Performance Analysis and Design of a Hessenberg Reduction using Stabilized Blocked Elementary Transformations for New Architectures

Khairul Kabir
University of Tennessee
kkabir@vols.utk.edu

Azzam Haidar
University of Tennessee
haidar@icl.utk.edu

Stanimire Tomov
University of Tennessee
tomov@icl.utk.edu

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester
dongarra@icl.utk.edu

ABSTRACT

The solution of nonsymmetric eigenvalue problems, $Ax = \lambda x$, can be accelerated substantially by first reducing A to an upper Hessenberg matrix H that has the same eigenvalues as A. This can be done using Householder orthogonal transformations, which is a well established standard, or stabilized elementary transformations. The latter approach, although having half the flops of the former, has been used less in practice, e.g., on computer architectures with well developed hierarchical memories, because of its memory-bound operations and the complexity in stabilizing it. In this paper we revisit the stabilized elementary transformations approach in the context of new architectures – both multicore CPUs and Xeon Phi coprocessors. We derive for a first time a blocking version of the algorithm. The blocked version reduces the memory-bound operations and we analyze its performance. A performance model is developed that shows the limitations of both approaches. The competitiveness of using stabilized elementary transformations has been quantified, highlighting that it can be 20 to 30% faster on current high-end multicore CPUs and Xeon Phi coprocessors.

ACM Classification Keywords

G.1.3 Numerical Analysis: Numerical Linear Algebra—*Eigenvalues and eigenvectors*

Author Keywords

Eigenvalues problem, Hessenberg reduction, Stabilized Elementary Transformations, Multi/Many-core

INTRODUCTION

Eigenvalue problems are fundamental for many engineering and physics applications. For example, image processing, compression, facial recognition, vibrational analysis of mechanical structures, and computing energy levels of electrons in nanostructure materials can all be expressed as eigenvalue problems. The solution of these problems, in particular for nonsymmetric matrices that is of interest to this work, can be accelerated substantially by first reducing the matrix at hand to an upper Hessenberg matrix that has the same eigenvalues as the original one (see Section). This can be done in several ways, e.g., using Householder transformations, which is a well established standard, through elementary orthogonal transformations, or stabilized elementary transformations. The latter approach, although having half the flops of the Householder Hessenberg, has been used less in practice because of its memory-bound operations and the complexity in stabilizing it. In this paper we revisit the stabilized elementary transformations approach in the context of multicore CPUs and Xeon Phi coprocessors.

The reduction approach can be used for other two sided-factorizations as well, e.g., in the tridiagonal reduction algorithm for symmetric eigenvalue problems, or in the bidiagonal reduction for singular value decomposition problems. Besides applications based on the nonsymmetric eigenvalue problem, the Hessenberg reduction is applicable to other areas that exploit for example the fact that the powering of a Hessenberg matrix and solving a Hessenberg system of equations is cheap compared to corresponding algorithms for general matrices[22].

It is challenging to accelerate the two-sided factorizations on new architectures because they are rich in Level 2 BLAS operations, which are bandwidth limited and therefore do not scale on multicore architectures and run only at a fraction of the machine's peak performance. There are techniques that can replace Level 2 BLAS operations with Level 3 BLAS. For example, in factorizations like LU, QR, and Cholesky, the application of consecutive Level 2 BLAS operations that oc-

HPC 2015, April 12-15, 2015, Alexandria, VA.

cur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation be applied at once as a Level 3 BLAS (see LAPACK [1]). This approach totally removes Level 2 BLAS from Cholesky, and reduces its amount to $O(n^2)$ in LU, and QR, thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to the Hessenberg reduction based on orthogonal transformations[6], but in contrast to the one-sided factorizations, it still leaves about 20% of the total number of operations as Level 2 BLAS. In practice, these 20% of Level 2 BLAS do not scale well on current architectures and dominate the total execution time. Therefore, a very important aspect in enabling the Hessenberg reduction using stabilized elementary transformations to run efficiently on new architectures, is to what extend blocking can be applied, and what is the number of flops remaining in Level 2 BLAS.

Besides the algorithmic and performance modeling aspects related to the importance of reducing the Level 2 BLAS flops, this work is also focused on the computational challenges of developing high-performance routines for new architectures. We describe a number of optimizations that lead to performance as high as 95% of the theoretical/model peak for multicore CPUs and 80% of the model peak for Intel Xeon Phi coprocessors. These numbers are indicative for a high level of optimization achieved – note that the use of accelerators is known to achieve smaller fraction of the peak compared to non-accelerated systems, e.g., for the Top500 HPL benchmark for GPU/MIC-based supercomputers this is about 60% of the peak, and for LU on single coprocessor is about 70% of the peak [4].

BACKGROUND

The eigenvalue problem is to find an eigenvector x and eigenvalue λ that satisfy

$$Ax = \lambda x,$$

where A is a symmetric or nonsymmetric $n \times n$ matrix. When the full eigenvalue decomposition is computed we have

$$A = X \Lambda X^{-1},$$

where Λ is a diagonal matrix of the eigenvalues and X is a matrix of the eigenvectors of A.

In general, solving the eigenvalue problem can be split into three main phases:

1. **Reduction phase**: orthogonal matrices Q are applied on both the left and the right side of A to reduce it to a condensed form matrix – hence these are called "two-sided factorizations." Note that the use of two-sided orthogonal transformations guarantees that A has the same eigenvalues as the reduced matrix, and the eigenvectors of A can be easily derived from those of the reduced matrix (step 3);
2. **Solution phase**: an eigenvalue solver further computes the eigenpairs Λ and Z of the condensed form matrix;
3. **Back transformation phase**: if required, the eigenvectors of A are computed by multiplying Z by the orthogonal matrices used in the reduction phase.

In this paper we are interested in the nonsymmetric eigenvalue problem. Thus, the reduction phase reduces the nonsymmetric matrix A to an upper Hessenberg form,

$$H = Q^T A Q.$$

For the second phase, QR iteration is used to find the eigenpairs of the reduced Hessenberg matrix H by further reducing it to (quasi) upper triangular Schur form, $S = E^T H E$. Since S is in a (quasi) upper triangular form, its eigenvalues are on its diagonal and its eigenvectors Z can be easily derived. Thus, A can be expressed as:

$$A = QHQ^T = Q\ E\ S\ E^T\ Q^T,$$

which reveals that the eigenvalues of A are those of S, and the eigenvectors Z of S can be back-transformed to eigenvectors of A as $X = Q\ E\ Z$.

There are many ways to formulate mathematically and solve these problems numerically, but in all cases, designing an efficient computation is challenging because of the nature of the algorithms. In particular, the orthogonal transformations applied to the matrix are two-sided, i.e., transformations are applied on both the left and right side of the matrix. This creates data dependencies that prevent the use of standard techniques to increase the computational intensity of the computation, such as blocking and look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations. Thus, the reduction phase can take a large portion of the overall time, and it is very important to identify its bottlenecks. The classical approach (LAPACK algorithms dgehrd) to reduce a matrix to Hessenberg form is to use the Householder reflectors [24]. The computational complexity of this procedure is about $\frac{10n^3}{3}$.

The reduction to Hessenberg, besides the use of orthogonal transformations based on Householder reflectors, may also be achieved in a stable manner by either stabilized elementary matrices or elementary unitary matrices [18, 12]. This later approach reduces the computational cost by half. In this paper we study and focus on the reduction to Hessenberg using elementary matrices. We revisit the algorithm as well as we accelerate it by implementing a blocked version on both multicore CPUs and Xeon Phi coprocessors. We also revisited the use of elementary matrices to reduce the general matrix to tridiagonal form as described in [8].

Note that in this approach the transformations used in the reduction phase are not orthogonal anymore as explained for the general case at the beginning, and therefore Q^T is replaced by the inverse of the elementary transformation at hand, so that the reduced and the original matrix still have the same eigenvalues (see next).

RELATED WORK

The earliest standard method for computing the eigenvalues of a dense nonsymmetric matrix is based on the QR iteration algorithm [7]. This schema is prohibitively expensive compared to a two phases scheme that first reduces the matrix to Hessenberg form (using either elementary or orthogonal similarity transformations), and then uses a few QR iterations

to compute the eigenvalues of the reduced matrix. This two phase approach using Householder reflectors [24] was implemented in the standard EISPACK software [5]. Blocking was introduced in LAPACK, where a product of Householder reflectors $H_i = I - \tau_i v_i v_i^T$, $i = 1, \ldots, nb$ were grouped together using the so called *compact WY transform* [2, 20]:

$$H_1 H_2 \ldots H_{nb} \equiv I - VTV^T,$$

where nb is the blocking size, $V = (v_1 | \ldots | v_{nb})$, and T is $nb \times nb$ upper triangular matrix.

Alternatively to the Householder reflector approach, the use of stabilized elementary matrices for the Hessenberg reduction has been well known [18]. Later [8] proposed a new variant that reduce the general matrix further to tridiagonal form. The main motivation was that iterating with a tridiagonal form is attractive and extremely beneficial for non symmetric matrices. However, there was two major difficulty here, first is that the QR iteration does not maintain the tridiagonal form of a nonsymmetric matrix and second reducing the nonsymmetric matrix to tridiagonal by similarity transformations encounter stability and numerical issues. To overcome the first issue, [19] proposed the LR iteration algorithm which preserves the tridiagonal form. [8] proposed some recovery techniques in his paper and later [12, 23] proposed another variant that reduce the nonsymmetric matrix to a similar banded form and [13] provided an error analysis of its BHESS algorithm. Up to our knowledge, blocking to the stabilized elementary reduction is introduced in this paper, similar to the blocking for the one-sided LU, QR and Cholesky factorizations (see Section).

A hybrid Hessenberg reduction that uses both multicore CPUs and GPUs was introduced first through the MAGMA library [21]. The critical for the performance Level 2 BLAS were offloaded for execution to the high-bandwidth GPU and proper data mapping and task scheduling was applied to reduce CPU-to-GPU communications.

Recent algorithmic work on the two-sided factorizations has been concentrated on two- (or more) stage approaches. In contrast to the standard approach from LAPACK that uses a "single stage", the new ones first reduce the matrix to band form, and second, to the final form, e.g., tridiagonal for symmetric matrices. One of the first uses of a two-step reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [9], where a multistage method reduced a matrix to tridiagonal, bidiagonal, and Hessenberg forms [17]. With this approach, it was possible to recast the expensive memory-bound operations that occur during the panel factorization into a compute-bound procedure. Consequently, a framework called Successive Band Reductions (SBR) was created [3]. A multi-stage approach has also been applied to the Hessenberg reduction [16] as well as the QZ algorithm [15] for the generalized non-symmetric eigenvalue problem. These approaches were also developed for hybrid GPU-CPU systems [11].

ALGORITHMIC ADVANCEMENTS

In this section we describe the Hessenberg reduction algorithm that uses stabilized elementary matrices. A nonsymmetric $n \times n$ matrix A is reduced to upper Hessenberg form H by stabilized elementary matrices in $n-2$ steps. At step k the original matrix $A_1 \equiv A$ is reduced to A_{k+1} which is in upper Hessenberg form in its first k columns. Applying elementary transformation matrix L_{k+1} from the right, and then L_{k+1}^{-1} from the left to A_k introduces zeros below the first subdiagonal of column k and generates A_{k+1} by updating columns $k+1, \ldots, n$ of A_k. The algorithm is performed in-place where A_{k+1} overwrites A_k and elementary transformation matrix L_{k+1} can be stored in A_{k+1}. The relationship between A_{k+1} and A_k is expressed as,

$$A_{k+1} = L_{k+1}^{-1} A_k L_{k+1}. \tag{1}$$

The elementary transformation matrix L_{k+1} is defined as

$$L_{k+1} \equiv (I + l_{k+1} e_{k+1}^*),$$

where $l_{k+1} = [0, ..., 0, l_{k+2}, ..., l_n]^T$ with $l_i = A_{i,k}/A_{k+1,k}$ for $i = k+2, ..., n$, and $e_{k+1}^* = [0, ..., p_{k+1}, 0, ..., 0]$ with $p_{k+1} = 1$. The inverse of L_{k+1} is

$$L_{k+1}^{-1} = (I - l_{k+1} e_{k+1}^*),$$

as one can easily check that indeed $L_{k+1} L_{k+1}^{-1} = I$. Certain permutations can be introduced to stabilize the reduction, leading to the following reformulation of equation (1):

$$A_{k+1} = L_{k+1}^{-1} P_{k+1}\ A_k\ P_{k+1} L_{k+1}. \tag{2}$$

Here P_{k+1} is an elementary permutation matrix. For simplicity of the explanation, we can ignore the permutation matrix for the rest of the analysis. Namely, we can rewrite (1) as:

$$\begin{aligned} A_{k+1} = L_{k+1}^{-1} A_k L_{k+1} \ &= \ L_{k+1}^{-1} \ldots L_2^{-1}\ A_1\ L_2 \ldots L_{k+1} \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) \\ &\quad A_1 (I + l_2 e_2^*)(I + l_3 e_3^*) \ldots (I + l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) \\ &\quad A (I + l_2 e_2^*)(I + l_3 e_3^*) \ldots (I + l_{k+1} e_{k+1}^*) \end{aligned} \tag{3}$$

Since $e_k^*\ l_{k+1} = 0$, $(I + l_2 e_2^*)(I + l_3 e_3^*) \ldots (I + l_{k+1} e_{k+1}^*) = (I\ +\ l_2\ e_2^* + l_3 e_3^* + \cdots + l_{k+1} e_{k+1}^*)$. Therefore, (3) becomes:

$$\begin{aligned} A_{k+1} &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) \\ &\quad A (I + l_2 e_2^* + l_3 e_3^* + \cdots + l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) \\ &\quad (A + A l_2 e_2^* + A l_3 e_3^* + \cdots + A l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) B \ = \ R. \end{aligned} \tag{4}$$

Here $B = (A + A l_2 e_2^* + A l_3 e_3^* + \cdots + A l_{k+1} e_{k+1}^*)$ and $R = (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) B$. Let $L_{2:(k+1)}$ be the product of the elementary transformations $L_2 L_3 \ldots L_{k+1}$. Then,

$$R = (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \ldots (I - l_2 e_2^*) B$$

is written as,

$$B = (I + l_2 e_2^*)(I + l_3 e_3^*)\dots(I + l_{k+1} e_{k+1}^*)R$$
$$= (I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*)R = L_{2:(k+1)}R. \quad (5)$$

Based on (4) and (5), we can derive the blocked version of the reduction algorithm. Now the product of elementary transformation matrices, $L_{2:(k+1)}$ is partitioned as:

$$\begin{aligned} L_{2:(k+1)} &= L_2 L_3 \dots L_{k+1} \\ &= (I + l_2 e_2^*)(I + l_3 e_3^*)\dots(I + l_{k+1} e_{k+1}^*) \\ &= (I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*) \\ &= \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & l_{3,2} & 1 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & l_{4,2} & l_{4,3} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & l_{k+1,2} & l_{k+1,3} & \cdots & 1 & 0 & \cdots & 0 \\ 0 & l_{k+2,2} & l_{k+2,3} & \cdots & l_{k+2,k} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & l_{n,2} & l_{n,3} & \cdots & l_{n,k} & 0 & \cdots & 1 \end{pmatrix} \\ &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \end{aligned}$$

If we partition B and R matrix as well we can rewrite equation (5) for block matrix as,

$$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \times \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix}$$

If $[B_{11}B_{21}]^T$ is in upper Hessenberg form we can update $[R_{12}R_{22}]^T$ as follows,

$$L_{11}R_{12} = B_{12}$$
$$and\ B_{22} = R_{22} + L_{21}R_{12}$$
$$=> R_{22} = B_{22} - L_{21}R_{12}$$

Block R_{12} is computed using triangular solve and block R_{22} is updated using matrix multiplication. After k steps the original matrix A is replaced by matrix R where it is in upper Hessenberg form in its first k column.

$$\begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix}^{-1} \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ A_{21}^{(1)} & A_{22}^{(1)} \end{bmatrix} \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix}$$
$$= \begin{bmatrix} R_{11} & R_{12} \\ R_{12} & R_{22} \end{bmatrix}$$

Here $[R_{11}R_{12}]^T$ is in upper Hessenberg form. To reduce the rest of the matrix, A_{22}^{k+1} we proceed the same way as above and repartition $A^{(k+1)}$ as follows,

$$\begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} & A_{23}^{(k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(k+1)} & A_{33}^{(k+1)} \end{bmatrix}.$$

Then, in next k steps, $[A_{22}^{(k+1)}A_{32}^{(k+1)}]^T$ is reduced to upper Hessenberg form and the trailing matrix $[A_{23}^{(k+1)}A_{33}^{(k+1)}]^T$ is updated in the same way as A_{22}^{k+1} was updated after the first k steps. When we worked on $[A_{22}^{(k+1)}A_{32}^{(k+1)}]^T$, the reduction does not have any impact on $[A_{11}^{(k+1)}A_{21}^{(k+1)}A_{31}^{k+1}]^T$. Then, after $2k$ steps, $A = A_1$ is updated by A_{2k+1} as follows,

$$\begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} & A_{13}^{(1)} \\ A_{21}^{(1)} & A_{22}^{(1)} & A_{23}^{(1)} \\ A_{31}^{(1)} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} & A_{23}^{(k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(k+1)} & A_{33}^{(k+1)} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(2k+1)} & A_{23}^{(2k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(2k+1)} & A_{33}^{(2k+1)} \end{bmatrix}.$$

We have not updated $[A_{12}^{(k+1)}A_{13}^{(k+1)}]$ yet – the application of $L_{(k+2):(2k+1)}^{-1}$ from the left does not impact it, while the application of $L_{(k+2):(2k+1)}$ from the right has the following effect:

$$\begin{bmatrix} A_{12}^{(2k+1)} & A_{13}^{(2k+1)} \end{bmatrix} = \begin{bmatrix} A_{12}^{(k+1)} & A_{13}^{(k+1)} \end{bmatrix} \times \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix}$$
$$= \begin{bmatrix} A_{12}^{(k+1)}L_{22} + A_{13}^{(k+1)}L_{32} & A_{13}^{(k+1)} \end{bmatrix} \quad (6)$$

To summarize the description, we give the pseudo-code of the reduction for the non-blocked and the blocked version in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Non-blocked algorithm of Hessenberg reduction using elementary transformation.

for $k = 1$ *to* $n - 2$ *by* 1 **do**
- find max in $A(k+1:n, k)$ and j is its index
- Interchange rows $k+1$ and j
- Interchange columns $k+1$ and j
- $A(k+2:n, k) = A(k+2:n, k)/A(k+1, k)$ (xscal)
- $A(1:n, k+1) += A(1:n, k+2:n)A(k+2:n, k)$ (xgemv)
- $A(k+2:n, k+1:n) -= A(k+2:n, k)A(k+1, k+1:n)$ (xger)

OPTIMIZATIONS AND PERFORMANCE ANALYSIS

We benchmark our implementations on an Intel multicore system with two 8-core Intel Xeon E5-2670 (Sandy Bridge) CPUs, running at 2.6 GHz. Each CPU has a 24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1 caches. The system is equipped with 52 GB of memory. The theoretical peak in double precision is 20.8 Gflop/s per core, giving 332 Gflop/s in total. For the accelerators experiments, we used an Intel Xeon-Phi KNC 7120 coprocessor. It has 15.1 GB, runs at 1.23 GHz, and yields a theoretical double precision peak of $1,208$ Gflop/s. We used the MPSS 2.1.5889-16 software stack, the icc compiler that comes with the composer_xe_2013_sp1.2.144 suite, and BLAS implementation from MKL (Math Kernel Library) 11.01.02 [14].

The EISPACK library has routine elmhes which computes the Hessenberg reduction using elementary transformations. This routine is a serial, non-blocked implementation, and

Algorithm 2: Blocked algorithm of Hessenberg reduction using elementary transformation.

for $k = 1$ *to* $n-2$ *by* nb **do**
  $nb = min(nb, n-k-1)$
  for $i = 1$ *to* nb *by* 1 **do**
    if $i > 1$ **then**
      $A(k+1:k+i-1, k+i-1) = L(1:i-1, 1:i-1)^{-1} A(k+1:k+i-1, k+i-1)$ (xtrsv)
      $A(k+i:n, k+i-1) -= A(k+i:n, k:k+i-2) A(k+1:k+i-1, k+i-1)$ (xgemv)
    find max in $A(k+i:n, k+i-1)$ and j is its index
    Interchange rows $k+i$ & j, columns $k+i$ & j
    $A(k+i+1:n, k+i-1) /= A(k+i, k+i-1)$ (xscal)
    $L(i,i) = 1$
    $L(i+1:n-k-i, i) = A(k+i+1:n, k+i-1)$
    $A(k:n, k+i) += A(k:n, k+i+1:n) A(k+i+1:n, k+i-1)$ (xgemv)
  if $k > 1$ **then**
    $A(1:k, k+1:k+nb) = A(1:k, k+1:n) L(1:n-k, 1:nb)$ (xgemm1)
  $A(k+1:k+nb, k+nb:n) = L(1:nb, 1:nb)^{-1} A(k+1:k+nb, k+nb:n)$ (xtrsm)
  $A(k+nb+1:n, k+nb:n) -= A(k+nb+1:n, k:k+nb-1) A(k+1:k+nb, k+nb:n)$ (xgemm2)

thus depends on the performance of the Level 2 BLAS operations. We implemented a similar serial version since we realized that the EISPACK version unrolls the BLAS operations and this might slow the code. Our serial version is about $3-5\times$ faster, as shown in Figure 1. This is due to the fact that we use optimized xgemv and xger kernels from the MKL library. In order to evaluate its performance in both serial and parallel, we developed a parallel nonblocked version as described in Algorithm 1. We illustrate in Figure 1 a comparison of the performance of the EISPACK routine (EISPACK_Elementary_nonblocked_CPU) and our nonblocked version (MAGMA_Elementary_nonblocked_CPU). As expected, both performances are limited by the performance of the Level 2 BLAS operations. Even if our nonblocked implementation is parallel (here using 16 threads), we can observes only around $4\times$ speedup. This is due to the fact that the parallel Level 2 BLAS performance is limited by the memory bandwith. As consequence, it is clear that further significant improvements can be obtained only through algorithmic redesign, as in the blocked version of the algorithm where we replace as many as possible Level 2 BLAS operations (xger) by Level 3 BLAS (xtrmm, xtrsm, xgemm). The performance of dgemm on our testing machine is around $20\times$ higher than the performance of dger, and since around 60% of the flops are in xgemm, we expect a maximum of $2.5\times$ speedup in double precision over the nonblocked version. Figure 1 shows the performance obtained from the parallel blocked implementation (MAGMA_Elementary_blocked_CPU), as described in Algorithm 2. Our blocking factor n_b is equal to 64. The gain obtained here is around $2.3\times$, which corresponds to our expectation. However, even though 60% of the flops are now in Level 3 BLAS, the overall performance in Gflop/s obtained (around 35 Gflop/s) remains low compared to the capability of the machine (> 300 Gflop/s).

Performance Bound Analysis

In order to evaluate if the obtained performance results are acceptable and to analyse if there is an opportunity to more improvement as well as to compare the cost of this approach to the classical Householder technique, we conducted a computational analysis of the reduction to Hessenberg using either the elementary or the Householder transformations. For simplicity we show the cost of double precision implementation but it is easily to derive the other precision.

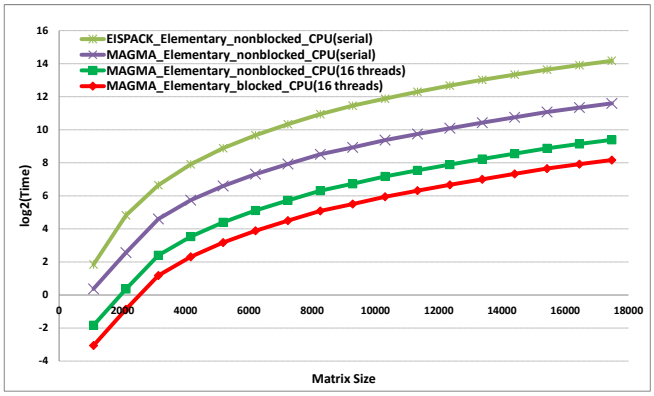

Figure 1. Performance of Hessenberg reduction using elementary transformation on CPU

Similar to the one-sided factorizations (Cholesky, LU, QR), the two-sided reduction to Hessenberg (either using Householder or Elementary) is split into a *panel factorization* and a *trailing matrix update*. Unlike the one-sided factorizations, the panel factorization requires computing Level 2 BLAS matrix-vector product (xgemv) involving the entire trailing matrix. This requires loading the entire trailing matrix into memory incurring a significant amount of memory bound operations. This creates data dependencies and produces artificial synchronization points between the panel factorization and the trailing submatrix update steps that prevent the use of standard techniques to increase the computational intensity of the computation, such as look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations. Let us compute the cost of the algorithm. The blocked implementation of the reduction proceeds by steps of size n_b where the cost of each step consists of the cost of the panel and the cost of the update.

- The panel is of size n_b columns. The factorization of every column involves one matrix-vector product (xgemv) with the trailing matrix that constitutes 95% of its operations. Thus the cost of a panel is $n_b \times 2l^2 + \Theta(n)$. Note that l is the size of the matrix at a step i. For simplicity, we omit $\Theta(n)$ and round the cost of the panel by the cost of the matrix-vector product.

- The update of the trailing matrix consists of applying either the Householder reflectors or the Elementary matrices generated during the panel factorization to the trailing matrix from both the left and the right side.

  For Householder: The update follows three steps, first and second are the application from the right and third is the application from left:

1- $A_{1:n,i+n_b:n} \leftarrow A_{1:n,i+n_b:n} - Y \times V^{\mathsf{T}}$ using dgemm,
2- $A_{1:i,i+1:i+n_b} \leftarrow A_{1:i,i+1:i+n_b} - Y_{1:i} \times V^{\mathsf{T}}$ using dtrmm,
3- $A_{i:n,i+n_b:n} \leftarrow A_{i:n,i+n_b:n}(I - V \times T^{\mathsf{T}} V^{\mathsf{T}})$ using dlarfb.
Its cost is $2n_b\ kn + n_b\ i^2 + 4n_b\ k\ (k + n_b)$ flops where $k = n - i - n_b$ is the size of the trailing matrix at a step i. Note that V, T, and Y are generated by the panel phase.

For Elementary: The update follows three steps:
1- $A_{1:i,i+1:i+n_b} \leftarrow A_{1:i,i+1:n} \times A_{i:n,i+1:i+n_b}$ using dgemm,
2- $A_{i+1:i+n_b,i+n_b:n} \leftarrow A^{-1}_{i+1:i+n_b,i+1:i+n_b} \times A_{i+1:i+n_b,i+n_b:n}$ using dtrsm,
3- $A_{i+n_b:n,i+n_b:n} \leftarrow A_{i+n_b:n,i+n_b:n} - A_{i+n_b:n,i+1:i+n_b} \times A_{i+1:i+n_b,i+n_b:n}$ using dgemm.
Its cost is $2n_b\ i\ (n - i) + n_b^2\ k + 2n_b\ k^2$ flops.

For all steps (n/n_b), the trailing matrix size varies from n to n_b by steps of n_b, where l varies from n to n_b and k varies from $(n - n_b)$ to $2n_b$. Thus the total cost for the n/n_b steps is:

For Householder:

$$\begin{aligned} &= 2n_b \sum_{n_b}^{n/n_b} l^2 + 2n_b n \sum_{2n_b}^{\frac{n-n_b}{n_b}} k + n_b \sum_{n_b}^{n/n_b} i^2 + 4n_b n \sum_{2n_b}^{\frac{n-n_b}{n_b}} k(k+n_b) \\ &= \tfrac{2}{3}n^3_{\text{dgemv}} + n^3_{\text{Level 3}} + \tfrac{1}{3}n^3_{\text{Level 3}} + \tfrac{4}{3}n^3_{\text{Level 3}} \\ &= \tfrac{10}{3}n^3 \quad flops. \end{aligned} \tag{7}$$

For Elementary:

$$\begin{aligned} &= 2n_b \sum_{n_b}^{n/n_b} l^2 + 2n_b \sum_{n_b}^{\frac{n-n_b}{n_b}} i(n-i) + n_b^2 \sum_{n_b}^{\frac{n-n_b}{n_b}} k + 2n_b n \sum_{2n_b}^{\frac{n-n_b}{n_b}} k^2 \\ &= \tfrac{2}{3}n^3_{\text{dgemv}} + \tfrac{1}{3}n^3_{\text{Level 3}} + \Theta(n^2) + \tfrac{2}{3}n^3_{\text{Level 3}} \\ &= \tfrac{5}{3}n^3 \quad flops. \end{aligned} \tag{8}$$

The maximum performance P_{max} that the reduction using elementary transformations can achieve is

$$\begin{aligned} P_{max} &= \frac{number\ of\ operations}{minimum\ time\ t_{min}} \\ &= \frac{\frac{5}{3}n^3}{t_{min}(\frac{2}{3}n^3\ flops\ in\ gemv) + t_{min}(\frac{3}{3}n^3\ flops\ in\ Level3)} \\ &= \frac{\frac{5}{3}n^3}{\frac{2}{3}n^3 * \frac{1}{P_{gemv}} + \frac{3}{3}n^3 * \frac{1}{P_{Level3}}} \\ &= \frac{5 * P_{Level3} * P_{gemv}}{2 * P_{Level3} + 3P_{gemv}} \approx \frac{5}{2} * P_{gemv} \quad when \quad P_{Level3} \gg P_{gemv} \end{aligned}$$

Since the Level 3 BLAS operations are considered as compute bound while the Level 2 are considered as memory bound the gap in performance between these level is large enough such a way that allow us to consider that $P_{Level3} \gg P_{gemv}$ In our testing machine, the maximum performance of dgemv is about 14 Gflop/s while the performance of Level 3 BLAS is more than 280 Gflop/s. As consequence, the upper bound limit of the performance that the reduction to band algorithm can reach is always less than $2.5\times$ the performance of dgemv. We shows in Figure 2 the performance of the dgemv routine as well as the theoretical upper bound as described above and the performance of our Magma implementation of the Hessenberg reduction on CPU.

Optimization and design for accelerators

It is clear that the performance obtained from our CPU implementation reaches close to its theoretical bound and thus we believe that there is no more room for improvement. For that we decided to take advantage of accelerators that provide higher range of dgemv performance and thus we can expect that the reduction can be accelerated. For the Intel Xeon Phi the dgemv peak performance is around 39 Gflop/s while the dgemm is more than 800 Gflop/s. Since the dgemv is more than twice faster on Xeon-Phi we can expect more than $2\times$ speedup of the reduction on the coprocessor. To verify that, we implemented the reduction to Hessenberg algorithm using Elementary transformation on the Xeon-Phi coprocessor. The code on high level is the same, using the MAGMA MIC APIs [10] to offload the computation to the Xeon Phi. Only certain kernels, like the swapping, had to be specifically designed and optimized. For the other kernels we used MKL, which is highly optimized. We depict in Figure 3 the results in Gflop/s that our implementation achieves, its theoretical upper bound, as well as the performance of dgemv. Similarly our implementation reaches asymptotically its upper bound. The gap observed for the Xeon Phi is larger than the one observed for CPU. This is due to hierarchical cache effect and to the fact that the cost of some Level 1 operation is considered marginal on CPU while it is more expensive on Xeon Phi introducing this difference. We implemented an optimized xswap kernel since we need to swap both row and column at every step. In our parallel swap implementation we had to design our parallelism to force each set of thread to read/write data aligned with cache for optimal performance. This was useful for column swapping since data is coalescent in memory while for row swapping we had to think differently. For the row swapping we only swap the rows within the current panel and delay the remaining till the end of the panel factorization when we swap the remaining rows in parallel. We split the thread pool over the data so that every set of threads tries to read/write as much as possible data within the same bank of memory. Also another improvement we had to implement for the Xeon Phi, it is not always advantageous to use all the threads i.e 240 threads for Level 1 BLAS because of the overhead of OMP thread management as OMP puts active threads in sleep mode after certain period and wakes them up when necessary. Since the swap function is needed for every step (n times) this overhead become unaffordable. To reduce this overhead we have changed the number of threads dynamically based on the number of elements needed to be swapped.

Figure 4 shows the performance of the reduction on CPUs *vs.* Xeon Phi. As analyzed, the reduction on the Xeon-Phi is about $2.5\times$ faster than on the CPUs. The Xeon-Phi implementation is native (uses only the Phi) and thus the CPU is idle or can performs other work. We have implemented a hybrid version that uses both CPU and Phi. Since the reduction is bound by the dgemv performance and also since both the dgemv and the dgemm performance achieve higher

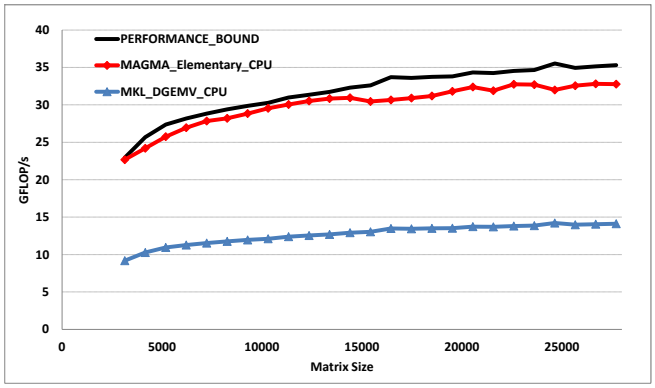

Figure 2. Performance bound for CPU

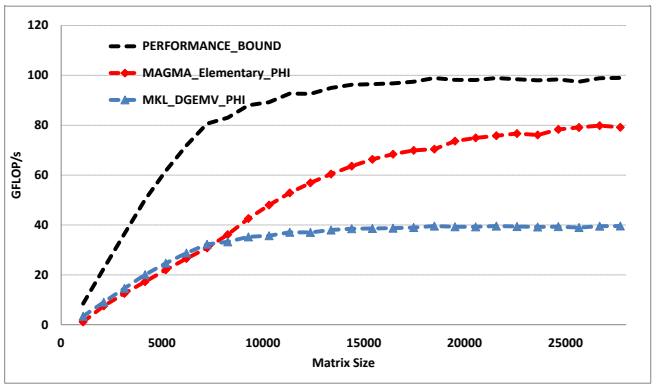

Figure 3. Performance bound for Intel Xeon Phi

ratio on the Phi, the hybrid implementation was around 5% slower and consumed more resources. For the hybrid version the dgemv and the dgemm are performed on the Phi, otherwise the performance drops down more than 15%. So, only the Level 1 BLAS operations are computed on the CPU since the CPU can handle better Level 1 routines. However, the cost of copying a vector back and forth between the CPU and the Phi at every step is negating the gain obtained and slows down the overall performance by 5%.

EXPERIMENT RESULTS

We performed a set of experiments to compare the performance of our proposed Hessenberg reduction using stabilized elementary transformations with the classical reduction that uses Householder transformations on both CPUs and Xeon-Phi. For the comparison we use the dgehrd routine from MKL for CPUs, as well as for the Phi in native mode (using only the Xeon-Phi). We illustrate in Figure 5 the required elapsed time in seconds to perform either the classical dgehrd or our MAGMA_Elementary on both CPU and Phi. First of all, we should mention that both implementations (Magma_Elementary or MKL_Householder) are optimized and reach their theoretical upper bounds on either CPUs or Phi. As expected, our proposed Elementary reduction implementation is between 20% to 30% faster than the

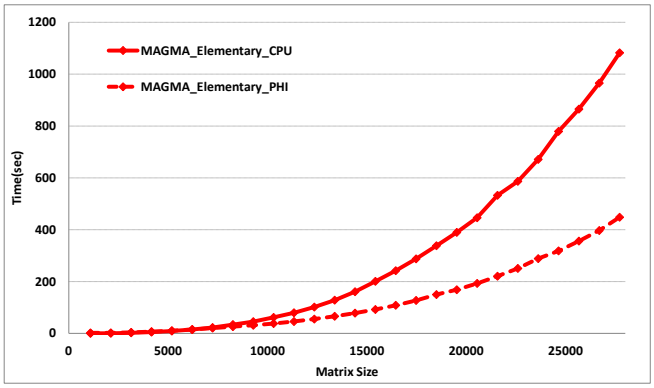

Figure 4. Accelerating the Hessenberg reduction on Xeon Phi

Householder one on either the CPU or Phi architecture. According to equations (7) and (8), the Elementary transformations reduce the amount of Level 3 BLAS operations by 62% while keeping the same amount of Level 2 operations. This results in reducing the overall cost of the Hessenberg reduction by 20%. The other optimizations that we have implemented, such as finding the pivot and directly scaling the corresponding vector at the same time as well as the optimized parallel row/column swapping, gave us an additional 5% to 10% improvement. Finally, our proposed Elementary implementation showed fast and efficient reduction to Hessenberg form and was accelerated by more than 2.3× by the use of the Xeon-Phi coprocessors.

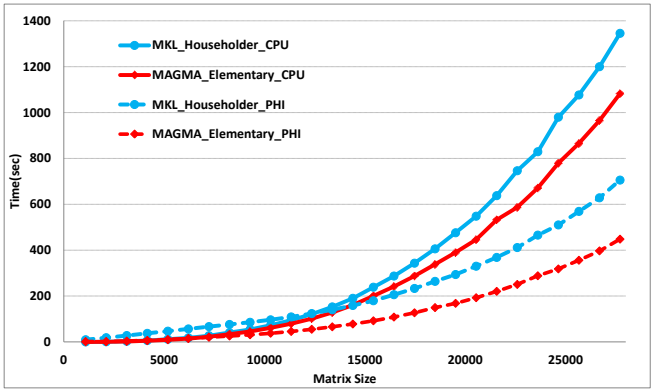

Figure 5. Performance comparison of CPUs *vs. Xeon Phi*

CONCLUSIONS AND FUTURE WORK

We derived for a first time a blocked algorithm for the reduction to upper Hessenberg form using stabilized elementary transformations and developed highly optimized implementations for multicore CPUs and Xeon Phi coprocessors. The blocking significantly imroved the performance of the approach, and even made it 20 to 30% higher than the standard, Householder-based approach. Still, both approaches are memory bound as they feature the same amount of flops in

Level 2 BLAS operations. We designed a model for the theoretical peak for both approaches that clearly shows their limitations, and also illustrates how optimal our implementations are. In particular, we reach up to 95% of the peak on multicore CPUs and up to about 80% on the Xeon Phi architecture. Our future work will explore the feasibility of using random butterfly transformations to avoid the need for pivoting, while still getting acceptable stability. Further, we will study the reduction of nonsymmetric matrices to band or tridiagonal form using elementary transformation.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, the Department of Energy, and Intel. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

REFERENCES

1. Anderson, E., Bai, Z., Bischof, C., Blackford, S. L., Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. *LAPACK User's Guide*, Third ed. Society for Industrial and Applied Mathematics, Philadelphia, 1999.
2. Bischof, C., and van Loan, C. The WY representation for products of Householder matrices. *J. Sci. Stat. Comput. 8* (1987), 2–13.
3. Bischof, C. H., Lang, B., and Sun, X. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Transactions on Mathematical Software 26*, 4 (2000), 602–616.
4. Dongarra, J., Gates, M., Haidar, A., Kabir, K., Luszczek, P., Tomov, S., and Yamazaki, I. MAGMA MIC 1.3 Release: Optimizing Linear Algebra for Applications on Intel Xeon Phi Coprocessors. `http://icl.cs.utk.edu/projectsfiles/magma/pubs/MAGMA_MIC_SC14.pdf`, Nov. 2014.
5. Dongarra, J. J., and Moler, C. B. EISPACK: A package for solving matrix eigenvalue problems. 1984, ch. 4, 68–87.
6. Dongarra, J. J., Sorensen, D. C., and Hammarling, S. J. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics 27*, 1-2 (1989), 215 – 227. Special Issue on Parallel Algorithms for Numerical Linear Algebra.
7. Francis, F. The QR transformation, part 2. *Computer Journal 4* (1961), 332–345.
8. Geist, G. Reduction of a general matrix to tridiagonal form. *SIAM J. Mat. Anal. Appl 12* (1991), 362–373.
9. Grimes, R. G., and Simon, H. D. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software 14* (September 1988), 241–256.
10. Haidar, A., Cao, C., Yarkhan, A., Luszczek, P., Tomov, S., Kabir, K., and Dongarra, J. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, IEEE Computer Society (Washington, DC, USA, 2014), 491–500.
11. Haidar, A., Tomov, S., Dongarra, J., Solca, R., and Schulthess, T. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications 28*, 2 (May 2014), 196–209.
12. Howell, G. W., and Diaa, N. Algorithm 841: Bhess: Gaussian reduction to a similar banded hessenberg form. *ACM Trans. Math. Softw. 31*, 1 (Mar. 2005), 166–185.
13. Howell, G. W., Geist, G., and Rowan, T. Error analysis of reduction to banded hessenberg form. *Tech. Rep. ORNL/TM-13344*.
14. Intel. Math kernel library. `https://software.intel.com/en-us/en-us/intel-mkl/`.
15. Kågström, B., Kressner, D., Quintana-Orti, E., and Quintana-Orti, G. Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. *BIT Numerical Mathematics 48* (2008), 563–584.
16. Karlsson, L., and Kågström, B. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Computing* (2011). DOI:10.1016/j.parco.2011.05.001.
17. Lang, B. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing 25*, 7 (1999), 845–860.
18. Martin, R., and Wilkinson, J. Similarity reduction of a general matrix to Hessenberg form. *Numerische Mathematik 12*, 5 (1968), 349–368.
19. Rutishauser, H. Solution of eigenvalue problems with the LR transformation. *Nat. Bur. Standards Appl. Math. Ser. 49* (1958), 47–81.
20. Schreiber, R., and van Loan, C. A storage-efficient WY representation for products of Householder transformations. *J. Sci. Stat. Comput. 10* (1991), 53–57.
21. Tomov, S., Nath, R., and Dongarra, J. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput. 36*, 12 (Dec. 2010), 645–654.
22. Van Loan, C. Using the hessenberg decomposition in control theory. 102–111.
23. Wachspress, E. L. similarity matrix reduction to banded form . *manuscript* (1995).
24. Wilkinson, J. H. Householder's method for the solution of the algebraic eigenproblem. *The Computer Journal 3*, 1 (1960), 23–27.