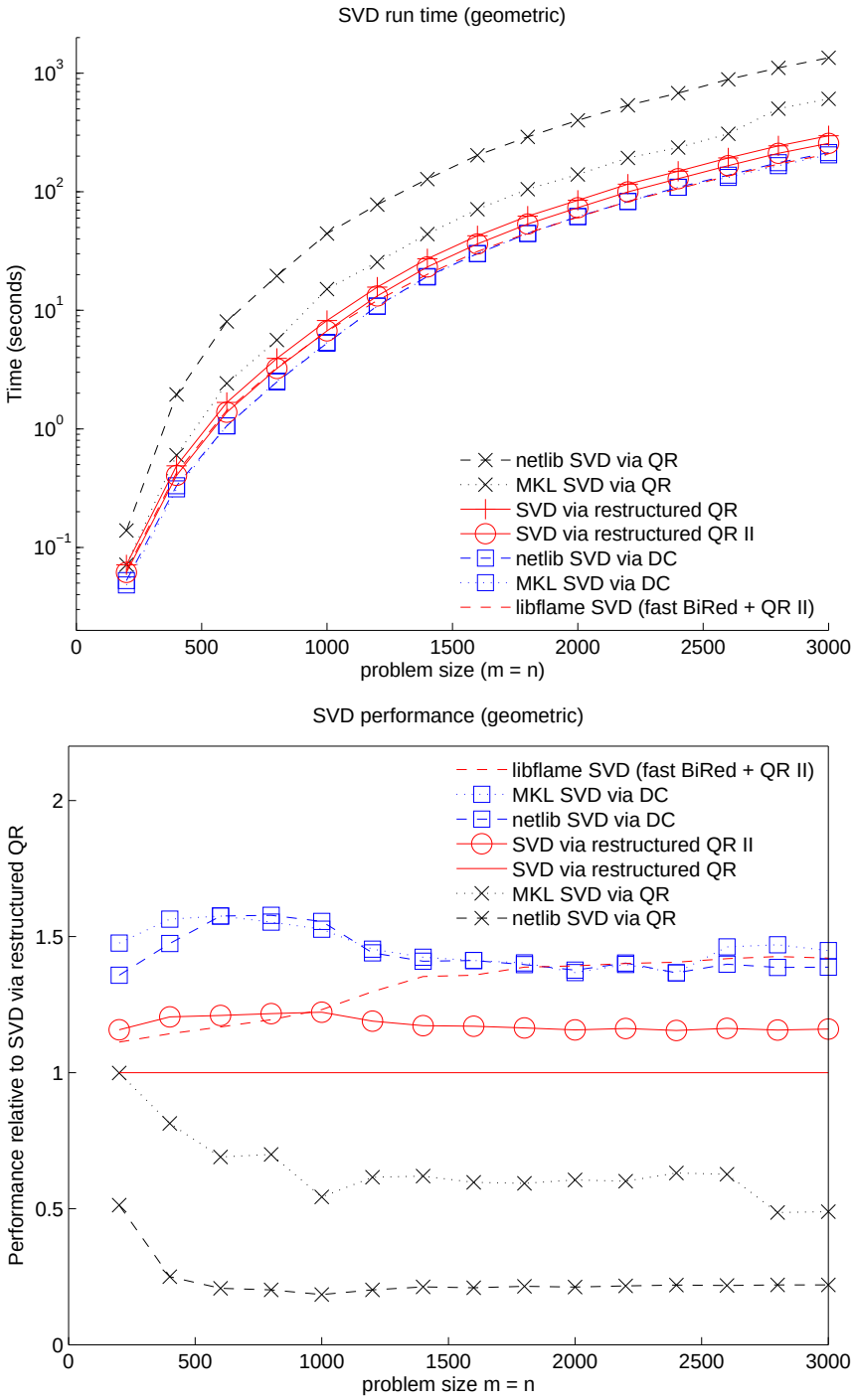

Figure 31: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with geometric distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.

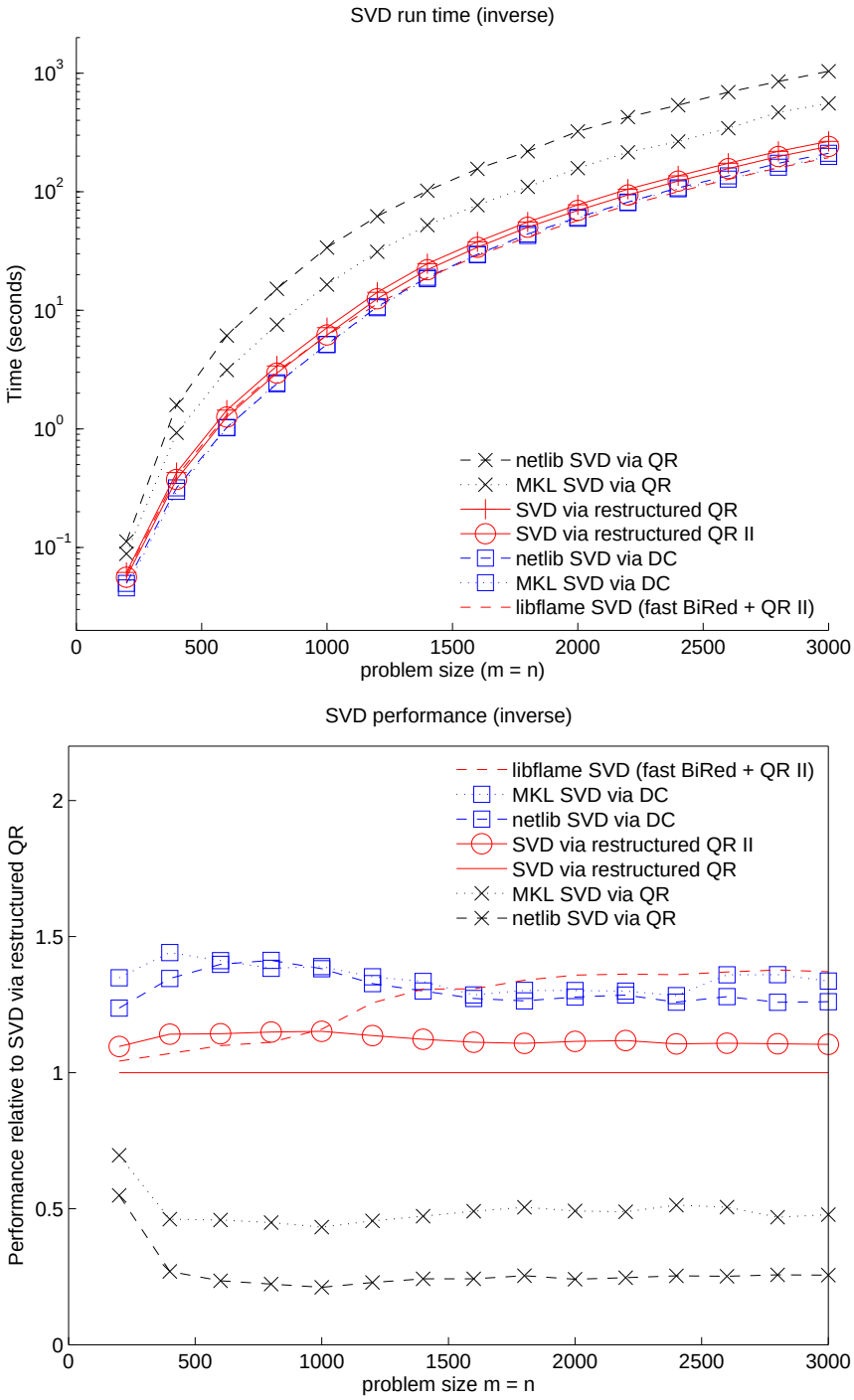

Figure 32: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with inverse distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.

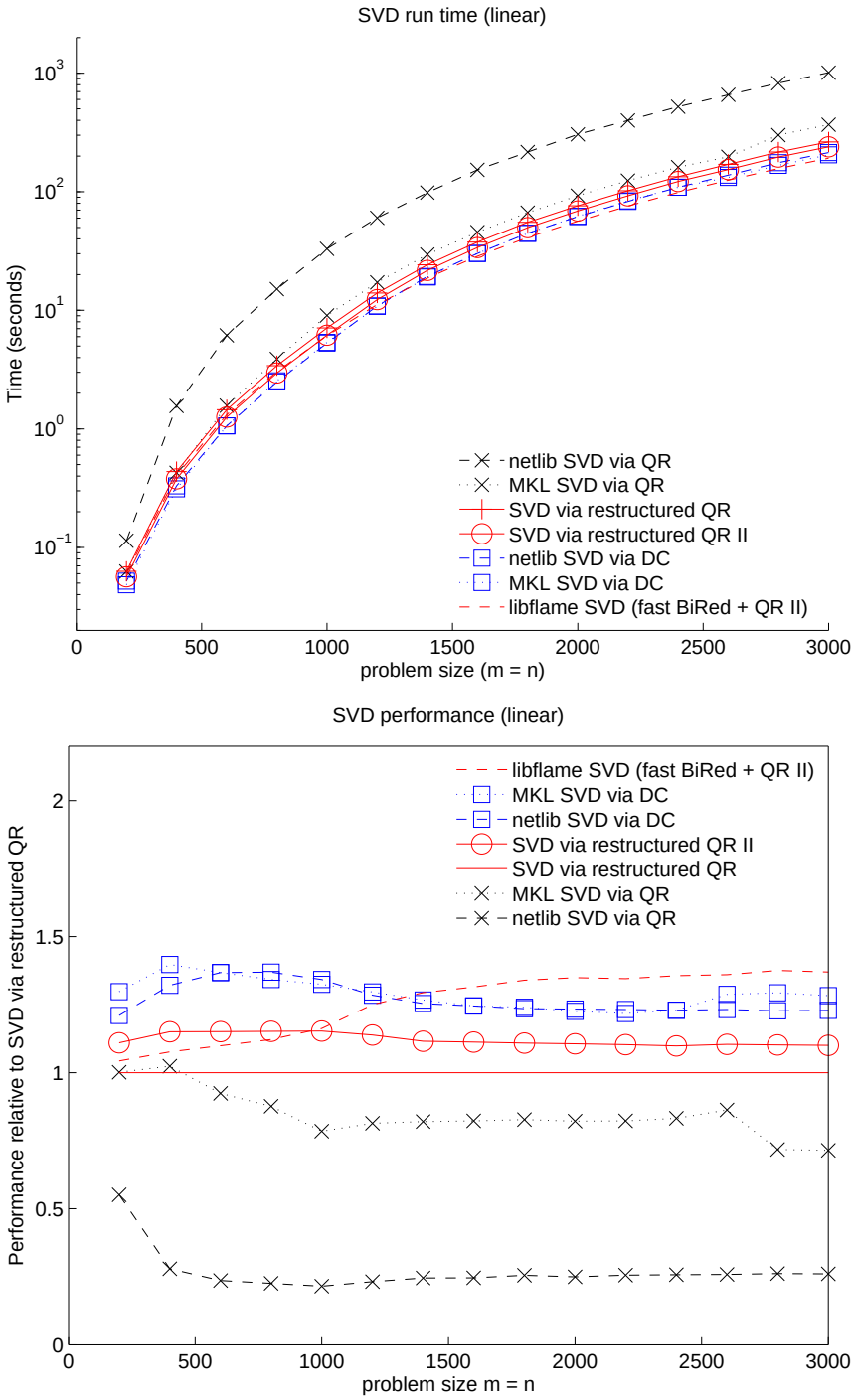

Figure 33: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with linear distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.

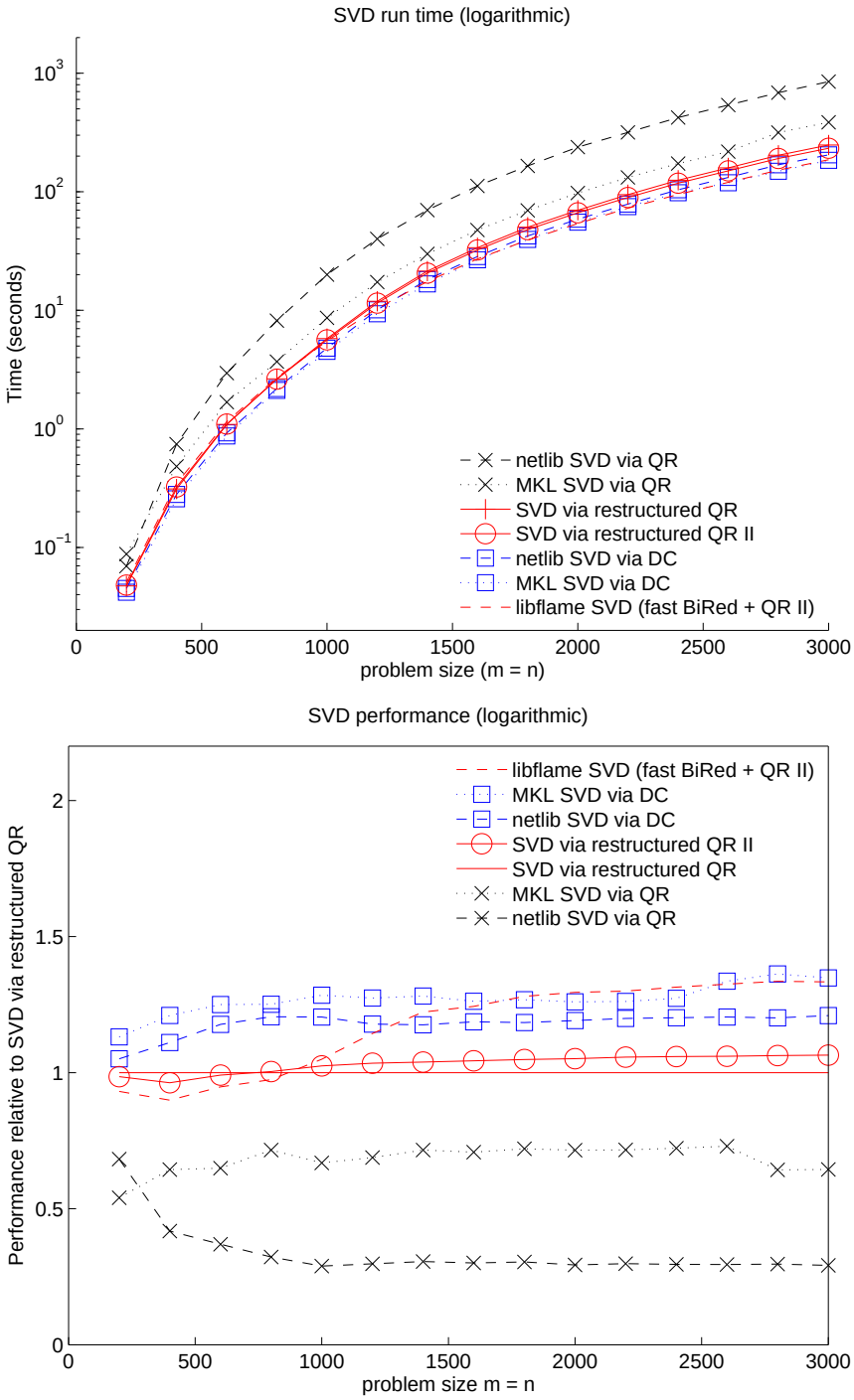

Figure 34: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with logarithmic distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.

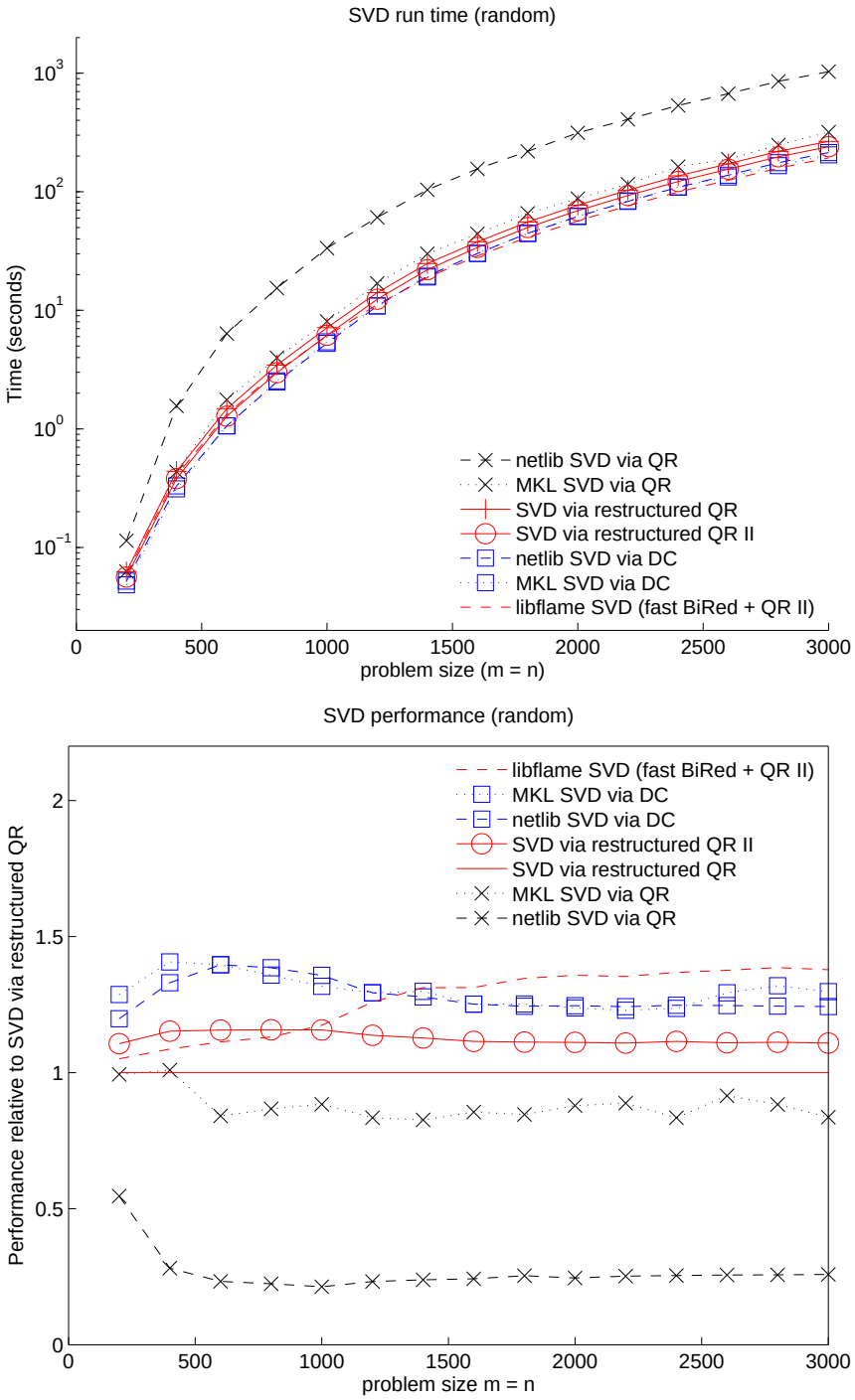

Figure 35: Top: Run times for various implementations of SVD. Bottom: Performance of implementations relative to SVD via restructured QR. Experiments were run with complex matrices generated with random distributions of singular values. Here, the problem sizes listed are $m = n$. In the case of "SVD via restructured QR" and "SVD via restructured QR II", matrices were reduced to bidiagonal form using a conventional implementation similar to that of `zgebrd`. Results for "libflame SVD (fast BiRed + QR II)" combine the higher-performing bidiagonal reduction presented in [43] with the restructured QR II algorithm.