# MIXED-PRECISION ALGORITHM FOR FINDING SELECTED EIGENVALUES AND EIGENVECTORS OF SYMMETRIC AND HERMITIAN MATRICES *

YAOHUNG M. TSAI†, PIOTR LUSZCZEK‡, AND JACK DONGARRA§

**Abstract.** As the new hardware is being equipped with powerful low-precision capabilities driven primarily by the needs of the burgeoning field of Artificial Intelligence (AI), mixed-precision algorithms are now showing far greater potential and renewed interest in scientific computing community. The multi-precision methods commonly follow approximate-iterate scheme by first obtaining the approximate solution from a low-precision factorization and solve. Then, they iteratively refine the solution to the desired accuracy that is often as high as what is possible with traditional approaches. While targeting symmetric and Hermitian eigenvalue problems of the form $Ax = \lambda x$, we revisit the SICE algorithm proposed by Dongarra et al. By applying the Sherman-Morrison formula on the diagonally-shifted tridiagonal systems, we propose an updated SICE-SM algorithm. By incorporating the latest two-stage algorithms from the PLASMA and MAGMA software libraries for numerical linear algebra, we achieved up to 3.6× speedup using the mixed-precision eigensolver with the blocked SICE-SM algorithm for iterative refinement when compared with full double complex precision solvers for the cases with a portion of eigenvalues and eigenvectors requested.

**Key words.** mixed-precision algorithms, eigenvalue solver, hardware accelerators

**AMS subject classifications.** 65F15, 65F25, 65Y20, 68N01

## 1. Introduction.

The symmetric eigenvalue problem is one of the most important problems in numerical linear algebra for analysis of invariant subspace. For real matrices, the objective is to find an eigenvalue $\lambda$ and the corresponding eigenvector $x$ such that

$$Ax = \lambda x \text{ where } A = A^{\intercal},\ A \in \mathbb{R}^{n\times n} \tag{1.1}$$

The Hermitian eigenvalue problem is to find the eigenvalues and eigenvectors in complex domain. For an Hermitian matrix $A$, the conjugate transpose (adjoin) operation is idempotent: $A = A^*$ and the eigenvalues are real which implies shared properties with the symmetric eigenvalue problem in real domain.

As mixed-precision algorithms for solving a linear system of equations experienced a substantial interest that resulted in recent developments [10, 11, 25]. These were mostly driven by the introduction of new hardware platforms that provide increased low-precision performance for AI workloads. However, there was not as much focus on eigenvalue problems. And with the latest two-stage tridiagonalization approach [24, 26], the multicore and multi-GPU eigensolvers' algorithms for refining eigenvalues should be reviewed carefully in order to ascertain the possibility to improve the performance especially on this new hardware.

*This material is also based upon work supported by the National Science Foundation under OAC Grant No. 2004541 and the University of Tennessee grant MSE E01-1315-038 as Interdisciplinary Seed funding. This research used the computational resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 provided by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The software library integration work was supported by the National Science Foundation under OAC Grant No. 2004541.

†University of Tennessee (ytsai2@icl.utk.edu)

‡University of Tennessee (luszczek@icl.utk.edu)

§University of Tennessee, Oak Ridge National Laboratory, University of Manchester (dongarra@icl.utk.edu)

**2. Contribution.** The motivation of this paper is to develop a mixed-precision algorithm for the symmetric/hermitian eigenvalue problem. In this context, we made the following contributions:

- We formulted and implemented the SICE-SM algorithm for symmetric and Hermitian matrices using Sherman-Morrison formula to solve the resulting tridiagonal systems with rank-one updates based on the SICE algorithm proposed by Dongarra et al. for iteratively refining a pair of eigenvalues and corresponding eigenvectors.
- We also developed the blocked SICE-SM algorithm to refine multiple eigenvalues and corresponding eigenvectors simultaneously and improve the performance by aggregating the matrix-vector multiplication operations for improved utilization of the memory hierarchy.
- We implemented a mixed-precision algorithm by rearranging the tasks in the 2-stage eigensolver on the machines with heterogeneous architecture by improving the utilization of both the CPUs and GPUs.
- We achieved performance improvement of the mixed-precision algorithm with the iterative refinement for the case that only a portion of eigenpairs are requested.

## 3. Releated Work.

**3.1. Eigenvalue refinement.** Symm and Wilkinson[41] proposed an algorithm to determine the error bounds of computed eigenvalues and eigenvectors, which can also be used to improve the accuracy of a given eigen-pair. Dongarra, Moler, and Wilkinson[18, 19, 20] later improved the algorithm with reduced computational cost and provided additional error analysis, including the comparison to Newton's method[36, 47], numerical results, and discussion of extending the algorithm for ill-conditioned problems with multiple close eigenvalues. More detail will be reviewed in subsection 3.4 as it is also the core of the algorithm used in this work.

Other related work from Stewart[40] and Chatelin[13] answered the same question from the point of view of the invariant subspace problem. Demmel[16] later pointed out that these two methods and the one from Dongarra, Moler, and Wilkinson [18, 19, 20] can all be reduced to solving the same Riccati equation. He also extended the algorithm for the generalized eigenvalue problem of the form $Ax = \lambda Bx$.

Alefeld and Spreuer[3] followed the same approach but specifically targeted problems with doubly-repeated or numerically close eigenvalues. Tisseur[42] did the analysis of Newton's method under floating-point arithmetic for generalized eigenvalue problems. Prikopa and Gansterer[37] used the symmetry of the matrix and Householder tridiagonalization $A = QTQ^{\intercal}$ to reduce the computational cost.

Ogita and Aishima[31] proposed a different iterative scheme, which heavily relies on matrix-matrix multiplication for those applications which require accuracy that is higher than the base IEEE-754 double precision. The algorithm is applied on the entire spectrum of eigenvalues but it is capable of improving at the same time the orthogonality and eigenvalue accuracy. However, it requires high-precision computation for the most parts of the algorithm, making it costly in practice. Later the authors extended the algorithm for clustered eigenvalues and singular value decomposition[32, 33].

**3.2. Parallel Eigensolvers.** To build an efficient mixed-precision algorithm, the latest advances in parallel eigensolvers should also be incorporated. The symmetric dense eigensolvers are mainly composed of two phases: tridiagonal reduction and tridiagonal eigensolver. Firstly, through similarity transformations based on orthogonal/unitary matrices, the symmetric/Hermitian matrix is reduced to a tridiagonal form without altering the spectrum in infinite precision or with numerically stable perturbation in final precision. Then the problem is solved in tridiagonal form with much less cost than operating on a full matrix by applying different methods which will be described later in the section. If needed, the eigenvectors can be computed from the eigenvectors of the tridiagonal system and applying back-transformations of tridiagonal reduction.

**3.2.1. Tridiagonal Reduction.** The first phase is to convert a full dense matrix into upper Hessenberg form, which has zeros below the first subdiagonal. The real symmetric and complex Hermitian cases result in even better structured form: a symmetric tridiagonal matrix with only nonzeros on the diagonal, the first superdiagonal, and the first subdiagonal. The tridiagonalization of complex Hermitian matrix is usually chosen to be real tridiagonal symmetric matrix to reduce the computation cost in following steps. The Householder transformation is a natural choice for the reduction because of its simplicity and numerical stability. Furthermore, Dongarra et al.[21] introduced a blocked version of Householder vector application in which the transformations are aggregated and applied in a blocked fashion, so they can benefit from the high performance matrix-matrix multiplications rather than be bound by matrix-vector performance.

Bischof et al.[8] proposed the approach based on successive band reduction (SBR). Each reduction sweep results in a narrower band matrix, and the reduction is done via a bulge-chasing procedure. The algorithm consists of a series of sweeps: each sweep will zero-out one column below subdiagonal but create fill-ins down the diagonal as the transformations are applied to the remaining matrix. Then additional transformations are applied to zero out the fill-in which was just created and this is repeated all the way down to the lower-right corner until it disappears from the matrix, hence the algorithm name: the bulge chasing. The algorithm is naturally parallelizable as the subsequent sweeps can be chosen to not overlap with each other, making it especially suitable for multicore CPUs in shared-memory environments.

Later work introduced a hybrid 2-stage algorithm[24, 26]. The first stage still consisted of blocked Householder transformations but it only reduced the matrix to a band form. Then, the left transformation will only be needed, as the right transformation will not be touching the first block of columns. It thus becomes an $LQ$ factorization for the block of columns, which is much faster than applying the transformations from both sides (LQ and QR). The second stage uses the bulge-chasing algorithm from the successive band reductions.

**3.2.2. Tridiagonal Eigensolvers.** After tridiagonalization completes, a few standard eigensolver algorithms could be considered. As this is not the main focus of this work, these will only be reviewed briefly. The **QR algorithm** with shifts[46] is one of the most popular choices because of its superb stability and cubic convergence rate in general case. At each iteration, it computes a $QR$ factorization and multiplies them back in reverse order: $Q_k R_k = A_k - \mu_k I; A_{k+1} = R_k Q_k + \mu_k I$. There are other variants of QR iteration for strategically choosing the shifts $\mu_k$.

Another algorithm is called **divide and conquer**[14] that observes that with a rank-1 update, the initial problem can be divided into two independent subproblems with half the size. This results in repeatably reducing the problem down to the $1 \times 1$ case which admits a trivial solution. In practice, there is a threshold size and the implementation switches to another method for below-threshold sizes for better performance on small problems. The independent problems can easily be parallelized.

There are other methods based on the $LDL^\intercal$ factorization. The **Bisection method**[45] uses a suitable factorization to identify the number of eigenvalues present within a section and then it consecutively reduces the size of sections until the eigenvalues of interest are located with desired accuracy.

Finally, **Multiple relatively robust representations (MRRR)**[35] takes the bisection further by the theoretically estimating the gaps between neighboring eigenvalues. This algorithm divides the whole spectrum into clusters of eigenvalues that each have a relatively robust representation ($LDL^\intercal$ factorization).

### 3.3. Software Packages for Symmetric/Hermitian Eigenvalue Problems.

This section provides details on the software packages that are available for numerical linear algebra and include dense eigensolvers.

EISPACK[39] is one the earliest open source software libraries to solve eigenproblems. It contains subroutines for the following nine classes of matrices: complex general, complex Hermitian, real general, real symmetric, real symmetric banded, real symmetric tridiagonal, special real tridiagonal, generalized real, and generalized real symmetric matrices. Providing performance portability of EISPACK motivated establishment of Basic Linear Algebra Subprograms (BLAS)[29] as the standard building blocks for performing basic vector and matrix operations. BLAS was later extended to include three levels of operations: Level 1 scalar-vector and vector-vector, Level 2 matrix-vector, and Level 3 matrix-matrix. Availability of BLAS proliferated as almost all hardware vendors provided their own optimized implementations and thus unified interface for numerical linear algebra software became the de facto standard upon which more complex methods are implemented including eigensolvers. The vendor renditions of BLAS for particular architectures include Intel MKL[28] and oneMKL, IBM ESSL[27], ARM Performance Libraries[7], NVIDIA cuBLAS[30], AMD AOCL[4] for CPUs and rocBLAS[5] for GPUs. The implementations from academia and open-source communities also exist and include BLIS[44] and OpenBLAS[34], both of which build on the success story of portable performance of GotoBLAS[23].

LAPACK[6] was designed to utilize Level 3 BLAS routines by introducing blocked algorithms to bring out the performance from hardware platforms based on then modern architecture of deep memory hierarchies. LAPACK provides routines for all the major numerical linear algebra problems, ranging from solving systems of linear equations, least-squares solutions of linear systems, eigenvalue problems, and singular value problems. Over the years, the library kept expanding and became the standard reference for dense numerical linear algebra applications as it includes the implementations of all the major algorithms in the field.

Several software libraries were subsequently developed that aimed to provide similar functionality as LAPACK while targeting different kinds of hardware platforms and environments. ScaLAPACK[9] was designed to scale on distributed-memory machines by partitioning the matrices into blocks and cyclically distributing the data across the nodes. Its algorithms were implemented to iterate over these blocks to achieve parallelism. As the multicore CPUs were emerging, PLASMA[2] took a similar idea of breaking the matrix down, but instead used smaller submatrices called tiles that better exploit the hardware structure of these shared-memory multicore systems. A task-based scheduler was introduced to remove the synchronization points in the algorithms and replace them with runtime scheduling of small tasks which operate on the tiles and are tracked based on their data dependences. MAGMA[43] was designed for heterogeneous architecture settings by exploiting hybrid hardware environment. These systems were equipped with hardware accelerators, usually GPUs, along with multicore CPUs. As the GPU brought a lot of computational power in terms of floating-point operations, the communications between the CPU and GPU remained a bottleneck, as the bandwidth between the two continues to be much more limited in comparison to internal memory structure of either a CPU or GPU. Thus the implementations in MAGMA were redesigned to distribute different tasks to the CPU and GPU to optimally fit their strengths and at the same time overlap the CPU-GPU communication with computations as much as possible. Software for Linear Algebra Targeting Exascale (SLATE)[22] aims to replace the venerable ScaLAPACK library. As the latest supercomputer installations are commonly accelerated by multiple GPUs on every distributed node, it would be hard to modify ScaLAPACK to take advantage of such machines. SLATE is designed with this modern HPC hardware in mind and features support for multiple computational backends. SLATE also embraces the open standards like MPI and OpenMP to promote portability while retaining performance and parallel efficiency.

**3.4. The SICE Algorithm.** In this section, we review the SICE algorithm by Dongarra el al. [18, 19, 20]. Given the base eigenpair $\lambda, x$ and its nearby eigenpair $\lambda + \mu$, $x + \tilde{y}$, then based on the original eigenproblem we have:

$$A(x+\tilde{y}) = (\lambda+\mu)(x+\tilde{y}) \tag{3.1}$$

Assuming that $x$ is normalized in infinite norm: $|x|_\infty = 1 \equiv x_s$, we can remove one degree of freedom by requiring $\tilde{y}_s = 0$. Rearranging Eq. (3.1) we get:

$$(A-\lambda I)\tilde{y} - \mu x = \lambda x - Ax - \mu\tilde{y} \tag{3.2}$$

The last term is the second order term for the error in $\lambda$ and $x$. By simplify the equation, we introduce vector $y$, defined as:

$$y^\intercal \stackrel{\text{def}}{=} (\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_{s-1}, \mu, \tilde{y}_{s+1}, \ldots, \tilde{y}_{n-1}, \tilde{y}_n) \tag{3.3}$$

So $y$ would encode information from both $\tilde{y}$ and $\mu$ and thus Eq. (3.2) becomes:

$$By = r + y_s\tilde{y} = r + \mu\tilde{y} \tag{3.4}$$

where $r = \lambda x - Ax$ is the residual vector of $\lambda$ and $x$ and $B$ is the matrix $A - \lambda I$ with column $s$ replaced by $-x$.

We can also view it as the Newton's method. In particular, by setting $v = \binom{x}{\lambda}$ we can be formulate the eigenvalue problem as:

$$f(v) \equiv \begin{pmatrix} Ax - \lambda x \\ e_s^\intercal x - 1 \end{pmatrix} = 0 \tag{3.5}$$

where $e_s$ is the $s$-th column of the identity matrix of size $n$. The Newton's method then solves the linear system of the Jacobian matrix:

$$J\begin{pmatrix} \tilde{y} \\ \mu \end{pmatrix} = \begin{pmatrix} A - \lambda I & -x \\ e_s^\intercal & 0 \end{pmatrix}\begin{pmatrix} \tilde{y} \\ \mu \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix} = f(v) \tag{3.6}$$

Expanding it, we arrive at Eq. (3.2) without the second-order term:

$$(A - \lambda I)\tilde{y} - \mu x = r \tag{3.7}$$

This is the basic idea of the SICE algorithm: by iteratively solving Eq. (3.4) we obtain both the correction to the eigenvalue and to the eigenvector. The original algorithm uses Schur decomposition and applies two steps of Givens rotation in order to solve Eq. (3.4). For any real matrix $A$, there exists an orthogonal matrix $Q$ and an upper quasi-triangular matrix $T$, such that

$$A = QUQ^{\intercal} \tag{3.8}$$

where $U$ is upper quasi-triangular with some $2\times 2$ diagonal blocks arising from complex conjugate eignevalue pairs. Here, we define $Z_\lambda \equiv Z - \lambda I$ and $z_{\lambda s} \equiv Z_\lambda e_s = (Z - \lambda I)e_s$. By rewriting Eq. (3.4), we get:

$$[A_\lambda - (x + a_{\lambda s})e_s^{\intercal}]y = (A + ce_s^{\intercal})y = r + y_s\tilde{y} \tag{3.9}$$

where $c = -x - a_{\lambda s}$. Using the Schur decomposition $A = QUQ^{\intercal}$, we have:

$$Q(U_\lambda + Q^{\intercal}ce_s^{\intercal}Q)Q^{\intercal}y = r + y_s\tilde{y} \tag{3.10}$$

$$(U_\lambda + df^{\intercal})Q^{\intercal}y = Q^{\intercal}g \tag{3.11}$$

where $d = Q^{\intercal}c$, $f^{\intercal} = e_s^{\intercal}Q$ and $g = r + y_s\tilde{y}$. Matrix $d \times f^{\intercal}$ constitutes a rank-1 update. Then two steps of Givens rotation are introduced: the first one $Q_1$ is constructed so that

$$Q_1 d = (P_2 P_3 \dots P_n)d = \gamma e_1 \quad \text{where } \gamma = \|d\|_2 \tag{3.12}$$

and $P_i$ is the rotation in $(i-1, i)$ plane that eliminates the $i$-th component in $P_{i+1} \dots P_n d$. We also have:

$$Q_1(U_\lambda + df^{\intercal}) = Q_1 U_\lambda + \gamma e_1 f^{\intercal} \tag{3.13}$$

The transformation $Q_1$ introduces one more nonzero element in the subdiagonal direction of $U_\lambda$. The new rank-one update $\gamma e_1 \times f^{\intercal}$ has nonzero elements only in the first row, which preserves the original structure. The second step of Givens rotation $Q_2$ can be applied subsequently in order to obtain the upper triangular form $\bar{U}_\lambda = Q_2 Q_1 (U_\lambda + d \times f^{\intercal})$ in

$$\bar{U_\lambda} Q^{\intercal} y = Q_2 Q_1 Q^{\intercal} g \tag{3.14}$$

The triangular solve requires $O(n^2)$ operations while the remaining steps of the iteration are only $O(n)$. This procedure is shown in Algorithm 3.1.

**4. Algorithm and Implementation.** The original SICE algorithm is designed for a general real matrices and here we first focus on symmetric ones. The proposed algorithm utilizes the tridiagonalization as well as the Sherman–Morrison formula to solve the linear system for eigenvalue and eigenvector corrections. The blocked version will also be discussed with the implementation details based on PLASMA and MAGMA software libraries.

**Algorithm 3.1** SICE algorithm

**Input:** Matrix $A \in \mathbb{R}^{n\times n}$. An approximate eigenvalue $\lambda$ and the corresponding eigenvector $x$. $\text{iter}_{\max}$ denotes the maximum number of iterations.
**Output:** Refined eigenvalue $\lambda$ and its eigenvector $x$.
**function** $[\lambda, x] \leftarrow \textbf{SICE}(A, \lambda, x, iter)$
  $[Q, U] \leftarrow \text{schur}(A)$ ▷ obtain Schur decomposition $A = QUQ^\intercal$, $QQ^\intercal = I$.
  $[m, s] \leftarrow max(abs(x)); x \leftarrow x/m$ ▷ Normalizing $x$ so that $\|x\|_\infty = s_x = 1$.
  **for** $i$ in $1 : \text{iter}_{\max}$ **do**
    $r \leftarrow \lambda x - Ax$
    $c \leftarrow -x - a_{\lambda s}$
    $d \leftarrow Q^\intercal c$
    $f^\intercal \leftarrow Q(s, :) = e_s^\intercal Q$ ▷ $s$-th row of $Q$.
    $\bar{U}_\lambda \leftarrow Q_1(U - \lambda I); \bar{d} \leftarrow Q_1 d = \|d\|_2 e_1$ ▷ Givens rotations $Q_1$ from Eq. (3.12)
    $\bar{U}_\lambda \leftarrow \bar{U}_\lambda + \bar{d}(1) f^\intercal$
    $\bar{U}_\lambda \leftarrow Q_2 \bar{U}_\lambda$ ▷ Givens rotations $Q_2$ to introduce upper triangular form.
    Solve the triangular system $\bar{U}_\lambda z = Q_2 Q_1 Q^\intercal r$
    $y \leftarrow Qy$
    $\lambda \leftarrow \lambda + y(s)$ ▷ Update eigenvalue.
    $y(s) \leftarrow 0$ ▷ Set $y(s)$ to 0.
    $x \leftarrow x + y$ ▷ Update eigenvector.
    **if** desired accuracy is reached **then**
      **break**
    **end if**
  **end for**
**end function**

**4.1. SICE-SM Algorithm.** For symmetric eigenvalue problems, the matrix $A$ is first reduced to tridiagonal through unitary similarity transformations: $T = Q^\intercal AQ$ where $QQ^\intercal = I$ and $T$ is a symmetric tridiagonal matrix. This corresponds to LAPACK routines `SSYTRD` and `DSYTRD` for single- and double-precision arithmetic, respectively. In the same fashion as SICE algorithm in Section 3.4, we start with Eq. (3.9) and apply the tridiagonal reduction to it. Eqs. (3.10) and (3.11) in this case become

$$Q\left(T_\lambda + Q^\intercal c e_s^\intercal Q\right) Q^\intercal y = r + y_s \tilde{y} \tag{4.1}$$

and

$$\left(T_\lambda + d \times f^\intercal\right) Q^\intercal y = Q^\intercal g \tag{4.2}$$

the same with $d = Q^\intercal c$, $f^\intercal = e_s^\intercal Q$ and $g = r + y_s \tilde{y}$. Dongarra[18] discussed the approach of using the Sherman–Morrison formula[38]

$$\left(A - uv^\intercal\right)^{-1} = A^{-1} - \frac{A^{-1} uv^\intercal A^{-1}}{1 + v^\intercal A^{-1} u} \tag{4.3}$$

for solving the rank-one updated system. Eq. (4.2) does not apply since $T_\lambda = T - \lambda I$ is singular by construction. However, this may not be so in mixed-precision setting. Consider the scheme that first performs the tridiagonal reduction in single precision and then solves the tridiagonal eigenvalue problem in double precision. The initial $\lambda_T$ will be the eigenvalue of $T$ with double-precision accuracy, but it only approximates $\lambda_A$, the eigenvalue of $A$ with single-precision accuracy. With suitably chosen offset $\delta$

**Algorithm 4.1** SICE-SM algorithm: SICE algorithm with Sherman–Morrison formula

1: **Input:** Matrix $A = A^\intercal \in \mathbb{R}^{n\times n}$. An approximate eigenvalue $\lambda$ and the corresponding eigenvector $x$. $\text{iter}_{\text{max}}$ denotes the maximum number of iterations.
2: **Output:** Refined eigenvalue $\lambda$ and eigenvector $x$.
3: **function** $[\lambda, x] \leftarrow$ **SICE_SM**$(A, \lambda, x, iter)$
4: $\quad [Q, T] \leftarrow \text{tridiag}(A)$ ▷ Tridiagonalization $A = QTQ^\intercal$, $QQ^\intercal = I$.
5: $\quad [m, s] \leftarrow max(abs(x)); x \leftarrow x/m$ ▷ Normalization of $x$ so that $\|x\|_\infty = s_x = 1$.
6: $\quad$ **for** $i$ in $1 : \text{iter}_{\text{max}}$ **do**
7: $\qquad r \leftarrow \lambda x - Ax$
8: $\qquad c \leftarrow -x - a_{\lambda s}$
9: $\qquad d \leftarrow Q^\intercal c$
10: $\qquad f^\intercal \leftarrow Q(s, :) = e_s^\intercal Q$ ▷ $s$-th row of $Q$.
11: $\qquad rhs \leftarrow Q^\intercal r$
12: $\qquad u \leftarrow (T - \lambda I)^{-1} d$
13: $\qquad v \leftarrow (T - \lambda I)^{-1} rhs$
14: $\qquad y \leftarrow v - \frac{f^\intercal v}{1+f^\intercal u} u$ ▷ Sherman–Morrison formula
15: $\qquad y \leftarrow Qy$
16: $\qquad \lambda \leftarrow \lambda + y(s)$ ▷ Update eigenvalue.
17: $\qquad$ **if** $i \neq 1$ **then**
18: $\qquad\quad y(s) \leftarrow 0$ ▷ Set $y(s)$ to 0.
19: $\qquad\quad x \leftarrow x + y$ ▷ Update eigenvector.
20: $\qquad$ **end if**
21: $\qquad$ **if** desired accuracy reached **then**
22: $\qquad\quad$ **break**
23: $\qquad$ **end if**
24: $\quad$ **end for**
25: **end function**

of order of $\epsilon_{\text{single}}$, $T - (\lambda + \delta)I$ will no longer be singular in double precision, and the Sherman–Morrison formula can be applied. The special case in which this would fail is when $\|\lambda_T - \lambda_A\| = O(\epsilon_{\text{double}})$: the initial eigenvalue is also an accurate eigenvalue of $A$ in double precision. In such a case, we do not need to refine the eigenvalue and can simply apply the inverse iteration to find the eigenvector.

Applying Sherman–Morrison formula from Eq. (4.3) to Eq. (4.2) we get

$$Q^\intercal y = \left( T_\lambda^{-1} - \frac{T_\lambda^{-1} d \times f^\intercal T_\lambda^{-1}}{1 + f^\intercal T_\lambda^{-1} d} \right) Q^\intercal g \tag{4.4}$$

or

$$Q^\intercal y = T_\lambda^{-1} Q^\intercal g - \frac{f^\intercal (T_\lambda^{-1} Q^\intercal g)}{1 + f^\intercal (T_\lambda^{-1} d)} T_\lambda^{-1} d \tag{4.5}$$

These involve solving the tridiagonal system $T_\lambda$ with two different right hand sides $d$ and $Q^\intercal g$. It can be easily done with the Thomas algorithm which is a special case of Gaussian elimination. There are other parallel tridiagonal solvers available and we will discuss them in Section 4.3.1. We outline the SICE algorithm with Sherman–Morrison formula in Algorithm 4.1.

The main difference between Algorithms 3.1 and 4.1 is the use of the Sherman–Morrison formula to solve the system from line 12 to 14 instead of using the

Table 1

*Performance of $n \times n$ matrix times $n \times m$ aggregated vectors on NVIDIA V100-SXM2-32GB GPU,* `DGEMM` *routine from cuBLAS v11.0.*

| Matrix size | Number of vectors | Time (ms) | Performance (GFLOP/s) |
|---|---|---|---|
| 20000 | 1 | 3.76 | 212.65 |
| 20000 | 8 | 3.79 | 1688.17 |
| 20000 | 32 | 6.48 | 3949.32 |
| 20000 | 128 | 13.57 | 7544.43 |

Givens rotations for that purpose. It is applied to solving the same tridiagonal system $T_\lambda$ with two different right hand sides $d$ and $Q^\intercal g$. The two vector inner products are needed to obtain the scalar in order to form the solution. Note that in line 17, we only update the eigenvalue at the first iteration and leave the eigenvector unchanged because $T_\lambda$ at the first iteration is nearly singular. Other approaches to this issue include manually applying a shift to the initial eigenvalue or using the Ritz value $\frac{x^\intercal Ax}{x^\intercal x}$ as the starting point. Apart from tridiagonalization, the computational cost for algorithm 4.1 is dominated by the matrix-vector multiplications which require $O(n^2)$ operations. The remaining steps of the algorithm are all order $O(n)$ including the tridiagonal solve.

Alternatively, as described in [37], one can also solve the Jacobian matrix with the special structure $J = \begin{pmatrix} T-\lambda I & y \\ z^\intercal & 0 \end{pmatrix}$, which is a tridiagonal system with an extra row and column at the bottom and right. However, it is hard to parallelize the corresponding solver for this special structure and it is even harder make it scalable. This is in stark contrast with the approach of solving the tridiagonal system which is well studied and admits several parallel implementations that target a variety of computing environments.

**4.2. Blocked SICE-SM Algorithm.** The computational cost of Algorithm 4.1 is dominated by matrix-vector multiplications especially inside the refinement iteration. In the matrix-vector multiplication, the whole matrix is read once and only a single multiplication and addition are performed per each of the fetched elements. This results in a low arithmetic intensity of 2, which results in very low inefficient on modern hardware including CPU, GPUs, and computational accelerators. To improve on this implementation aspect, we can aggregate several eigenpairs simultaneously and refine them at the same time while they are cached in higher levels of the memory hierarchy. This blocking strategy is common in numerical linear algebra since it was introduced in LAPACK[6] and relies on grouping computations so that Level 3 BLAS may be utilized to perform operations that are rich in matrix-matrix multilications. These operations perform more efficiently as they have higher arithmetic intensity resulting from higher data reuse in fast portions of the cache hierarchy. In our case, we assume that the matrix size is far greater than the number of eigenpairs to refine. Then the matrix-vector multiplication is dominated by the reading of the matrix elements. And with the blocked version, it the additional cost of refining extra eigenpairs is negligible. In Table 1, we show examples of the performance rates and execution times for different numbers of vectors submitted to the `DGEMM` routine from cuBLAS on the NVIDIA V100 GPU. The times for 1 and 8 vectors are almost the same. And for 32 or 128 vectors the elapsed time increases 3.6×.

There are a few issues we need to solve while formulating a blocked variant of the algorithm. First, in SICE, the eigenvector is first normalized in infinity norm. The index $s$ is also picked so that $\|x\|_\infty = s_x = 1$. If we allow different $s$ for each of the eigenpairs, then we will have to access different columns in $A$ to construct vector $c$, and also different rows of $Q$ for vector $f^\intercal$. The row access required for the latter is performed in column major layout and results in non-coalescing memory accesses which are extremely slow and should be avoided as much as possible due to their low utilization of the GPU's memory bandwidth. To show that it is fine to choose $s$ arbitrarily, we need to take a closer look at the matrix in Eq. (4.1) and expand it without canceling any terms we get

$$(QT_\lambda Q^\intercal + QQ^\intercal v e_s^\intercal QQ^\intercal)y = r + y_s\tilde{y} \tag{4.6}$$

Again, for our mixed-precision scheme, we would like to perform the tridiagonalization in single precision. Hence $QT_\lambda Q^\intercal$ is only an approximation of $A$ with precision $\epsilon_{\text{single}}$, i.e. $\|A_\lambda - QT_\lambda Q^\intercal\| \sim O(\epsilon_{\text{single}})$. The same applies to $QQ^\intercal$ which is only an approximation of $I$ with $\|QQ^\intercal - I\| \sim O(\epsilon_{\text{single}})$. So no matter which index $s$ we pick, we will always get an error of order $\epsilon_{\text{single}}$ in the correction of eigenvalue $y_s$ coming from the other elements in the solution vector $y$. There could be a potential problem if the eigenvalue itself is small and the error is preventing the eigenvalue to be refined to desire accuracy. This can be remedied by pre-scaling the matrix so that the eigenvalues are not too small.

The other issue is that by treating the eigenpairs independently they might lose their orthogonality. In the worst case, they might all converge to the same eigenpair. However, it is easy to reorthogonalize with

$$X' = X + \frac{1}{2}X(I - X^\intercal X) \tag{4.7}$$

In practice, we found that it is sufficient to reorthogonalize after the refinement is done. Doing so in each iteration would not speed up the convergence. The computation of $I - X^\intercal X$ also lets us detect if they converged to the same eigenvector. By combining these considerations, we arrive at Algorithm 4.2.

Because a Hermitian matrix can also be tridiagonalized into real matrix, algorithm 4.2 can easily be extended to be applied on Hermitian matrices. The transformation matrix $Q$ now becomes complex, as well as the intermediate vectors. However, the coefficients in $T - \lambda I$ are all real so it can be optimized to avoid doing all the operations in complex space.

**4.3. Implementation Details.** In this section, we will describe some of the details of our implementation. We implemented the Blocked SICE-SM (Algorithm 4.2) in two software packages: PLASMA[2] and MAGMA[43].

PLASMA is a dense linear algebra software package targeting multi-core shared-memory environments with OpenMP directives. It divides the work into small submatrices called tiles in order to exploit the parallelism and dynamically schedule tasks based on data interdependence. PLASMA used to have a runtime scheduler called QUARK but it is now based on OpenMP tasking directives to embrace the open and portable standard for runtime scheduling of computational Direct Acyclic Graphs (DAGs). OpenMP 4 added the `depend` clause for task dependencies and is able to resolve the task DAGs from PLASMA algorithms. PLASMA has two-stage eigensolver implemented in one of its development branches.

MAGMA is also a linear algebra software package but it targets heterogeneous hardware accelerated with GPUs. Due to the characteristically high floating-point

**Algorithm 4.2** Blocked SICE-SM algorithm

**Input:** $A = A^T \in \mathbb{R}^{n\times n}$, initial eigenvectors $X = [x_1|x_2|...|x_\ell] \in \mathbb{R}^{n\times \ell}$ and the corresponding initial eigenvalues $\Lambda = (\lambda_1, \lambda_2, \dots, \lambda_\ell)^T \in \mathbb{R}^l$. $\text{iter}_{\max}$ denotes the maximum number of iterations.
**Output:** Refined eigenvectors $X$ and refined eigenvalues $\Lambda$.
**function** $[X, \Lambda] \leftarrow$ **SICE_SM_BLK**$(A, X, \Lambda, iter)$
    $[Q, T] \leftarrow \text{tridiag}(A)$ ▷ Tridiagonalization $A = QTQ^\intercal$, $QQ^\intercal = I$.
    **for** $i$ in $1 : \text{iter}_{\max}$ **do**
        $s \leftarrow i$
        $R \leftarrow X \times \text{diag\_matrix}(\Lambda) - A \times X$ ▷ Residual vectors need higher precision.
        **for** $j$ in $1 : \ell$ **do**
            $c_j \leftarrow -x_j - A(:, s)$
        **end for**
        Compose matrix $C = [c_1|c_2|...|c_\ell]$ from column vectors $c_j$
        $C(s, :) \leftarrow C(s, :) + \Lambda^T$
        $D = [d_1|d_2|...|d_\ell] \leftarrow Q^T \times C$ ▷ Can be in lower precision.
        $RHS = [rhs_1|rhs_2|...|rhs_\ell] \leftarrow Q^T \times R$ ▷ Can be in lower precision.
        $f \leftarrow Q(s, :)$ ▷ s-th row of Q.
        **for** $j$ in $1 : \ell$ **do**
            $u_i \leftarrow (T - \lambda I)^{-1} d_i$
            $v_i \leftarrow (T - \lambda I)^{-1} rhs_i$
            $y_i \leftarrow v_i - \frac{f^\intercal v_i}{1 + f^\intercal u_i} u_i$ ▷ Sherman–Morrison
        **end for**
        Compose matrix $Y = [y_1|y_2|...|y_\ell]$ from correction vectors $y_j$
        $Y \leftarrow Q \times Y$
        $\Lambda \leftarrow \Lambda + Y(s, :)^T$ ▷ Update eigenvalues.
        **if** $i \neq 1$ **then**
            $Y(s, :) \leftarrow 0$ ▷ Set $y_i(s)$ to 0.
            $X \leftarrow X + Y$ ▷ Update eigenvectors.
            Normalize eigenvectors $x_i$ in $X$.
        **end if**
        **if** desired accuracy reached **then**
            **break**
        **end if**
    **end for**
    $X \leftarrow X + \frac{1}{2} X (I - X^\intercal X)$ ▷ Orthogonalization.
**end function**

performance of GPUs and the limited bandwidth between the CPUs and GPUs, MAGMA algorithms need to be redesigned and refactored to split up the work between CPU and GPU and to overlap communication and computation. MAGMA includes both one- and two-stage eigensolvers. And we used them as building blocks for implementing Algorithm 4.2 for both solvers.

The one-stage eigensolver has the following components with its corresponding LAPACK routine names:

**Algorithm 4.3** One stage symmetric eigensolver

1: DSYTRD: Tridiagonalization via Householder transformations.
2: DSTEDC: Tridiagonal symmetric eigensolver (divide and conquer).
3: DORMTR: back transformation for eigenvectors.

First the system is transformed to the tridiagonal form via Householder transformations. Then the tridiagonal eigensolver is called. We will not discuss the details of eigensolvers here, as it is not the focus of this work. After the eigenvalues and eigenvectors of the tridiagonal system are computed, the back transformation is applied, which is the inverse of the Householder transformations from tridiagonalization stage. Because the transformation is orthogonal, the inverse is simply a transpose. If only a portion of the eigenvectors are requested, the transform would not be explicitly formed for performance reasons. The transform in the form of elementary reflectors is directly applied on eigenvectors of the tridiagonal system to obtain the eigenvectors for the original matrix.

For the mixed-precision eigensolver, we first perform tridiagonalization in single precision as it is computationally intensive requiring $O(n^3)$ operations. After the system is transformed to tridiagonal form, the eigensolver is applied. The eigensolver operates in double precision as we need to be able to distinguish nearby eigenvalues that are closer than $\epsilon_{single}$ but not closer than $\epsilon_{double}$. If single precision is used for this case, the eigenvalues are very likely to be considered as repeated, and the returned eigenvectors could be an arbitrary orthogonal basis of the eigenspace. For the back transformation, the matrix $Q$ needs to be explicitly formed in order for us to solve Eq. (4.2). Then the Blocked SICE-SM (Algorithm 4.2) is used to iteratively refine the eigenpairs to the desired accuracy. Most of the operations in the refinement process are matrix-matrix operations, which have been developed internally. The batched tridiagonal solver in line 16 will be discussed in section 4.3.1.

**Algorithm 4.4** Mixed precision one stage symmetric eigensolver with iterative refinement

SSYTRD: Tridiagonalization via Householder transformations in single precision.
DSTEDC: Tridiagonal symmetric eigensolver (divide and conquer) in double precision.
SORGTR: Generate the transformation matrix $Q$ from elementary reflectors in single precision.
Blocked SICE-SM (algorithm 4.2) for iterative refinement.

For two-stage algorithms, the structure is similar to the one-stage method but both the forward- and back-transformations are split into two staps:

**Algorithm 4.5** Two stages symmetric eigensolver

First stage symmetric to band via Householder transformations.
Second stage band to tridiagonal via bulge chasing.
Tridiagonal symmetric eigensolver (divide and conquer).
back transformation for second stage on eigenvectors.
back transformation for first stage on eigenvectors.

In MAGMA, the first stage is similar to QR factorization with the panel performed completely on the CPU and the update of the trailing matrix performed on the GPU. The second stage bulge chasing is implemented only for the CPU as the multicore architecture with larger cache is a more suitable compared to the GPU. The divide-and-conquer eigensolver is also mainly performed on the CPU except for the final step of merging with large blocks. Both back transformations are applied on the GPU as they are aggregated into matrix-matrix operations.

**Algorithm 4.6** Mixed precision two stages symmetric eigensolver with iterative refinement

1: First stage symmetric to band via Householder transformations in single precision.
2: Second stage band to tridiagonal via bulge chasing in single precision.
3: Tridiagonal symmetric eigensolver (divide and conquer) in double precision.
4: Generate the transformation matrix $Q$ from first stage in single precision. This can start as soon as 1. finishes.
5: Apply the back transformation for second stage onto $Q$ in single precision. This can start as soon as both 2. and 4. finish.
6: Blocked SICE-SM (algorithm 4.2) for iterative refinement.

Mixed precision for a two-stage eigensolver is actually more problematic performance-wise. The main reason is that accumulation of the back transformations from the second stage of bulge chasing is costly: it has a lot of small transformations and is expensive to apply on a square transform matrix $Q$ compared to the case of only computing the eigenvectors. However, we need to explicitly form $Q$ for the later refinement. Here, we exploit the fact that the back transformation is not applied on the eigenvectors; it can actually start as soon as the first stage is finished. So we are reversing the order of back transformations to start it first. Similarly, the back transformation of the second stage can start when both the second stage and the back transformation of the first stage are completed. This is shown in Algorithm 4.6. For the case of MAGMA implementation, this would enable more parallelism. The back transformation of the first stage can be done on the GPU while the second stage of bulge chasing is done on the CPU. The eigensolver, which is mainly done on the CPU, can be overlapped with the back-transformation of the second stage on the GPU.

**4.3.1. Batched Tridiagonal Solver.** Line 16 in Algorithm 4.2 iterates over all the eigenvalues and solves the shifted tridiagonal system for each of them. This kind of computational pattern is suitable for batched interface. The term “batched” comes from the Batched BLAS[17, 1] that defines the interface for performing identical operation on multiple matrices independently and simultaneously. In our case, all the systems are also independent and we can solve them in a batched fashion. On multicore CPUs, the straightforward and efficient approach is to assign one system to each thread at a time which is likely bound to a single CPU core. Each thread can use the Thomas algorithm, which is a special case of Gaussian elimination. But on the GPU, we need more parallelism to saturate the computational potential of the hardware. There are previous studies[48, 15, 12] that investigated the solving of one big tridiagonal system on GPUs. One of the techniques is based on the cyclic reduction (CR). Consider a tridiagonal system with 8 unknowns:

$$\begin{bmatrix} b_1 & c_1 & & & & & & \\ a_2 & b_2 & c_2 & & & & & \\ & a_3 & b_3 & c_3 & & & & \\ & & a_4 & b_4 & c_4 & & & \\ & & & a_5 & b_5 & c_5 & & \\ & & & & a_6 & b_6 & c_6 & \\ & & & & & a_7 & b_7 & c_7 \\ & & & & & & a_8 & b_8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} \tag{4.8}$$

By combing all the even-indexed equations with odd-indexed equation, we are able to have an updated system with half of the size:

$$\begin{bmatrix} b'_1 & c'_1 & & \\ a'_3 & b'_3 & c'_3 & \\ & a'_5 & b'_5 & c'_5 \\ & & a'_7 & b'_7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_5 \\ x_7 \end{bmatrix} = \begin{bmatrix} y'_1 \\ y'_3 \\ y'_5 \\ y'_7 \end{bmatrix} \tag{4.9}$$

The coefficients of the updated system can be computed with the following formulas:

$$\begin{gathered} k_1 = \frac{a_i}{b_{i-1}},\, k_2 = \frac{c_i}{b_{i+1}} \\ a'_i = -a_{i-1}k_1,\, b'_i = b_i - c_{i-1}k_1 - a_{i+1}k_2 \\ c'_i = -c_{i+1}k_2,\, y'_i = y_i - y_{i-1}k_1 - y_{i+1}k_2 \end{gathered} \tag{4.10}$$

By recursively reducing the size of the system by half, it is possible to bring the size down to a single unknown with a trivial solution. Then, the back-substitutions follows the same path in reverse order and thus the solution of the full system is obtained. Alternatively, while reducing the size of systems, we can produce two independent systems, one with odd-indexed unknowns and the other with the even-indexed unknowns. Both systems can be solved independently with only its own coefficients. By repeating the process, we will arrive at trivial systems with a single unknown $b''_i x_i = y''_i$ for all of the unknowns $x_i$. The back substitutions wold not be needed for this approach, which is called *parallel cyclic reduction* (PCR). The PCR method exposes more parallelism towards the end but with requires more computation which represents a design trade-off. For our GPU implementation, we used PCR to solve one tridiagonal system by each of the thread blocks.

**5. Numerical Results and Performance Experiments.** The numerical experiments in this section will be divided into two parts. The first one examines the convergence behavior for refining different portions of the eigenvalues and eigenvectors in the spectrum. Then the performance results with PLASMA and MAGMA software libraries are be given with detailed profiling data to highlighted particular performance cases.

**5.1. Numerical Convergence.** The numerical experiments in this section were performed in MATLAB version R2020a with implementations of Algorithm 4.2 (blocked SICE-SM). The expression `A = gallery('randsvd',n,-cond)` was used to generate symmetric test matrices with a prescribed condition number from random eigenvectors and geometrically distributed eigenvalues in range $(1, \frac{1}{\texttt{cond}})$. The input matrix is first converted to `single` precision and subsequently tridiagonalized using `[Q,T] = hess(A)` function in single precision. Then converted back to double precision for finding the eigenvalues and eigenvectors using expression `[V,D] = eig(A)`. The eigenvectors in `D` and column eigenvectors in `QV` will be used as the starting point of our refinement algorithms.

Figure 1 shows the convergence of Algorithm 4.2: the blocked SICE-SM. The input symmetric input matrix had size 100 with geometrically distributed eigenvalues from 1 to $10^{-7}$. The convergence in terms of residual $\|Ax - \lambda x\|_\infty$ of each eigenvalues are plotted in different colors from blue as largest eigenvalue 1 to red as the smallest eigenvalue $10^{-7}$. For the first iteration, we only updated the eigenvalues so there was no initial improvement. For large eigenvalues, the method converges quickly in two iterations. However, for small eigenvalues, that are much closer to each other due to the geometrical distribution and thus we observer the resulting slowdown of convergence.

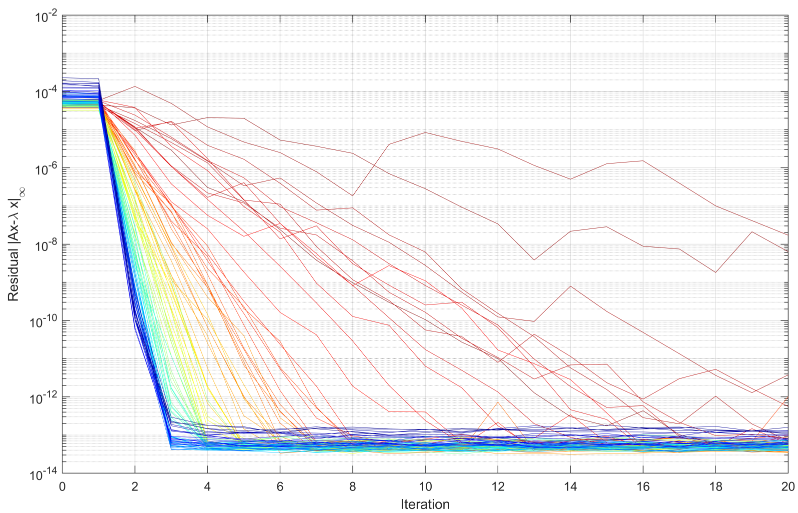


FIG. 1. *Blocked SICE-SM convergence of a* $100 \times 100$ *matrix with geometrically distributed eigenvalues from 1 (blue) to* $10^{-7}$ *(red).*

**5.2. Performance Results.** The system we are using has two sockets of Intel(R) Xeon(R) CPU E5-2650 v3 CPUs. But only one is being used for more stable results. The system is accelerated by a Tesla V100 GPU. The theoretical peak performance of a V100 is 7.8 TFLOP/s in double precision and 15.6 TFLOP/s in single precision. The software stacks was composed of Intel Parallel Studio Cluster 2020. (for C and Fortran compilers and BLAS rouintes from MKL library), NVIDIA CUDA v11.0.2, and MAGMA version 2.5.4. The input symmetric matrix $A \equiv [a_{ij}]$ was generated with random elements from a uniform distribution in range $(0, 1)$: $a_{ij} \sim \mathcal{U}(0, 1)$ and $a_{ij} = a_{ji}$. The Hermitian matrix is also generated in the same fashion for it's imaginary part. The largest eigenvalues in the spectrum were requested. The blocked SICE-SM algorithm was implemented in both PLASMA and MAGMA.

First, we show the profiling results from the PLASMA experiments in left of Figure 2. PLASMA was used in a CPU-only mode and no GPUs were used in the system. The symmetric input matrix had size $n = 10000$. The three stacked bars represent the breakdown of time from mixed-precision with refinement, single precision, and double precision from the two-stage algorithm, respectively. The time for single precision is about half of that of double precision and each of the components take proportionally the same time for both precisions. The mixed-precision algorithm is slower than double precision in this setup because of the requirement of explicitly forming the transformation matrices from the first and second stages. They also take much more time compared to the double precision algorithm, which only applies transformations to the eigenvectors.

Figure 3 shows the performance results from the MAGMA. First the solid lines are the one-stage algorithm in double, single, and mixed precision (with iterative refinement). The input matrix sizes range from 1000 to 20000, and the largest 32 eigenpairs are requested. Single precision is about 1.7× faster than double precision

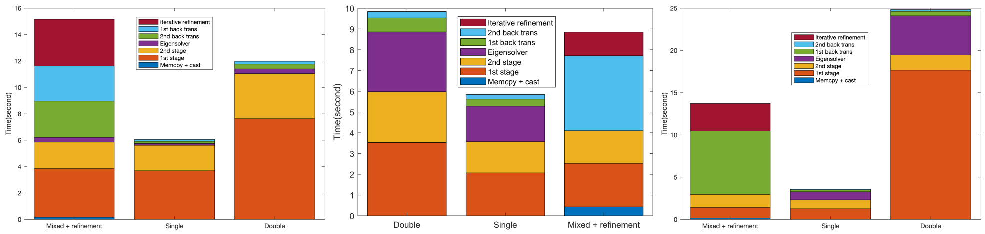


FIG. 2. *Breakdown of timings of two-stage eigensolvers with 32 largest eigenparis requested in PLASMA (left), MAGMA on NVIDIA Volta V100, and MAGMA on NVIDIA GTX1060. The problem sizes are* 10 000*,* 20 000*, and* 12 000*, respectively.*

and the mixed precision is about 1.3× faster. The dashed lines represent the two-stage algorithm. They are at least 2× faster than their corresponding single stage algorithm in general. The performance improvement over double precision is about 1.2×. Figure 4 shows the performance results of complex Hermitian solvers. Complex operations has higher arithmetic intensity so the performance gap between single and double would also be larger. Mixed precision algorithm can also have greater chance to benefit it. On the system wit NVIDIA V100, we are observing complex single is 2.44× faster than complex double and mixed precision solver is 1.45×

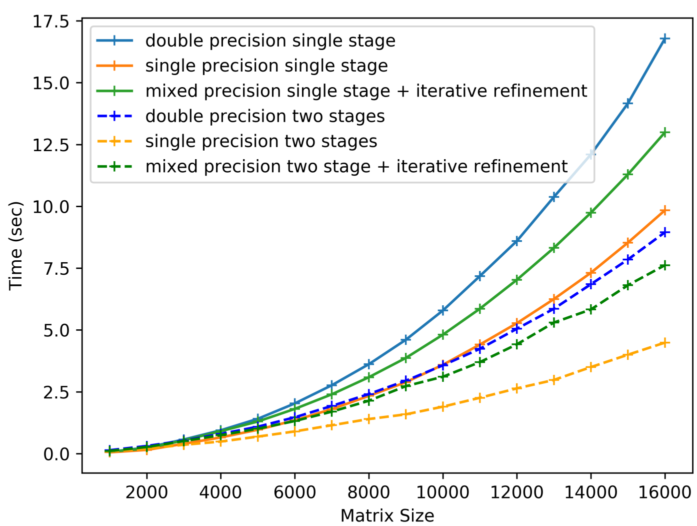


FIG. 3. *Performance comparison of single, double, and mixed precision solvers for real symmetric matrix on MAGMA for both single stage and two-stage algorithms on NVIDIA V100 GPU with varying sizes of matrices and fixed number of requested eigenpairs.*

Figure 5 shows the performance when requesting different numbers of eigenpairs with the input matrix size fixed at $n = 20000$. Mixed precision is noticeably faster than double precision if 64 or fewer eigenpairs are requested. For larger eigenpair count, the time in iterative refinement grows linearly with the number of requested eigenpairs and it eventually looses its performance advantage.

The middle of Figure 2 shows the detailed profile for matrix size $n = 20000$ and

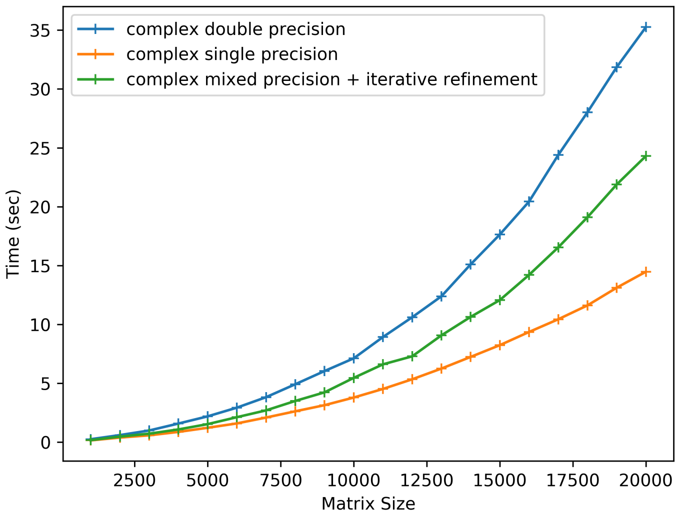


FIG. 4. *Performance of single, double, and mixed precision solvers for complex Hermitian matrix based on MAGMA two-stage algorithm on NVIDIA V100 GPU with varying sizes of matrices and fixed number of requested eigenpairs.*

32 eigenvalues/eigenvectors requested. The details of computational components were explained in Section 4.3. The single precision routine took 60% of time compared to double, and the ratios between components across precisions were about the same. For mixed precision, there is a 0.5 second overhead at the beginning to convert the whole matrix from double to single precision. Then the two-stage reduction is done in single precision which is about twice as fast in single precision. The back-transformation of the first stage is overlapped with the second stage, and is not shown in the bar. The same applies for the eigensolver, which is overlapped with the back-transformation from the second stage. Finally, at the top is the timing for the iterative refinement stage. As can be easily observed, the back transformation of second stage for mixed precision is the bottleneck as it takes almost 40% of the total time in this case.

We tested another machine with a drastically different setup by using a consumer-grade gaming GPU. It has the same CPUs as the V100 system. The GPU is NVIDIA GTX1060 6GB GPU. The theoretical peak performance of GTX1060 is 136.7 GFLOP/s in double and 4.375 TFLOP/s in single precision. This is a notable different as the gaming maintains 1:32 double-single ratio compared to server-grade NVIDIA V100 with the ratio being 1:2. Figure **??** shows the performance with different matrix sizes on GTX1060 when requesting the largest 32 eigenpairs. The performance of single precision is about 8× better than that of double precision and the mixed precision with refinement is about 2× better than double precision. Figure 6 is the complex Hermitian solver and the the speed up over complex double is 3.6× as In Figure 7 we show performance results when the matrix size was fixed at $n = 12000$ but with varied number of requested eigenpairs. The mixed precision solver is still faster than double precision when 128 eigenpairs are requested, but the time in iterative refinement became significant if more eigenvalues and eigenvectors were requested.

The right of Figure 8 shows the profiling results with timing breakdown for ma-

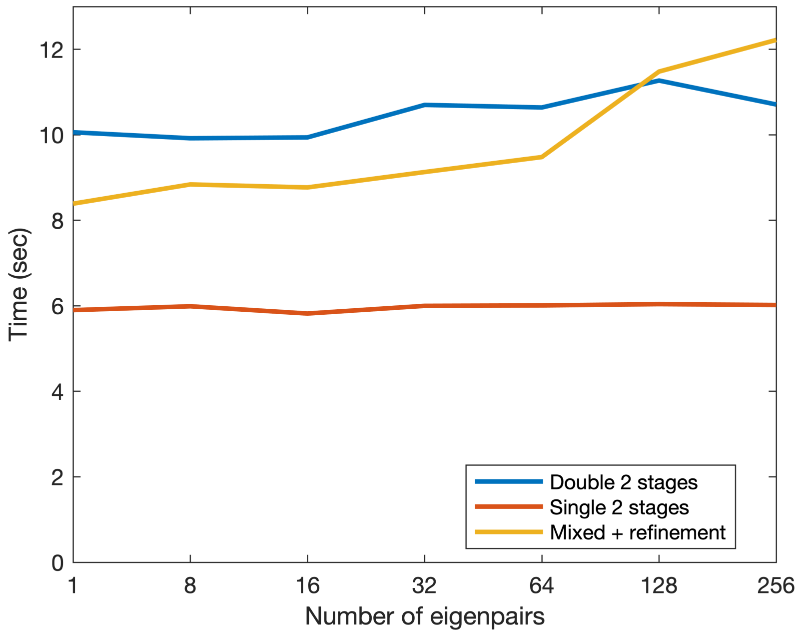


FIG. 5. *Performance comparison of single, double, and mixed precision solvers on top of MAGMA on NVIDIA V100 GPU with varying number of requested eigenpairs and fixed matrix size.*

trix size $n = 12000$ and the 32 largest eigenpairs requested. In double precision, almost 80% of time was spent at the first stage to reduce the matrix from symmetric to band-symmetric form. The operation is compute-bound and relies on GPU's matrix-matrix multiplication efficiency. But the consumer-grade GPU does not have hardware to support high-efficiency processing for the double floating-point units and consequently extra clock cycles are used to emulate higher precision with single precision instructions. The mixed-precision algorithm does the first-stage reduction in single precision and does not suffer from the same penalty. The back-transformation of second stage is still costly but it is done with single precision on the GPU. Overall, the performance of mixed precision with the iterative refinement algorithm is $2\times$ faster over purely double two-stage algorithm.

**6. Conclusions and Future Work.** We developed an iterative refinement algorithm for symmetric and Hermitian eigenvalue problems based on the initial work from the SICE algorithm. By utilizing the Sherman–Morrison formula, our new solver has more opportunity to be parallelized compared to the serial Givens rotations in the SICE algorithm. The blocked version of the algorithm was also proposed in order to refine multiple pairs of eigenvalues and eigenvectors simultaneously for higher utilization of the computational resources with lower demand for memory bandwidth. The implementation of the mixed-precision algorithm is based on the two-stage eigensolver in either the PLASMA and MAGMA software libraries for numerical linear algebra, which gives our implementation the advantage of both portability and performance. The computational components inside the mixed-precision algorithm have been reordered to create more parallelism at runtime and allow additional overlap to computational stages more efficiently. Compared to the double-precision solver, the performance benefit has been shown for the cases in which only a portion of eigenval-

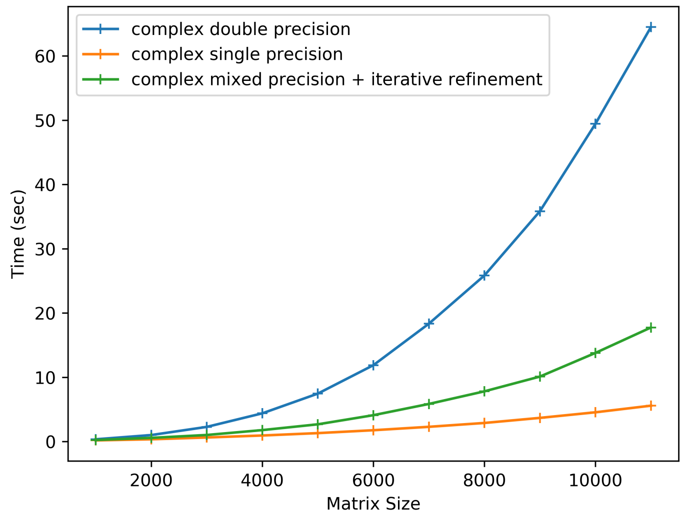


FIG. 6. *Performance of single, double, and mixed precision solvers for complex Hermitian matrix based on MAGMA two-stage algorithm on the NVIDIA GTX1060 GPU.*

ues and corresponding eigenvectors are requested. This remains true across hardware with a varying ratio of performance of single and double precision units.

As we can see in the profiling result featuring time breakdown of the computational tasks, the back-transformation of the second stage that performs bulge chasing is slow on either CPU or GPU and becomes the bottleneck for some experiments. Although the two stage reduction is a far superior method in terms of performance, if only the forward transforms are considered then back-transformations take over the performance and must be taken into account while designing mixed-precision algorithms. One possible approach would be to start aggregating the transformations on the GPU as soon as they are generated by GPU-based bulge chasing and not wait until all the reductions have been computed.

For distributed systems, the matrix is usually too large and it might not be feasible to explicitly form the transform matrix. Consequently, the cost of applying the transformation $Q$ during iterative refinement needs to be reevaluated. Also, if different eigenpairs are being distributed and refined on different nodes, synchronizing and applying $Q$ to eigenvectors across disparate nodes needs to be designed and implemented with care as this is not a usual operation.

Another direction is to try different low-precision formats in addition to just mixing single and double precisions. The recently released NVIDIA Ampere GPU provides `TF32` Tensor Cores, which uses all 8 exponent bits and 10 out of 23 mantissa bits from the `FP32` single precision format, and thus offering $8\times$ speedup. Because our initial eigenpairs and the reduced systems are all coming from the low-precision tridiagonalization, the convergence rate of the iterative refinement is affected significantly. Based on our experiments, the `FP16` half-precision tensor cores do not provide sufficient accuracy and `TF32` might appears to be a more promising target with more balanced mix of precision and performance.

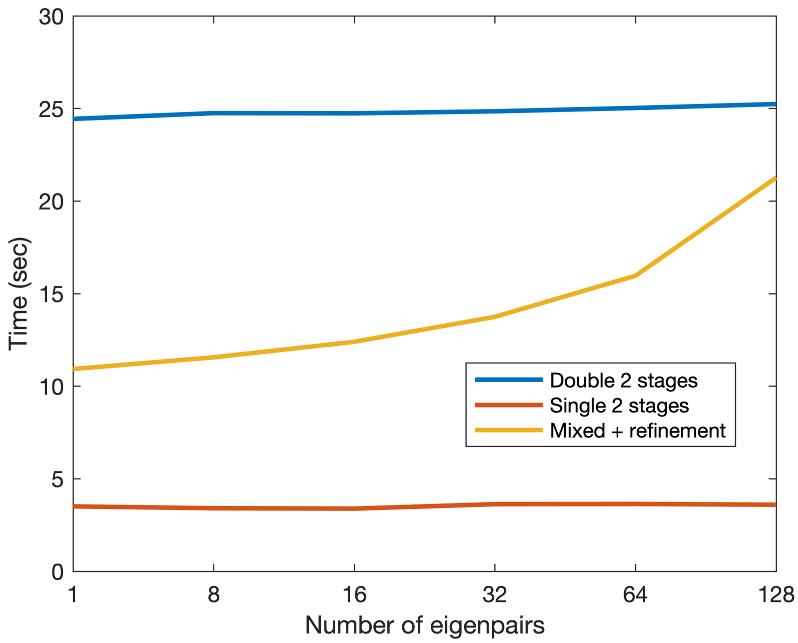


Fig. 7. *Performance comparison of single, double, and mixed precision solvers on top of MAGMA on NVIDIA GTX1060 GPU with varying number of requested eigenpairs and fixed matrix size $n = 12000$.*

## REFERENCES


[1] A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Kurzak, P. Luszczek, S. Tomov, and M. Zounon, *A set of batched basic linear algebra subprograms and LAPACK routines*, ACM TOMS, 47 (2020), p. 1–23, https://doi.org/10.1145/3431921. DOI: 10.1145/3431921.

[2] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, *PLASMA users guide*, tech. report, Technical report, ICL, UTK, 2009.

[3] G. Alefeld and H. Spreuer, *Iterative improvement of componentwise errorbounds for invariant subspaces belonging to a double or nearly double eigenvalue*, Computing, 36 (1986), pp. 321–334.

[4] *AMD optimizing CPU libraries (AOCL)*. https://developer.amd.com/amd-aocl/, 2020. [Online; accessed 2-August-2020].

[5] *rocBLAS: AMD's library for BLAS on ROCm*. https://github.com/ROCmSoftwarePlatform/rocBLAS, 2020. [Online; accessed 2-August-2020].

[6] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., *LAPACK Users' guide*, SIAM, 1999.

[7] *ARM performance libraries*. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux/arm-performance-libraries, 2020. [Online; accessed 2-August-2020].

[8] C. H. Bischof, B. Lang, and X. Sun, *A framework for symmetric band reduction*, ACM Transactions on Mathematical Software (TOMS), 26 (2000), pp. 581–601.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al., *ScaLAPACK users' guide*, SIAM, 1997.

[10] E. Carson and N. J. Higham, *A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems*, SIAM Journal on Scientific Computing, 39 (2017), pp. A2834–A2856.

[11] E. Carson and N. J. Higham, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM Journal on Scientific Computing, 40 (2018), pp. A817–

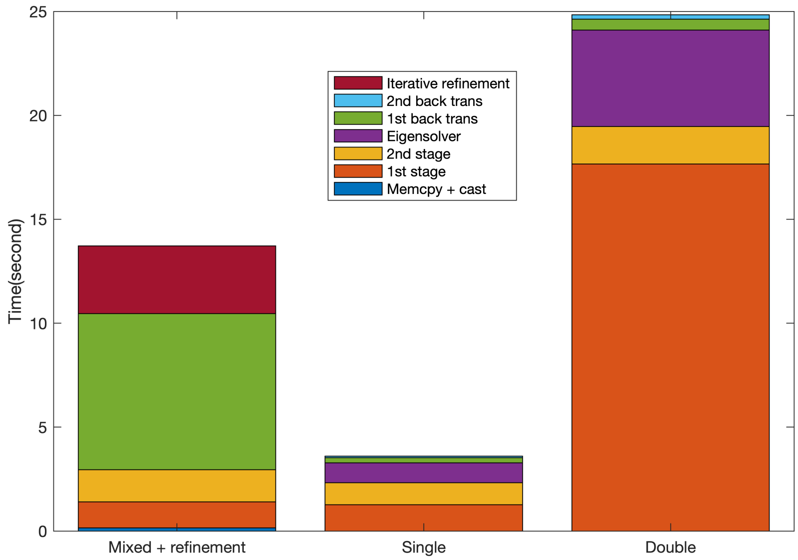


FIG. 8. *Breakdown of timings of two-stage eigensolvers based on MAGMA on the NVIDIA GTX1060 GPU with size* $n = 12000$ *and 32 largest eigenpairs requested.*

A847.

[12] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu, *A scalable, numerically stable, high-performance tridiagonal solver using GPUs*, in SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–11.

[13] F. Chatelin, *Simultaneous Newton's iteration for the eigenproblem*, in Defect correction methods, Springer, 1984, pp. 67–74.

[14] J. J. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numerische Mathematik, 36 (1980), pp. 177–195.

[15] A. Davidson, Y. Zhang, and J. D. Owens, *An auto-tuned method for solving large tridiagonal systems on the GPU*, in 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE, 2011, pp. 956–965.

[16] J. W. Demmel, *Three methods for refining estimates of invariant subspaces*, Computing, 38 (1987), pp. 43–57.

[17] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, *The design and performance of batched BLAS on modern high-performance computing systems*, Procedia Computer Science, 108 (2017), pp. 495–504.

[18] J. J. Dongarra, *Improving the accuracy of computed matrix eigenvalues*, tech. report, Argonne National Lab., Chicago, IL, USA, 1980.

[19] J. J. Dongarra, *Algorithm 589: SICEDR: a FORTRAN subroutine for improving the accuracy of computed matrix eigenvalues*, ACM Transactions on Mathematical Software (TOMS), 8 (1982), pp. 371–375.

[20] J. J. Dongarra, C. B. Moler, and J. H. Wilkinson, *Improving the accuracy of computed eigenvalues and eigenvectors*, SIAM Journal on Numerical Analysis, 20 (1983), pp. 23–45.

[21] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling, *Block reduction of matrices to condensed forms for eigenvalue computations*, Journal of Computational and Applied Mathematics, 27 (1989), pp. 215–227.

[22] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, *SLATE: design of a modern distributed and accelerated linear algebra library*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–18.

[23] K. Goto and R. Van De Geijn, *High-performance implementation of the Level 3 BLAS*, ACM Transactions on Mathematical Software (TOMS), 35 (2008), pp. 1–14.

[24] A. Haidar, H. Ltaief, and J. Dongarra, *Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels*, in

Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–11.

[25] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, *Harnessing GPU tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers*, in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 603–613.

[26] A. Haidar, S. Tomov, J. Dongarra, R. Solcà, and T. Schulthess, *A novel hybrid CPU–GPU generalized eigensolver for electronic structure calculations based on fine-grained memory aware tasks*, The International journal of high performance computing applications, 28 (2014), pp. 196–209.

[27] *IBM engineering and scientific subroutine library (ESSL) version 6.3*. https://www.ibm.com/support/knowledgecenter/SSFHY8_6.3/navigation/welcome.html, 2020. [Online; accessed 2-August-2020].

[28] *Intel Math Kernel Library*. https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html, 2020. [Online; accessed 2-August-2020].

[29] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for fortran usage*, ACM Transactions on Mathematical Software (TOMS), 5 (1979), pp. 308–323.

[30] *NVIDIA cuBLAS*. https://docs.nvidia.com/cuda/cublas/index.html, 2020. [Online; accessed 2-August-2020].

[31] T. Ogita and K. Aishima, *Iterative refinement for symmetric eigenvalue decomposition*, Japan Journal of Industrial and Applied Mathematics, 35 (2018), pp. 1007–1035.

[32] T. Ogita and K. Aishima, *Iterative refinement for symmetric eigenvalue decomposition II: clustered eigenvalues*, Japan Journal of Industrial and Applied Mathematics, 36 (2019), pp. 435–459.

[33] T. Ogita and K. Aishima, *Iterative refinement for singular value decomposition based on matrix multiplication*, Journal of Computational and Applied Mathematics, 369 (2020), p. 112512.

[34] *OpenBLAS*. https://github.com/xianyi/OpenBLAS, 2020. [Online; accessed 2-August-2020].

[35] B. N. Parlett and I. S. Dhillon, *Relatively robust representations of symmetric tridiagonals*, Linear Algebra and its applications, 309 (2000), pp. 121–151.

[36] G. Peters and J. H. Wilkinson, *Inverse iteration, ill-conditioned equations and Newton's method*, SIAM review, 21 (1979), pp. 339–360.

[37] K. E. Prikopa and W. N. Gansterer, *On mixed precision iterative refinement for eigenvalue problems*, Procedia Computer Science, 18 (2013), pp. 2647–2650.

[38] J. Sherman and W. J. Morrison, *Adjustment of an inverse matrix corresponding to a change in one element of a given matrix*, The Annals of Mathematical Statistics, 21 (1950), pp. 124–127.

[39] B. T. Smith, J. M. Boyle, B. Garbow, Y. Ikebe, V. Klema, and C. Moler, *Matrix eigensystem routines-EISPACK guide*, vol. 6, Springer, 2013.

[40] G. W. Stewart, *Error and perturbation bounds for subspaces associated with certain eigenvalue problems*, SIAM review, 15 (1973), pp. 727–764.

[41] H. Symm and J. H. Wilkinson, *Realistic error bounds for a simple eigenvalue and its associated eigenvector*, Numerische Mathematik, 35 (1980), pp. 113–126.

[42] F. Tisseur, *Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems*, SIAM Journal on Matrix Analysis and Applications, 22 (2001), pp. 1038–1057.

[43] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Computing, 36 (2010), pp. 232–240.

[44] F. G. Van Zee and R. A. Van De Geijn, *BLIS: a framework for rapidly instantiating BLAS functionality*, ACM Transactions on Mathematical Software (TOMS), 41 (2015), pp. 1–33.

[45] J. Wilkinson, *Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection*, Numerische Mathematik, 4 (1962), pp. 362–367.

[46] J. H. Wilkinson, *Global convergene of tridiagonal QR algorithm with origin shifts*, Linear Algebra and its Applications, 1 (1968), pp. 409–420.

[47] T. Yamamoto, *Error bounds for computed eigenvalues and eigenvectors*, Numerische Mathematik, 34 (1980), pp. 189–199.

[48] Y. Zhang, J. Cohen, and J. D. Owens, *Fast tridiagonal solvers on the GPU*, ACM Sigplan Notices, 45 (2010), pp. 127–136.