Accelerating computation of eigenvectors in the dense nonsymmetric eigenvalue problem

Mark Gates[1], Azzam Haidar[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, TN, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[3] University of Manchester, Manchester, UK
{mgates3, haidar, dongarra}@eecs.utk.edu

Abstract. In the dense nonsymmetric eigenvalue problem, work has focused on the Hessenberg reduction and QR iteration, using efficient algorithms and fast, Level 3 BLAS. Comparatively, computation of eigenvectors performs poorly, limited to slow, Level 2 BLAS performance with little speedup on multi-core systems. It has thus become a dominant cost in the solution of the eigenvalue problem. To address this, we present improvements for the eigenvector computation to use Level 3 BLAS and parallelize the triangular solves, achieving good parallel scaling and accelerating the overall eigenvalue problem more than three-fold.

1 Introduction

Eigenvalue problems are fundamental for many engineering and physics applications. For example, image processing, facial recognition, vibrational analysis of mechanical structures, seismic reflection tomography, and computing electron energy levels can all be expressed as eigenvalue problems. The eigenvalue problem is to find an eigenvalue λ and eigenvector x that satisfy $Ax = \lambda x$, where A is an $n \times n$ matrix. When the entire eigenvalue decomposition is computed we have $A = X \Lambda X^{-1}$, where Λ is a diagonal matrix of eigenvalues and X is a matrix of eigenvectors. In this paper we consider the case when A is dense and nonsymmetric. We concentrate on computing the eigenvectors, and present optimizations that accelerate the overall eigenvalue problem more than three-fold.

The solution of the eigenvalue problem proceeds in three phases [5], as implemented in LAPACK's geev routine. First, the matrix is reduced to upper Hessenberg form by applying orthogonal Q matrices on the left and right, to form $H = Q_1^T A Q_1$. This phase takes $\frac{10}{3}n^3$ floating point operations (flops), and is formulated [1, 2] so that 80% of these occur in efficient Level 3 BLAS matrix-matrix products (gemm), while the remaining 20% occur in memory-bound Level 2 BLAS matrix-vector products (gemv). Performance is limited by the memory bandwidth for gemv operations, which can take 70% of the time. Using multi-threaded BLAS, it achieves a modest 7 times speedup on 16 cores. A recent two-stage implementation reduces the amount of gemv operations [7]. A GPU accelerated version [9] is an additional 4 times faster than the 16-core performance. For large matrices, the Hessenberg reduction accounts for approximately 20% of the overall time using 16 cores.

The second phase, QR iteration, is an iterative process that reduces the Hessenberg matrix to upper triangular Schur form, $T = Q_2^T A Q_2$. Being based on similarity transformations, the eigenvalues of A are the same as the eigenvalues of T, which are simply the diagonal elements of T. The QR iteration method takes $O(n^3)$ flops, but being an iterative method, the exact count depends heavily on the convergence rate and techniques such as aggressive early deflation [3, 4]. It includes a mixture of Level 1 BLAS for applying Givens rotations and Level 3 BLAS for updating H and accumulating Q_2. Parallel versions also exist [6]. While for small matrices, QR iteration can take over 50% of the time, for large matrices this reduces to about 15% of the time on 16 cores.

Finally, the third phase computes eigenvectors Z of the Schur form T and back-transforms them to eigenvectors X of the original matrix A. The eigenvectors of A are related to the eigenvectors of T by multiplying with the orthogonal matrices used in the Hessenberg reduction and QR iteration as $X = QZ$, where $Q = Q_1 Q_2$. In the LAPACK implementation (trevc), computation of each eigenvector involves a triangular solve (latrs) and matrix-vector product (gemv). This phase takes $\frac{4}{3}n^3$ flops. However, due to a lack of parallelization and involving only Level 2 BLAS operations, it has the lowest Gflop/s performance of the three phases, asymptotically taking over 60% of the time on 16 cores.

Thus, despite having the least flops of the three phases, the computation of eigenvectors has become the dominant cost and limited the overall parallel speedup of the solution of the eigenvalue problem. This paper is therefore concerned with accelerating the eigenvector computation through three improvements. First, for the back-transformation, we block multiple Level 2 gemv products into an efficient Level 3 gemm product, discussed in Section 3. Second, we parallelize the triangular solves using a task-based scheduler, as described in Section 4. Finally, using a GPU, the back-transformation is further accelerated and done in parallel with the triangular solves, in Section 5. Combined, these improvements significantly increase the performance and scalability of the overall eigenvalue problem, demonstrated by the results in Section 6.

2 Eigenvector Computation

When eigenvectors are desired, the third phase computes eigenvectors of the triangular Schur form T, then back-transforms them to eigenvectors of the original matrix A. In LAPACK, this phase is implemented in the trevc (triangular eigenvector computation) routine. We will assume only right eigenvectors are desired; the computation of left eigenvectors is similar and amenable to the same techniques described here. After the Hessenberg and QR iteration phases, the diagonal entries of T are the eigenvalues λ_k of A. To determine the corresponding eigenvectors, we solve $Tz_k = \lambda_k z_k$ by considering the decomposition [5]

$$\begin{bmatrix} T_{11} & u & T_{13} \\ 0 & \lambda_k & v^T \\ 0 & 0 & T_{33} \end{bmatrix} \begin{bmatrix} \hat{z} \\ 1 \\ 0 \end{bmatrix} = \lambda_k \begin{bmatrix} \hat{z} \\ 1 \\ 0 \end{bmatrix}, \tag{1}$$

which yields $(T_{11} - \lambda_k I)\hat{z} = -u$. Thus computing each eigenvector z_k of T involves a $k-1 \times k-1$ triangular solve, for $k = 2, \ldots, n$. Each solve has a slightly different T matrix, with the diagonal modified by subtracting λ_k. The resulting eigenvector z_k of T must then be back-transformed by multiplying with the Q formed in the Hessenberg and QR iteration phases to get the eigenvector $x_k = Qz_k$ of the original matrix A.

Note that if two eigenvalues, λ_k and λ_j $(k > j)$, are identical, then $T_{11} - \lambda_k I$ is singular. More generally, $T_{11} - \lambda_k I$ can be badly conditioned. Therefore, instead of using the standard BLAS triangular solver (trsv), a specialized triangular solver (latrs) is used, which scales columns to protect against overflow, and can generate a consistent solution for a singular matrix.

This method works in complex arithmetic, however the case in real arithmetic is more complicated. For a real matrix A, the eigenvalues can still be complex, coming in conjugate pairs, λ_k and $\bar{\lambda}_k$. The eigenvectors are likewise conjugate pairs, z_k and $\bar{z}_k$. In real arithmetic, the closest that QR iteration can come to triangular Schur form is quasi-triangular real Schur form, which has a 2×2 diagonal block for each conjugate pair of eigenvalues. A specialized quasi-triangular solver is required, which factors each 2×2 diagonal block, as well as protecting against overflow and dealing with singular matrices. In LAPACK this solver is implemented as part of the dtrevc routine.

3 Blocking back-transformation

Our first step to improve the eigenvector computation is to block the n gemv operations for the back-transformation into n/n_b gemm operations, where n_b is the block size. This requires two $n \times n_b$ workspaces: one for the vectors z_k, the second for the back-transformed vectors x_k before copying to the output V.

Pseudocode for the blocked back-transformation is shown in Algorithm 1, along with the parallel solver described in Section 4. For each block, we loop over n_b columns, performing a triangular solve for each column and storing the resulting eigenvectors z_k in workspace Z. After filling up n_b columns of Z, a single gemm back-transforms all n_b vectors, storing the result in workspace $\tilde{Z}$. The vectors are then normalized and copied to V. On input, the matrix $V = Q$. Recall from equation (1) that the the bottom $n - k$ rows of eigenvector z_k are 0, so the last $n - k$ columns of Q are not needed for the gemm. Therefore, we start from $k = n$ and work down to $k = 1$, writing each block of eigenvectors to V over columns of Q after they are no longer needed.

The real case is similar, but has the minor complication that complex conjugate pairs of eigenvalues will generate conjugate pairs of eigenvectors, $z_k = a + bi$ and $\bar{z}_k = a - bi$, which are stored as two columns, a and b, in Z. When the first eigenvalue of each pair is encountered, both columns are computed; then the next eigenvalue (its conjugate) is skipped. Once $n_b - 1$ columns are processed, if the next eigenvector is complex it must be delayed until the next block.

4 Multi-threading triangular solver

After blocking the back-transform, the triangular solver remains a major bottleneck because it is not parallelized. Recall that the triangular matrix being solved is different for each eigenvector — the diagonal is modified by subtracting λ_k. This prevents blocking multiple eigenvectors together using a Level 3 BLAS trsm operation to solve multiple eigenvectors together.

In the complex case, LAPACK's ztrevc uses a safe triangular solver, zlatrs. Unlike the standard ztrsv BLAS routine, zlatrs uses column scaling to avoid numerical instability, and handles singular triangular matrices. Therefore, to not jeopardize the accuracy or stability of the eigensolver, we continue to rely on zlatrs instead of the optimized, multi-threaded ztrsv. However, processing T column-by-column prevents parallelizing individual calls to zlatrs, as is typically done for BLAS functions. Instead, we observe that multiple triangular solves could occur in parallel. One obstacle is that a different λ_k is subtracted from the diagonal in each case, modifying T in memory. Our solution is to write a modified routine, zlatrsd (triangular solve with modified diagonal), which takes both the original unmodified T_{11} and the λ_k to subtract from the diagonal. The subtraction is done as the diagonal elements are used, without modifying T in

Algorithm 1 Multi-threaded eigenvector computation (complex-arithmetic).

function ztrevc(n, T, V)
    // T is $n \times n$ upper triangular matrix.
    // V is $n \times n$ matrix; on input $V = Q$, on output V has eigenvectors.
    // Z and $\tilde{Z}$ are $n \times n_b$ workspaces, n_b is column blocksize.
    $k = n$
    while $k \geq 1$
        $j = n_b$
        while $j \geq 1$ and $k \geq 1$
            $\lambda_k = T_{k,\,k}$
            enqueue latrsd to solve $(T_{1:k-1,\,1:k-1} - \lambda_k I) Z_{1:k-1,\,j} = -T_{1:k-1,\,k}$
            $Z_{k:n,\,j} = [\,1,\,0,\,\ldots,\,0\,]^T$
            $j\ -= 1$; $k\ -= 1$
        end
        sync queue
        $m = k + n_b - j$
        for $i = 1$ to n by $\lceil n/\mathrm{p} \rceil$
            $i_2 = \min(i + n_b - 1,\ n)$
            enqueue gemm to multiply $\tilde{Z}_{i:i_2,\,j+1:n_b} = V_{i:i_2,\,1:m} * Z_{1:m,\,j+1:n_b}$
        end
        sync queue
        normalize vectors in $\tilde{Z}$ and copy to $V_{1:n,\,k+1:k+n_b}$
    end
end function

memory. This allows us to pass the same T to each zlatrsd call and hence solve multiple eigenvectors in parallel, one in each thread.

As previously mentioned, the real case requires a special quasi-triangular solver to solve each 2×2 diagonal block. In the original LAPACK code, this quasi-triangular solver is embedded in the dtrevc routine. To support multi-threading, we refactor it into a new routine, dlaqtrsd, a quasi-triangular solver with modified diagonal. Unlike the complex case, instead of passing λ_k separately, dlaqtrsd computes it directly from the diagonal block of T. If λ_k is real, dlaqtrsd computes a single real eigenvector. If λ_k is one of a complex-conjugate pair, dlaqtrsd computes a complex eigenvector, as two real vectors.

To deal with multi-threading, we use a thread pool design pattern. As shown in Algorithm 1, the main thread inserts latrsd tasks into a queue. Worker threads pull tasks out of the queue and execute them. For this application, there are no dependencies to be tracked between the triangular solves. After a block of n_b vectors has been computed, we back-transform them with a gemm. We could call a multi-threaded gemm, as available in MKL, but to simplify thread management and avoid switching between our pthreads and MKL's threads, we found it more suitable to use the same thread pool for the gemm as for latrsd. For p threads, the gemm is split into p tasks, each task multiplying a single block row of Q with Z. After the gemm, the next block of n_b vectors is computed. Within each thread, the BLAS calls are single threaded.

5 GPU Acceleration

To further accelerate the eigenvector computation, we observe that the triangular solves and the back-transformation gemm can be done in parallel. In particular, the gemm can be done on a GPU while the CPU performs the triangular solves. Data transfers can also be done asynchronously and overlapped with the triangular solves. To facilitate this asynchronous computation, we double-buffer the CPU workspace Z, using it to store results from latrsd, then swapping with $\tilde{Z}$, which is used to send data to the GPU while the next block of latrsd solves are performed with Z. The difference from Algorithm 1 is shown in Algorithm 2.

Algorithm 2 GPU accelerated back-transformation replaces lines 15–21 of Algorithm 1. Also, V is sent asynchronously to dV at start of ztrevc.

15: // dV is $n \times n$ workspace on GPU.
16: // dZ and $d\tilde{Z}$ are $n \times n_b$ workspaces on GPU.
17: swap buffers Z and $\tilde{Z}$
18: async send $\tilde{Z}$ to dZ on GPU
19: async gemm $d\tilde{Z}_{1:n,\ j+1:n_b} = dV_{1:n,\ 1:m} * dZ_{1:m,\ j+1:n_b}$ on GPU
20: async receive $d\tilde{Z}$ to $V_{1:n,\ k+1:k+n_b}$ on CPU
21: normalize vectors in V

6 Results

We performed tests with two 8-core, 2.6 GHz Intel E5-2670 CPUs and an 875 MHz NVIDIA K40 GPU, with Intel MKL 11.0.5 for optimized, multi-threaded BLAS. Matrices were double precision with uniform random entries in $(0, 1)$. Inside our parallel trevc, we launch p pthreads and set MKL to be single-threaded; outside trevc, MKL uses the same number of threads, p.

Fig. 1 shows the total eigenvalue problem (dgeev) time, broken down into four phases: QR iteration (bottom, green tier), Hessenberg reduction (2nd, cyan tier), triangular solves (3rd, blue tier), and back-transformation (top, red tier). The triangular solves and back-transformation together form the eigenvector computation. Columns are grouped by number of CPU threads. We compare results to two reference implementations: the LAPACK CPU version in the first column, and the MAGMA [8] GPU-accelerated version in the fourth column.

The improvement due to blocking the back-transformation is shown by the second column of each group in Fig. 1. It is up to 14 times faster than the non-blocked back-transformation using 16 threads. Further, the solid red line (squares) in Fig. 2 show it has better parallel scaling, reaching a speedup of 12 times for 16 cores, compared to only 6 times for the LAPACK implementation. However, as the triangular solves are not yet parallelized, the overall improvement is limited, being at most 1.4 times faster, seen in the single threaded result in Fig. 1, and the red line (squares) in Fig. 3.

Parallelizing the triangular solves is the third column of each group in Fig. 1. For one thread, there is of course no parallel speedup, so the results are the same as the second column. With multiple threads, we see significant parallel

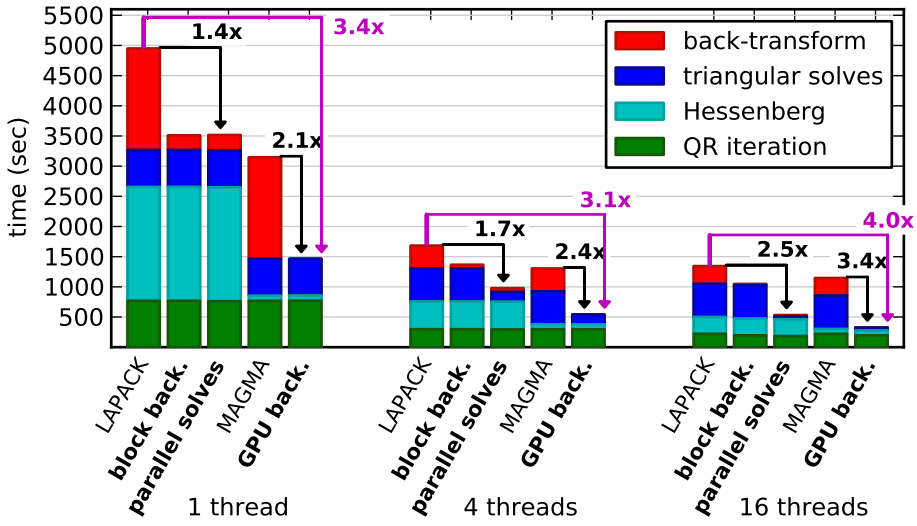

Fig. 1. Execution time of eigenvalue solver (dgeev) for matrix size $n = 16{,}000$.

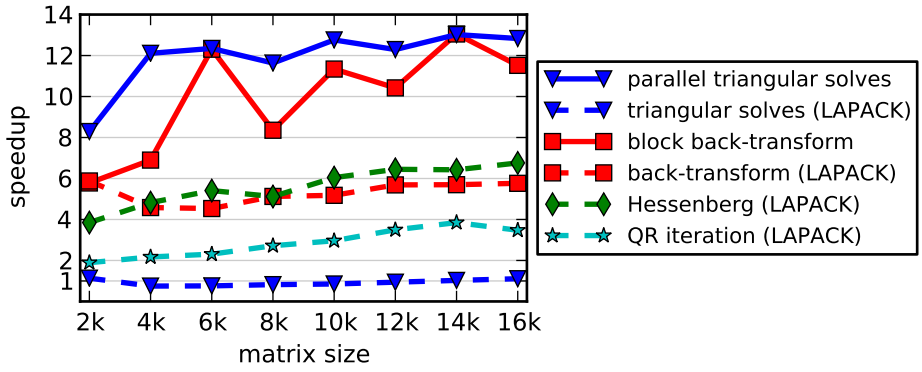

Fig. 2. Parallel speedup of each phase by itself, for $p = 16$, compared to $p = 1$.

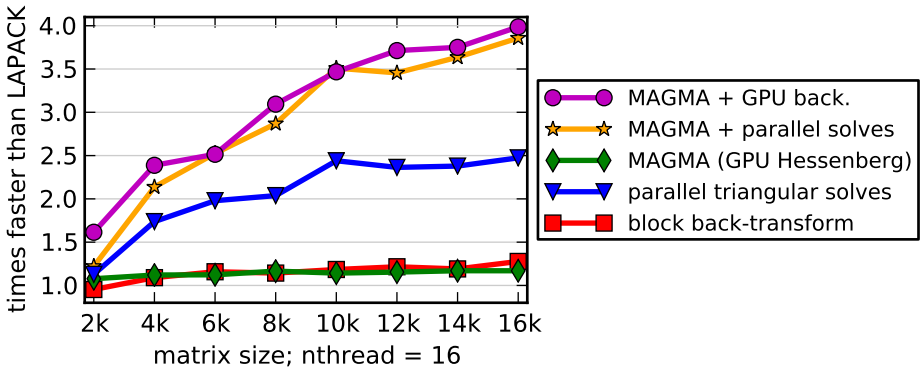

Fig. 3. Overall improvement of eigenvalue solver (dgeev) compared to LAPACK, after various improvements, using $p = 16$ threads.

speedup, up to 12.8 times for 16 threads, shown as the solid blue line (triangles) in Fig. 2. Combined with the blocked back-transformation, these two modifications significantly improve the solution of the overall eigenvalue problem by up to 2.5 times for 16 cores, shown by the blue line (triangles) in Fig. 3, and annotated with an arrow in Fig. 1. This is the total improvement available using only CPUs. Next we will look at the improvement also using GPUs.

The fourth column in Fig. 1 is the MAGMA reference time, which accelerates the Hessenberg reduction using the GPU. The MAGMA Hessenberg performance depends on the GPU, so is independent of the number of CPU threads. It is up to 3.4 times faster than the 16-core LAPACK Hessenberg. However, the LAPACK Hessenberg is only 20% of the total time, so accelerating it reduces the total dgeev time by only 1.15 times, shown by the green line (diamonds) in Fig. 3. When combined with the new blocked back-transform and parallel triangular solves, the performance substantially increases, as shown by the orange line (stars) in Fig. 3, being up to 3.8 times faster than LAPACK.

Our final improvement is to move the back-transformation gemm to the GPU, shown as the fifth column of each group in Fig. 1. The gemm can be almost entirely overlapped with the triangular solves, practically eliminating time spent on the back-transformation. This is a minor additional improvement on top of the blocked back-transform and parallel solves, shown by the magenta line (circles) in Fig. 3. The improvement from the MAGMA version using 16 cores is 3.4 times, while from the LAPACK version is 4.0 times, as annotated in Fig. 1.

7 Conclusion

It has been said that high performance computing is an exercise in chasing bottlenecks. Previously, the Hessenberg reduction and QR iteration have rightly been addressed as major bottlenecks in the solution of the nonsymmetric eigenvalue problem. Amdahl's Law requires that all phases of the algorithm receive attention. Indeed, while the Hessenberg was accelerated by 3.4 times with a GPU, the overall speedup was previously limited to 15% (Fig. 3). We accelerated the remaining eigenvector computation phase by 12 times through introducing Level 3 BLAS and parallelizing the remaining Level 2 BLAS triangular solves. This improved the overall eigenvalue problem by 2.5 times for CPU-only code and 3.5 times for the GPU-accelerated version. The bottleneck is now moved back to QR iteration, which is natural as it has the most flops and its iterative nature makes it the most complicated and difficult phase to parallelize.

References

1. C. Bischof. A summary of block schemes for reducing a general matrix to Hessenberg form. Technical report, ANL/MCS-TM-175, Argonne National Laboratory, 1993.
2. C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):s2–s13, 1987.
3. K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. part I: Maintaining well-focused shifts and level 3 performance. *SIAM Journal on Matrix Analysis and Applications*, 23(4):929–947, 2002.
4. K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. part II: Aggressive early deflation. *SIAM Journal on Matrix Analysis and Applications*, 23(4):948–973, 2002.
5. G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins, third edition, 1996.
6. B. Kågström, D. Kressner, and M. Shao. On aggressive early deflation in parallel variants of the QR algorithm. In *Applied Parallel and Scientific Computing*, pages 1–10. Springer, 2012.
7. L. Karlsson and B. Kågström. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Computing*, 37(12):771–782, 2011.
8. MAGMA. http://icl.eecs.utk.edu/magma/.
9. S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.