University of Tennessee, Knoxville

# Mixed-Precision Numerical Linear Algebra Algorithms: Integer Arithmetic Based LU Factorization and Iterative Refinement for Hermitian Eigenvalue Problem 

Yaohung Tsai<br>ytsai2@vols.utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_graddiss
Part of the Numerical Analysis and Scientific Computing Commons

## Recommended Citation

Tsai, Yaohung, "Mixed-Precision Numerical Linear Algebra Algorithms: Integer Arithmetic Based LU Factorization and Iterative Refinement for Hermitian Eigenvalue Problem. " PhD diss., University of Tennessee, 2020.
https://trace.tennessee.edu/utk_graddiss/6094

To the Graduate Council:
I am submitting herewith a dissertation written by Yaohung Tsai entitled "Mixed-Precision Numerical Linear Algebra Algorithms: Integer Arithmetic Based LU Factorization and Iterative Refinement for Hermitian Eigenvalue Problem." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor
We have read this dissertation and recommend its acceptance:
Michael W. Berry, James S. Plank, Vasileios Maroulas
Accepted for the Council:
Dixie L. Thompson
Vice Provost and Dean of the Graduate School
(Original signatures are on file with official student records.)

# Mixed-Precision Numerical Linear 

Algebra Algorithms: Integer

# Arithmetic Based LU Factorization 

 and Iterative Refinement for Hermitian Eigenvalue ProblemA Dissertation Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Yaohung Tsai

December 2020

Copyright © by Yaohung Tsai, 2020
All Rights Reserved.

To my wife Yingchia.

## Acknowledgments

It has been quite a journey for the last six years here at University of Tennessee. I still vividly remember the first day I arrived at McGhee Tyson airport, saw the campus for the first time, thinking and being excited about I will be studying here in the following years. Once a Vol, always a Vol. Now Knoxville is really a second home to me. Although I am not leaving right away, I know that I will definitely miss everything.

I would like to thank my advisor, Dr. Jack Dongarra. Innovative Computing Laboratory (ICL) is a tremendous place and it is a privilege to be part of the group. His passion toward high performance computing and numerical linear algebra will always be my pattern. All people at ICL are smart, skilled, and motivated. More than that, it is really a big family where I received so much support in all different ways.

I also would like to thank Dr. Michael Berry, Dr. Vasileios Maroulas, and Dr. James Plank for serving as my committee member. I greatly appreciate all their advices on my dissertation work.

Especially I would like to thank Dr. Jakub Kurzak and Dr. Piotr Luszczek for their guidance during my PhD. They gave me so much freedom to explore and work on things that I am interested in. So many times they have enlightened me when I was stuck and not sure how to move on.

I thank my wife Yingchia for her love and support. I would not be able to finish this without her. I thank my parents for giving me a chance to pursue my dream. Finally, I thank our Lord Jesus Christ who is all and in all.

## Abstract

Mixed-precision algorithms are a class of algorithms that uses low precision in part of the algorithm in order to save time and energy with less accurate computation and communication. These algorithms usually utilize iterative refinement processes to improve the approximate solution obtained from low precision to the accuracy we desire from doing all the computation in high precision. Due to the demand of deep learning applications, there are hardware developments offering different low-precision formats including half precision (FP16), Bfloat16 and integer operations for quantized integers, which uses integers with a shared scalar to represent a set of equally spaced numbers. As new hardware architectures focus on bringing performance in these formats, the mixed-precision algorithms have more potential leverage on them and outmatch traditional fixed-precision algorithms.

This dissertation consists of two articles. In the first article, we adapt one of the most fundamental algorithms in numerical linear algebra-LU factorization with partial pivoting - to use integer arithmetic. With the goal of obtaining a low accuracy factorization as the preconditioner of generalized minimal residual (GMRES) to solve systems of linear equations, the LU factorization is adapted to use two different fixed-point formats for matrices $L$ and $U$. A left-looking variant is also proposed for matrices with unbounded column growth. Finally, GMRES iterative refinement has shown that it can work on matrices with condition numbers up to 10000 with the algorithm that uses int16 as input and int32 accumulator for the update step.

The second article targets symmetric and Hermitian eigenvalue problems. In this section we revisit the SICE algorithm from Dongarra et al. By applying the ShermanMorrison 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 $3.6 \times$ 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.

## Table of Contents

1 Integer Arithmetic-Based LU Factorization ..... 2
1 Introduction ..... 3
2 Literature Review and Background ..... 4
2.1 Iterative Refinement and Mixed-Precision Algorithms for Numerical Linear Algebra ..... 4
2.2 Numerical Representations ..... 5
2.3 LU Factorization with Partial Pivoting ..... 7
2.4 Iterative Refinement with LU Factorization and Preconditioned Gen- eralized Minimal Residual Method (GMRES) ..... 11
3 Algorithm ..... 12
3.1 Fixed-Point Representation ..... 12
3.2 Integer Arithmetic-Based LU Factorization with Partial Pivoting ..... 14
3.3 Left-Looking Integer LU with Dynamic Column Scaling ..... 16
4 Numerical Results ..... 18
4.1 Column Growth ..... 18
4.2 Backward Error and Residual ..... 19
4.3 Iterative Refinement Results ..... 21
4.4 Discussion ..... 24
5 Conclusion and Future Work ..... 27
2 Iterative Refinement Algorithm for Symmetric Eigenvalue Problem on Modern Hardware ..... 29
1 Introduction ..... 30
2 Literature Review and Background ..... 31
2.1 Eigenvalue refinement ..... 31
2.2 Parallel Eigensolvers ..... 32
2.3 Software Packages for Symmetric/Hermitian Eigenvalue Problems ..... 35
2.4 The SICE Algorithm ..... 37
3 Algorithm and Implementation ..... 40
3.1 SICE-SM Algorithm ..... 40
3.2 Blocked SICE-SM Algorithm ..... 43
3.3 Implementation Details ..... 47
4 Numerical Experiments ..... 51
4.1 Numerical Convergence ..... 51
4.2 Performance Results ..... 52
5 Conclusions and Future Work ..... 63
Bibliography ..... 70
Vita ..... 79

## List of Tables

1.1 Summary of numerical representation formats ..... 6
1.2 Bit fields in Q3.12. It has one sign bit, 3 integer bits, and 12 fraction bits. ..... 13
1.3 Backward errors $\frac{\|A x-b\|_{\infty}}{\|A\|_{\infty}\|x\|_{\infty}+\|b\|_{\infty}}$ from different precisions and algorithms versus different input matrices of size 1000 . ..... 20
2.1 Performance of $n \times n$ matrix times $n \times m$ aggregated vectors on NVIDIA V100-SXM2-32GB GPU, DGEMM routine from cuBLAS v11.0. ..... 44

## List of Figures

1.1 Illustration of numerical representation formats. Each rectangle represent one bit. Different floating-number formats are constructed with different number
of exponent bits and mantissa bits (fraction bits).
1.2 Illustration of right-looking and left-looking algorithms. The right-looking algorithm updates the remaining unfactored matrix toward the right after factorizing the column, whereas the left-looking algorithm updates the current column just before the factorization using previous results from the left.
1.3 Column growth with respect to column index of 5 different matrices. Each dot is representing the largest value of the column in factorized matrix $U$. . .20
1.4 Histogram of elements in residual matrix $R=P A-L U$ ..... 22
1.5 The frequency of dynamic scaling happening in the int16 with int32 accumulation with column scaling algorithm. The input matrix is A1 with size 1000. A dot at $(i, j)$ represents scaling (via integer shift) happened at column $j$ while using the result from column $i$ to perform the update.23
1.6 LU-IR convergence of $1000 \times 1000$ matrices with pre-assigned geometrically distributed singular values25
1.7 Comparing LU-IR and GMRES-IR for a matrix of size 1000, arithmetically distributed singular values and $\operatorname{cond}(A)=10^{5}$.
2.1 Illustration of comparing one stage and two stages tridiagonalization algorithm. The one stage at the top uses Householder transformations to reduce the matrix directly into tridiagonal but will touch the whole matrix for each column. The two stages algorithm at the bottom will first reduce the matrix into band by performing QR factorization in a submatrix, which will open touch the whole matrix after each submatrix (block) is factored. Then the second stage bulge-chasing to further reduce it to tridiagonal.
2.2 Blocked SICE-SM convergence of a $100 \times 100$ matrix with geometrically distributed eigenvalues from 1 (blue) to $10^{-7}$ (red).53
2.3 PLASMA execution times and their breakdowns for matrix of size $n=10000$ and with 32 eigenpairs requested
2.4 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.56
2.5 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.57
2.6 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 $n=20000$.
2.7 Performance comparison of complex single, complex double, and complex mixed precision solvers on top of MAGMA on NVIDIA V100 GPU with varying number of requested eigenpairs and fixed matrix size $n=20000$.
2.8 Breakdown of timings of two-stage eigensolvers based on MAGMA on the NVIDIA V100 GPU with size $n=16000$ and 16 largest eigenpairs requested.
2.9 Profiling of two-stage eigensolvers based on MAGMA on the NVIDIA V100 GPU with size $n=20000$ and 32 largest eigenpairs requested.
2.10 Performance of single, double, and mixed precision solvers for real symmetric matrix based on MAGMA two-stage algorithm on the NVIDIA GTX1060 GPU. 64
2.11 Performance of single, double, and mixed precision solvers for complex Hermitian matrix based on MAGMA two-stage algorithm on the NVIDIA GTX1060 GPU. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
2.12 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=10000$66
2.13 Performance comparison of complex single, complex double, and complex mixed precision solvers on top of MAGMA on NVIDIA GTX1060 GPU with varying number of requested eigenpairs and fixed matrix size $n=10000$.
2.14 Breakdown of timings of two-stage eigensolvers based on MAGMA on the NVIDIA GTX1060 GPU with size $n=12000$ and 32 largest eigenpairs requested.68

## Introduction

In numerical linear algebra, it is natural to seek algorithms with higher performance. This enables us to solve more or larger problems under the same time and resource budget. Mixedprecision algorithms are one of the approaches to speed up the computation by using more than one floating-point precision in the algorithm. The IEEE-754 64-bit double precision is usually a safe choice if only one precision is used and the accuracy is sufficient for most of the applications. By switching to the 32 -bit single precision, it could be $2 \times$ faster or more depending on the hardware architectures. We can also only replace the computationally or numerically insensitive part of the algorithm with low precision. This offers a lot of opportunities to improve the performance. However, the accuracy of the solution might be too still low. To overcome this issue, iterative refinement algorithms are being implemented to refine the solution from low- or mixed-precision calculation. They allow us to perform lowprecision operations on matrices to obtain approximate factorization and an initial solution. Then, the iterative refinement process is utilized to improve the solution toward the desired accuracy.

## Chapter 1

## Integer Arithmetic-Based LU Factorization

This article and its research work is done by student Yaohung M. Tsai under the guidance of Professor Jack Dongarra and Assistant Professor Piotr Luszczek. It has not been published in any proceedings at the time the dissertation is submitted to the University of Tennessee, Knoxville.


#### Abstract

In this work, we adapt one of the most fundamental algorithm in numerical linear algebraLU factorization with partial pivoting - to use integer arithmetic. With the goal of obtaining a low accuracy factorization as the preconditioner of the generalized minimal residual (GMRES) method to solve linear systems of equations, the LU factorization is adapted to use two different fixed-point formats for matrices $L$ and $U$. A left-looking variant is also proposed for matrices with unbounded column growth. Finally, we show that the GMRES iterative refinement could work on matrices with condition number up to $10^{4}$ with the algorithm that uses int16 as input and int32 accumulator for the update step.


## 1 Introduction

As hardware development is pushed forward heavily by the need of AI model training and deployment, a lot of new architectures are designed to work on low precision to squeeze out more performance. These low precision formats include half precision (FP16), BFloat16, int16, and int8. Quantization is a technique widely used in deep learning inference [57, 64]. While the model is usually still trained in single precision, quantization compresses the data and uses lower precision to carry out the computation in inference stage, which applies the trained model to new data for real applications. For an int8 quantized model, the data is converted into 8 -bit integers. The computation and communication are reduced $4 \times$ compared to 32 -bit single precision, while the accuracy lost is acceptable (usually $<1 \%$ for predictive models). Integer arithmetic is available on most hardware architectures. field programmable gate arrays (FPGAs) are usually more capable in integer operations and might not have floating-point number arithmetic units. New application-specific integrated circuits
(ASICs) for deep learning inference are also moving toward using mostly integer arithmetic for quantized neural networks.

Back to numerical linear algebra, a new mixed-precision algorithm has been developed to utilize the low accuracy factorization results as the preconditioner of the generalized minimal residual (GMRES)-based iterative refinement for solving of linear systems of equations. In this work, we are looking for opportunities to replace the numerical linear algebra workload with integer arithmetic, starting with LU factorization with the same goal of being applied as the preconditioner of GMRES.

## 2 Literature Review and Background

### 2.1 Iterative Refinement and Mixed-Precision Algorithms for Numerical Linear Algebra

Wilkinson[69, 46] first proposed the iterative refinement method to iteratively improve the accuracy of numerical solutions to systems of linear equations. With each iteration, the method solves a linear system with residual error on the right-hand side for the correction vector. Then, the correction is added to the solution vector to obtain higher accuracy. It is implemented in LAPACK[5], and the numerical behavior is well studied[41, 33]. It has been applied in solving the system in 32-bit floating-point numbers, then refining the solution to 64 -bit accuracy $[43,13]$. It is also used for solving the system in 64 -bit and refining to 128 bit quadruple precision for applications that require higher accuracy than double precision. Because the major computation of factorizing the matrix is done in low precision, iterative refinement could take advantage of higher performance from low-precision matrix-matrix operations and still achieve the desired accuracy.

Other methods were suggested to replace the triangular solve from LU factorization in iterative refinement for ill-conditioned linear systems. By replacing it with the generalized minimal residual (GMRES) algorithm[59], which is an iterative method itself, the whole method becomes nested inner-outer iterations. This is in fact similar to the flexible GMRES[58] method, in which the outer iteration allows preconditioner changing. The
method has also been studied both in theoretical and in real computational terms[12, 8]. Carson and Higham[14, 15] extended the method to use three precisions $u, u_{f}, u_{r}$ for the working precision, the factorization precision, and the residual computation precision, respectively. In section 2.4, more detail regarding GMRES-based iterative refinement method(GMRES-IR) is covered.

The development of hardware architectures is also an important force propelling mixedprecision algorithms. We measure the performance here in how many floating-point operations can be carried out per second, or $F L O P / s$. While, natively, there is usually a $2 \times$ performance difference between 32 -bit single and 64 -bit double precision on traditional architectures like x86 CPUs, some specialized architectures would have more limited hardware support for double precision, causing a greater performance gap. For example, the IBM first generation CELL processor has $10 \times$ peak performance difference between single and double precision. [42]. This gap can also be found on gaming-grade GPUs, since they do not need hardware double precision support for graphics processing. With the rapidly increasing amount of computational power needed for deep learning [45], new hardware architectures started being equipped with low-precision capabilities. The NVIDIA Pascal P100 GPU[48] provides 16 -bit half precision with $4 \times$ performance over double precision. The NVIDIA Volta V100 GPU[49] that followed took it further with the Tensor Core, which computes a small $4 \times 4$ matrix-matrix multiplication of 16 -bit, half-precision input and 32bit single-precision accumulation. This yields $16 \times$ theoretical peak performance difference between the 16-bit tensor core operation and double precision. Haidar et al.[30] applied the GMRES-IR method on NVIDIA Volta and achieved $4 \times$ speedup over a pure double precision solver. However, due to the very limited range of these low-precision formats, the input matrix needs to be properly scaled to prevent loss of accuracy [34].

### 2.2 Numerical Representations

Table 1.1 and figure 1.1 summarizes all the common numerical representation formats. The most used floating-point number representation nowadays comes from the IEEE 754-1985 standard[37]. It defines the single-precision number with 32-bits width and double precision with 64-bits width. The format consists of three fields: a sign bit, exponent bits, and

Table 1.1: Summary of numerical representation formats

| Name | Exponent bits | Mantissa bits | Precision $\epsilon$ | Max |
| :--- | :---: | :---: | :---: | :---: |
| bfloat16 (BF16) | 8 | 7 | $O\left(10^{-2}\right)$ | $O\left(10^{3} 8\right)$ |
| half precision (IEEE FP16) | 5 | 10 | $O\left(10^{-3}\right)$ | 65504 |
| single precision (IEEE FP32) | 8 | 23 | $O\left(10^{-7}\right)$ | $O\left(10^{3} 8\right)$ |
| double precision (IEEE FP64) | 11 | 52 | $O\left(10^{-16}\right)$ | $O\left(10^{3} 08\right)$ |
| extended precision (IEEE FP80) | 15 | 64 | $O\left(10^{-20}\right)$ | $O\left(10^{4} 932\right)$ |
| quadruple precision (IEEE FP128) | 15 | 112 | $O\left(10^{-34}\right)$ | $O\left(10^{4} 932\right)$ |
| int16 | 0 | 15 | 1 | 32767 |
| int32 | 0 | 31 | 1 | $O\left(10^{9}\right)$ |

 FP16 $\|_{\|}^{\|}$



 INT16 $\|_{\|}^{\|}$
INT32 $\|_{\|}$
Figure 1.1: Illustration of numerical representation formats. Each rectangle represent one bit. Different floating-number formats are constructed with different number of exponent bits and mantissa bits (fraction bits).
mantissa bits (also called fraction bits). Denormalized numbers are also defined as a special case to represent more numbers close to zero and reduce the chance of underflow. IEEE 754 also defines the rounding rules and required operations, including add, subtract, multiply, divide, and fused multiply-add. The extended precision with 80 -bits width is for the cases in which double precision is not sufficient. However, the software language support for it is limited. In a 2008 revision[38], IEEE 754 added half precision with 16 -bits width and quadruple precision with 128 -bits width.

The Google Brain floating-point format(bfloat16, BF16)[9] was implemented on Google Tensor Processing Units (TPUs) v3 to overcome the problem of the very limited range from half precision. The bfloat16 has the same exponent bits as single precision, and thus roughly the same range.

There is also fixed-point number which uses an integer plus a fixed exponent to represent a number. It is usually used in applications with fixed range, such as digital signal processing. The resolution of fixed-point numbers is in the range - contrary to floating-point numbers, where the resolution varies with the exponent. The regular two's complement integers can also be viewed as special cases with an exponent of 0 , as listed in table 1.1.

### 2.3 LU Factorization with Partial Pivoting

LU factorization (Gaussian elimination) with partial pivoting is the standard method for solving linear systems of equations. Partial pivoting is performed for numerical stability: to prevent zeros or small elements occurring on the diagonal. As outlined in algorithm 1, each iteration factorizes one column of input matrix $A$. First, the pivoting step finds the largest element and performs the row swaps so it will be on diagonal. Then, it uses row operations to eliminate the elements below diagonal in the working column of $U$ and store the operations in $L$. The last step in the iteration is updating the remaining matrix, which is also where the dominating term of computational complexity $\frac{2}{3} n^{3}$ comes form. At the end, we will have the lower-triangular matrix $L$ with unit diagonal, the upper-triangular matrix $U$, and the permutation matrix $P$ such that $L \times U=P \times A$. Then the system of linear equations

```
Algorithm 1 (Right Looking) LU factorization with partial pivoting
    Input: Matrix \(A \in \mathbb{R}^{n \times n}\)
    Output: Lower triangular matrix \(L \in \mathbb{R}^{n \times n}\) with unit diagonal, upper triangular matrix
    \(U \in \mathbb{R}^{n \times n}\), and permutation matrix \(P\) such that \(L \times U=P \times A\).
    function \([L, U, P] \leftarrow \mathbf{l u}(A)\)
        \(U \leftarrow A ; L \leftarrow I ; P \leftarrow I \quad \triangleright\) Initialize matrices
        for \(i=1 \ldots n\) do
            pivot \(\leftarrow(\arg \max |U[i: n, i]|)+i-1 \triangleright\) Find the pivot index with largest element.
            \(\operatorname{swap}(U[i,:], U[\) pivot, :] \() \quad \triangleright\) Swap rows.
            \(\operatorname{swap}(L[i,:], L[\) pivot, :] \()\)
            \(\operatorname{swap}(P[i,:], P[\) pivot, : \(])\)
            \(L[i+1: m, i] \leftarrow U[i+1: n, i] \div U[i, i] \quad \triangleright\) Scale the column and store in \(L\).
            \(U[i+1: m, i] \leftarrow 0\)
            \(U[i+1: n, i+1: n] \leftarrow U[i+1: n, i+1: n]-L[i+1: n, i] \times U[i, i+1: n]\)
                        \(\triangleright\) Rank-1 update.
        end for
    end function
```

$$
\begin{equation*}
A x=b \tag{1.1}
\end{equation*}
$$

can be solved with triangular forward and backward substitutions.

$$
\begin{gather*}
P A x=P b  \tag{1.2}\\
L U x=P b  \tag{1.3}\\
x=U^{-1}\left(L^{-1}(P b)\right) \tag{1.4}
\end{gather*}
$$

In practice[5], this is usually implemented in a blocked fashion to work on $n b$ columns at once. The panel composed of the $n b$ columns will be factorized without touching the remaining unfactorized matrix. Then, the aggregated rank $n b$ update will be applied at once. This approach utilizes the memory hierarchies in modern hardware architectures to fit the panel a in high-bandwidth, low-latency cache, and it also takes advantage of high performance Basic Linear Algebra Subprograms (BLAS) level 3 matrix-matrix operation

```
Algorithm 2 Left Looking LU factorization with partial pivoting.
    Input: Matrix \(A \in \mathbb{R}^{n \times n}\)
    Output: Lower triangular matrix \(L \in \mathbb{R}^{n \times n}\) with unit diagonal, upper triangular matrix
    \(U \in \mathbb{R}^{n \times n}\), and permutation matrix \(P\) such that \(L \times U=P \times A\).
    function \([L, U, P] \leftarrow \operatorname{lu} \operatorname{ll}(A)\)
        \(U \leftarrow A ; L \leftarrow I ; P \leftarrow I \quad \triangleright\) Initialize matrices.
        for \(i=1 \ldots n\) do
            for \(j=1 \ldots i-1\) do
                \(U[j+1: n, i] \leftarrow U[j+1: n, i]-L[j+1: n, j] \times U[j, i]\)
                                    \(\triangleright\) Apply previous operations to update.
            end for
            pivot \(\leftarrow(\arg \max |U[i: n, i]|)+i-1 \triangleright\) Find the pivot index with largest element.
            \(\operatorname{swap}(U[i,:], U[\) pivot, \(:]) \quad \triangleright\) Swap rows.
            \(\operatorname{swap}(L[i,:], L[\) pivot, :] \()\)
            \(\operatorname{swap}(P[i,:], P[\) pivot, : \(])\)
            \(L[i+1: n, i] \leftarrow U[i+1: n, i] \div U[i, i] \quad \triangleright\) Scale the column and store in \(L\).
            \(U[i+1: n, i] \leftarrow 0\)
        end for
    end function
```

routines. The introduced block size $n b$ becomes a performance-tuning parameter, and the optimal would depend on the hardware capabilities, software environments, and problem properties. The permutation matrix $P$ is usually stored as a vector of row indicates for performance reasons. The whole factorization is also normally done in-place, with both output $L$ and $U$ stored in $A$ to reduce the memory usage.

Algorithm 1 is also called "right-looking" LU factorization, as it looks toward the right side after factorization to update the remaining matrix. In contrast, the "left-looking" LU factorization does not perform the update after factorization. Figure 1.2 is an illustration comparing the right-looking and left-looking algorithms. Before factorizing the column, it looks at the left side and applies all the previous row operations. We can see that algorithm 2 is almost the same as algorithm 1, except the update in line 7 happens before the factorization instead of after. The blocking strategy can also be applied to left-looking variants to improve the performance. In general, the left-looking variants have less I/O compared to the rightlooking variants, where more parallelism can be exploited from the update step. There are other variants $[6,28]$ maintaining a different intermediate status during the factorization, as well as the recursive approach, which recursively divides the matrix by half in each step.

## Right-looking



## Left-looking



Figure 1.2: Illustration of right-looking and left-looking algorithms. The right-looking algorithm updates the remaining unfactored matrix toward the right after factorizing the column, whereas the left-looking algorithm updates the current column just before the factorization using previous results from the left.

### 2.4 Iterative Refinement with LU Factorization and Preconditioned Generalized Minimal Residual Method (GMRES)

This section introduces iterative refinement with both LU factorization and GMRES as the solving step.

For solving the linear system of equations $A x=b$, let $x_{0}$ be our initial solution. The iterative refinement performs the following operations in each iteration:

1. Compute residual $r_{i} \leftarrow b-A x_{i}$.
2. Solve the linear system $A c_{i}=r_{i}$ for correction.
3. Update the solution $x_{i+1}=x_{i}+c_{i}$

If all the steps are computed in exact math, then the algorithm would converge in one step. However, the floating-point arithmetic is performed with finite accuracy and requires the iteration to be repeated, especially if the matrix $A$ is ill-conditioned. The solving step, if not explicitly specified, is usually done with LU factorization with partial pivoting: $c_{i} \leftarrow$ $U^{-1}\left(L^{-1}(P b)\right)$. We name this method as LU-IR for easier reference. Because the matrix is the same, the factorization can be repeatedly used through iterations. If the same precision is used for all the steps in each iteration, it is called fixed-precision iterative refinement. This can improve the backward error from LU factorization without strong, stable pivoting. For mixed-precision iterative refinement, different precisions are used during the refinement process. The factorization and solving steps usually use low precision, as they are the computationally costly components. Carson and Higham[15] extended the method to use three precisions: working precision $u$, factorization precision $u_{f}$, and residual precision $u_{r}$. Residual precision $u_{r}$ is used at the first step for computing and accumulating the residual vector $r_{i}$. Factorization precision $u_{f}$ is for LU factorization as well as the solving step. The solution and the update are in working precision $u$. It has been shown that LU-IR can refine the solution to double-precision backward error, with $u$ and $u_{r}$ in double and $u_{f}$ in half or single, if the condition number $\kappa_{\infty}(A)$ is smaller than $10^{4}$ or $10^{8}$, respectively.

Alternatively, one can use the preconditioned generalized minimal residual (GMRES) method [58] in the solving step. This will be referred to as GMRES-IR[14, 15]. The LU
factorization here is used as the preconditioner of GMRES. GMRES approximates the solution vector in a Krylov subspace with minimal residual. GMRES is an iterative method itself, and in each iteration it does the following:

1. Generate orthonormal vector $q_{i}$ with Arnoldi method to expand the basis of Krylov subspace $Q$ by one dimension.
2. Find $y_{i}$ to minimize the residual.
3. Compute $x_{i}=Q y_{i}$

Preconditioned GMRES solves the system $U^{-1} L^{-1} A x=r$. The higher the accuracy of the LU factorization, the faster GMRES converges as the preconditioned system is closer to identity. The GMRES-IR method involves nested iterations, and the stopping criteria of inner iteration (GMRES) needs to be tuned. Compared to LU-IR, GMRES-IR works for some matrices with greater condition numbers[15].

## 3 Algorithm

In this section, the number representations being used are first described. Then the proposed integer arithmetic-based LU factorization algorithm and blocked version with dynamic column scaling is described. Note that in this work, only real numbers are consideredbut it is straightforward to extend the algorithms to complex numbers.

### 3.1 Fixed-Point Representation

Binary fixed-point number representation is utilized to carry out the numerical operations with integer arithmetic. To describe the binary presentation, the Q format notation is usd, originating from digital signal processors (DSPs)[39] . Qx.y represents a sign bit in front of the most significant bit, followws by x bits in integer portion, and y bits in fractional portion, in two's complement. Figure 1.2 shows an example of Q3.12.

0010001000000000 represents $2^{1}+2^{-3}=2.125$ in Q3.12. The range of Q3.12 is $[-8,8)$ and the resolution is $2^{-12}$ in the range. One can easily covert a floating-point number to

Table 1.2: Bit fields in Q3.12. It has one sign bit, 3 integer bits, and 12 fraction bits.

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | $\ldots$ | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Value | $\pm$ | $2^{2}$ | $2^{1}$ | $2^{0}$ | $2^{-1}$ | $2^{-2}$ | $\ldots$ | $2^{-12}$ |

Qx.y by multiplying $2^{y}$ and rounding to the nearest integer. Another way to think of the format is scaling up the numbers by $2^{y}$ then doing the operations in integer arithmetic. For two Q numbers with the same scale $y$, the basic operation can be easily done with integer arithmetic plus shifts (for multiplying and dividing by $2^{y}$ ).

$$
\begin{align*}
& \frac{a}{2^{y}} \pm \frac{b}{2^{y}}=\frac{a \pm b}{2^{y}}  \tag{1.5}\\
& \frac{a}{2^{y}} \times \frac{b}{2^{y}}=\frac{(a \times b) \div 2^{y}}{2^{y}}  \tag{1.6}\\
& \frac{a}{2^{y}} \div \frac{b}{2^{y}}=\frac{(a \div b) \times 2^{y}}{2^{y}} \tag{1.7}
\end{align*}
$$

For operations of two numbers with different scale $y$, additional shifts are needed, but this is usually a cheap operation. Addition and subtraction would first need to be shifted into the same scale. For multiplication or division:

$$
\begin{align*}
& \frac{a}{2^{y}} \times \frac{b}{2^{v}}=\frac{(a \times b) \div 2^{v}}{2^{y}} \text { or } \frac{(a \times b) \div 2^{y}}{2^{v}}  \tag{1.8}\\
& \frac{a}{2^{y}} \div \frac{b}{2^{v}}=\frac{(a \div b) \times 2^{v}}{2^{y}} \text { or } \frac{(a \times b) \div 2^{2 v-y}}{2^{v}} \tag{1.9}
\end{align*}
$$

### 3.2 Integer Arithmetic-Based LU Factorization with Partial Pivoting

Algorithm 3 is the unblocked integer LU factorization with partial pivoting. First, the two fixed-point formats for output $L$ and $U$ have to be decided. The input double-precision matrix is first normalized into $[-1,1]$ with dividing by the maximum absolute value among the elements. Next, it is converted into Qz.w (the format for $U$ ). The main loop is almost the same as the standard floating-point LU factorization with partial pivoting, except some divisions of power of 2 via integer shift is required for converting and keeping the fixed-point formats. In the implementation, the permutation is usually stored as an index vector to save space instead of explicitly forming the matrix. With partial pivoting, the largest element in the column is picked as pivot. The remaining elements in $L$ will be smaller or equal to 1 . However, the range of Q. 31 does not include positive 1. So Q1. 30 would be the best format for $L$. If the system can clamp the overflowing int32 value back to $2^{31}-1, Q .31$ could be

```
Algorithm 3 LU factorization with partial pivoting with Qx.y for \(L\) and Qz.w for \(U\) using
32-bit integer arithmetic (Default: Q1. 30 for \(L\) and Q3. 28 for \(U\) ).
Input: Matrix \(A \in \mathbb{R}^{n \times n}\), integer parameters y and w for fixed-point formats.
Output: Lower triangular matrix \(L\) stored in Qx.y with unit diagonal, upper triangular
matrix \(U\) stored in Qz.w, permutation matrix \(P\), and scalar \(\alpha\) such that \(L \times U=\)
\(P \times(A / \alpha)\).
function \([L, U, P, \alpha] \leftarrow\) lu_int \((A)\)
    \(\alpha \leftarrow \max (|A|) ; A \leftarrow A / \alpha \quad \triangleright\) Normalize \(A\)
    \(U \leftarrow \operatorname{int} 32\left(A \times 2^{w}\right) \quad \triangleright\) Convert \(A\) into Qz.w and store in \(U\).
    \(L \leftarrow \operatorname{int32}(I) ; P \leftarrow I\)
    for \(i=1 \ldots n\) do \(\quad \triangleright\) Main loop over columns
        pivot \(\leftarrow(\arg \max |U[i: n, i]|)+i-1 \triangleright\) Find the pivot index with largest element.
        \(\operatorname{swap}(U[i,:], U[\) pivot, :] \() \quad \triangleright\) Swap rows.
        \(\operatorname{swap}(L[i,:], L[\) pivot, :] \()\)
        \(\operatorname{swap}(P[i,:], P[\) pivot, : \(])\)
        \(\beta \leftarrow \operatorname{int} 64\left(2^{62}\right) \div \operatorname{int} 64(U[i, i]) \quad \triangleright\) Compute the scalar.
        \(L[i+1: n, i] \leftarrow \operatorname{int} 32\left(\beta \times \operatorname{int} 64(U[i+1: n, i]) \div \operatorname{int} 64\left(2^{62-y}\right)\right)\)
                        \(\triangleright\) Scale the column and store in \(L\).
        \(U[i+1: n, i] \leftarrow 0\)
        \(U[i+1: n, i+1: n] \leftarrow U[i+1: n, i+1: n]-L[i+1: n, i] \times U[i, i+1: n] \div 2^{y}\)
                                    \(\triangleright\) Integer rank-1 update.
    end for
end function
```

used for $L$ with minimal accuracy loss for the case in which the element is the same as the pivot in the column. Clamping for overflowing is also called saturation arithmetic. Modular arithmetic, conversely, is easier to implement in hardware but the algorithm would fail when overflow occurs and the positive overflowing number wrap around and became negative.

The other format Qz.w is for the intermediate steps and the output in $U$. Although the matrix is normalized at the beginning, the elements will grow during factorization. Thus the range has to be wider to prevent overflow: this is a trade-off between accuracy and the ability to handle larger matrices with greater growth rates, which is a non-trivial property of the matrix.

Line 12 and 13 requires int64 arithmetic for intermediate values. Line 12 has integer division but only once per column. Effectively, these 2 lines are performing:

$$
\begin{equation*}
L[i+1: n, i] \leftarrow U[i+1: n, i] /\left(U[i, i] / 2^{y}\right)=U[i+1: n, i] \times\left(2^{62} / U[i, i]\right) /\left(2^{62} / 2^{y}\right) \tag{1.10}
\end{equation*}
$$

int32 division could be used, but it would require first scaling the pivot $U[i, i]$ to roughly square root of the range $\sqrt{2^{31}} \approx 2^{16}$ to prevent the divisor being too large. $2^{62}$ is selected as the largest positive order of 2 for easy division with shift. The computational cost of $\frac{2}{3} n^{3}$ comes from the update in line 16. It also requires the int64 intermediate result from multiplication of int32 numbers. However, most of the architectures have mulhi instruction, which will return the high 32 bits from the full 64 -bit multiplication result. For the case using Q1. 30 format for $L$, what is needed is a shift of 30 bits, but mulhi is effectively a 32 bits shift—resulting in the loss of 2 least significant bits of information but possibly a big performance gain in practice.

This algorithm can also be extended down to int16 with int32 for intermediate results, and default Q1. 14 for $L$, Q3. 11 for $U$. Alternatively, the values can still be stored in int32, and only before reading for the update in line 16 , truncating the input into int16 with the most significant bits. Line 16 would then become:

$$
\begin{align*}
& A_{\text {int }}[i+1: n, i+1: n] \leftarrow A_{\text {int }}[i+1: n, i+1: n]-\left(A_{\text {int }}[i+1: n, i] / 2^{16}\right)  \tag{1.11}\\
& \times\left(A_{\text {int }}[i, i+1: n] 2^{16}\right) \div\left(2^{y} / 2^{32}\right)
\end{align*}
$$

The truncation to fit int16 with the most significant bits is done by dividing $2^{16}$. The 32-bit result from multiplying 16-bit integers is fully accumulated, so no explicit type casting is shown here. This is similar to the NVIDIA Volta Tensor Core[49] method of using half precision FP16 as input and accumulating in single precision FP32.

### 3.3 Left-Looking Integer LU with Dynamic Column Scaling

Algorithm 3 has a fixed range for matrix $U$. In general, however, the element in $U$ would grow during factorization, even with pivoting. To overcome this issue, we first observe the row operations, Gaussian elimination. In the update step, each element will adds or subtracts

```
Algorithm 4 The left-looking dynamic column scaling LU factorization with partial pivoting with Qx.y for \(L\) and Qz.w for \(U\) using 32-bit integer arithmetic (Default: Q1. 30 for both \(L\) and \(U\) ).
Input: Matrix \(A \in \mathbb{R}^{n \times n}\), integer parameters y and w for fixed-point formats.
Output: Lower triangular matrix \(L\) stored in Qx.y with unit diagonal, upper triangular matrix \(U\) stored in Qz.w, column scaling diagonal matrix \(C\), permutation matrix \(P\), and scalar \(\alpha\) such that \(L \times U \times C=P \times(A / \alpha)\).
function \([L, U, P, C, \alpha] \leftarrow\) lu_int_ll \((A)\)
\(\alpha \leftarrow \max (|A|) ; A \leftarrow A / \alpha \quad \triangleright\) Normalize \(A\)
\(U \leftarrow \operatorname{int} 32\left(A \times 2^{w}\right) \quad \triangleright\) Convert \(A\) into Qz.w and store in \(U\).
\(L \leftarrow \operatorname{int32(I)} ; C \leftarrow I ; P \leftarrow I\)
for \(i=1 \ldots n\) do \(\quad \triangleright\) Main loop over columns for \(j=1 \ldots i-1\) do \(\triangleright\) Left-looking update loop
\(U[j+1: n, i] \leftarrow U[j+1: n, i]-U[j, i] \times L[j+1: n, j] \div 2^{y}\)
if \(\max (|U[j+1: n]|)>2^{w}\) then \(\triangleright\) Scale down if close to the range
\(U[j+1: n, i] \leftarrow U[j+1: n, i] \div 2\)
\(C[i, i] \leftarrow C[i, i] \times 2\)
end if
end for
pivot \(\leftarrow(\arg \max |U[i: n, i]|)+i-1 \triangleright\) Find the pivot index with largest element.
\(\operatorname{swap}(U[i,:], U[\) pivot, :] \() \quad \triangleright\) Swap rows.
\(\operatorname{swap}(L[i,:], L[\) pivot, :] \()\)
\(\operatorname{swap}(P[i,:], P[\) pivot, : \(])\)
\(\beta \leftarrow \operatorname{int} 64\left(2^{62}\right) \div \operatorname{int} 64(U[i, i]) \quad \triangleright\) Compute the scalar. \(L[i+1: n, i] \leftarrow \operatorname{int} 32\left(\beta \times \operatorname{int} 64(U[i+1: n, i]) \div \operatorname{int} 64\left(2^{62-y}\right)\right)\)
\(\triangleright\) Scale the column and store in \(L\). \(U[i+1: n, i] \leftarrow 0\)
end for end function
```

a value which comes from another element in the same column times a scalar. Because of partial pivoting, this scalar is always less than or equal to 1 . So, for each column the elements are likely in the same magnitude. Therefore, based on left-looking LU factorization, we monitor the range during the update step and dynamically scale the column if needed.

Algorithm 4 uses the property of left-looking LU, in which the column is updated with all the accumulated row operations at once. Line 8 is the update before pivoting and factorization. Line 10 checks for the growth of the working column. Here $[-1,1]$ is treated as the normal working range for $U$. If the maximum value is greater than 1 , it is considered too close to the range. The whole column will be scaled down by dividing by 2 , and the
corresponding element in column scaling diagonal matrix $C$ is updated. Doing the scaling ensures that $U$ does not grow out of the range. Thus, the format Q1. 30 is also sufficient for $U$; but if the range is greater, the check in line 10 can be performed less frequently as there is a bigger buffer for elements growth.

## 4 Numerical Results

The algorithms are implemented and experimented in MATLAB version R2020a update 3. We are using all the built-in datatypes: double, single, half, int64, int32, int16. For the LU factorization of floating-point number types double and single, the built-in function $[L, U, P]=l u(A)$ will be used. The half precision type was introduced in version R2018b. It is not supported by the lu() function, so algorithm 1 is implemented. $\mathrm{rng}(0)$ is also called before each numerical experiment for reproducibility. As we are focusing on using the low precisions for the factorization, all the test matrices are first generated in double precision. They are scaled and converted into the desired format for the target algorithm of factorization. Afterward, the factorized results are converted back to double, then the following error analysis or iterative will be performed, all in double precision.

### 4.1 Column Growth

Figure 1.3 shows column growth. Here we have 5 different matrices of size $n=2000$, all solved by double-precision LU factorization with partial pivoting. The 5 matrices are:

1. $A 1=r a n d(n):$ Each element is a uniformly distributed random number in $(0,1)$.
2. $\mathrm{A} 2=2 * \mathrm{rand}(\mathrm{n})-1$ : Each element is a uniformly distributed random number in $(-1,1)$.
3. A3 = gallery('rando',n,2) : Each element is randomly selected from -1 or 1 with equal probability 0.5 .
4. A4 = gallery('randsvd',n, cond=1e4): Random matrix with pre-assigned geometrically distributed singular values in $\left(1, \frac{1}{10^{4}}\right)$. The condition number of $A$ is $10^{4}$.
5. A5 = gallery('randsvd',n,cond=1e8) : Random matrix with pre-assigned geometrically distributed singular values in $\left(1, \frac{1}{10^{8}}\right)$. The condition number of $A$ is $10^{8}$.

Matrices A4 and A5 are first normalized so that the largest element is 1, to be comparable with other matrices. Scaling the matrix does not affect its condition number. Figure 1.3 plots the largest element in each column from the factorized matrix $U$ in double precision. The x -axis is column index and y-axis is its absolute value. It is well known that the worst case of column growth for partial pivoting is $2^{n-1}$, exponent of matrix size $n$. However, it is a very artificial matrix and in general the growth is mild. Matrix A3 has the largest growth among these 5 matrices, as all the elements have the same magnitude but with random signs - and the growth is far from the worst case $2^{n-1}$. Matrices A1 and A2 are being used in a lot of testers in numerical linear algebra software packages, including HPL[32] and HPL-AI[35] benchmarks. The growth is in the same trend but proportionally smaller than A3. Matrices A4 and A5 are with pre-assigned singular values, and the growth is bounded and would not increase with the column index. Thus, the default fixed-point representation Q3. 28 for $U$ in algorithm 3 would be sufficient with the range $[-8,8)$.

### 4.2 Backward Error and Residual

Here we show the backward error of using LU factorization with different precisions without iterative refinement. The backward error is computed as the following formula:

$$
\begin{equation*}
\text { Backward error }=\frac{\|A x-b\|_{\infty}}{\|A\|_{\infty}\|x\|_{\infty}+\|b\|_{\infty}} \tag{1.12}
\end{equation*}
$$

All the factorizations are first converted into double precision and perform the forward and backward substitutions in double precision as well.

Table 1.3 shows all the backward errors from different precisions and LU factorization algorithms. The same 5 matrices from section 4.1 are used. The matrix sizes are all $1000 \times$ 1000. int32, int16, and int16 with int32 accumulation are the results from algorithm 3, which is the basic right-looking integer LU with fixed range. The other set of results are from algorithm 4 with dynamic column scaling. The results from floating-point formats double, single and half are included for reference. Firstly, we can see that for the matrices A1,


Figure 1.3: Column growth with respect to column index of 5 different matrices. Each dot is representing the largest value of the column in factorized matrix $U$.

Table 1.3: Backward errors $\frac{\|A x-b\|_{\infty}}{\|A\|_{\infty}\|x\|_{\infty}+\|b\|_{\infty}}$ from different precisions and algorithms versus different input matrices of size 1000 .

| Matrix | A 1 | A 2 | A | A | A 4 |
| :--- | ---: | ---: | ---: | ---: | ---: |
| double precision | $4.1211 \mathrm{e}-16$ | $7.9431 \mathrm{e}-16$ | $7.3449 \mathrm{e}-16$ | $1.5717 \mathrm{e}-16$ | $1.3303 \mathrm{e}-16$ |
| single precision | $5.9032 \mathrm{e}-08$ | $9.8795 \mathrm{e}-08$ | $1.3103 \mathrm{e}-07$ | $2.5210 \mathrm{e}-08$ | $1.2840 \mathrm{e}-08$ |
| half precision | $4.9475 \mathrm{e}-04$ | 0.0011 | $9.2630 \mathrm{e}-04$ | $1.7014 \mathrm{e}-04$ | $5.7523 \mathrm{e}-05$ |
| int32 | $0.0131(\mathrm{~F})$ | $0.0181(\mathrm{~F})$ | $0.0128(\mathrm{~F})$ | $1.2754 \mathrm{e}-08$ | $1.4654 \mathrm{e}-08$ |
| int32 column scaling | $1.4950 \mathrm{e}-08$ | $3.7596 \mathrm{e}-08$ | $3.1432 \mathrm{e}-08$ | $1.0937 \mathrm{e}-08$ | $5.6571 \mathrm{e}-09$ |
| int16 | $0.0138(\mathrm{~F})$ | $0.0203(\mathrm{~F})$ | $0.0191(\mathrm{~F})$ | $1.8103 \mathrm{e}-04$ | $7.8584 \mathrm{e}-05$ |
| int16 column scaling | $2.9552 \mathrm{e}-04$ | $5.8992 \mathrm{e}-04$ | $5.4373 \mathrm{e}-04$ | $1.4690 \mathrm{e}-04$ | $3.3742 \mathrm{e}-05$ |
| int16 with <br> int32 accumulation | $0.0133(\mathrm{~F})$ | $0.0147(\mathrm{~F})$ | $0.0138(\mathrm{~F})$ | $7.2001 \mathrm{e}-05$ | $2.7742 \mathrm{e}-05$ |
| int16 with <br> int32 accumulation <br> column scaling | $8.5443 \mathrm{e}-05$ | $1.4724 \mathrm{e}-04$ | $1.3596 \mathrm{e}-04$ | $4.6795 \mathrm{e}-05$ | $1.6791 \mathrm{e}-05$ |

A2, and A3 with unbounded column growth, the fixed-range integer LU will not be able to accommodate them. They failed due to integer overflow in the factorization process. The dynamic column scaling algorithm can successfully factor all the test matrices. The backward errors from int32 algorithms are about the same, with single-precision floating-point LU at the order of $10^{-8}$. The int32 results are about the same as half precision at the order of $10^{-5}$. Using int16 input with int32 accumulation only slightly improves the backward error.

Figure 1.4 plots the histogram of the elements in residual matrix $R=P A-L U$. The input matrix is A4, size 1000 with pre-assigned geometrically distributed singular values and a condition number of $10^{4}$. There is no significant difference in the distributions of residual elements while comparing the floating-point LU with proposed integer LU algorithms. The elements are at the order or $10^{-16}, 10^{-6}, 10^{-7}$, and $10^{-2}$ for double, single, int32, and int16 respectively. Although the residual from int16 seems to be really large, we will show that it still contains enough information for iterative refinement to converge for wellconditioned matrices.

Figure 1.5 shows the frequency of dynamic scaling happening in the int16 with int32 accumulation with column scaling algorithm. The input matrix is A1 with size 1000. A dot at $(i, j)$ represents scaling (via integer shift) happened at column $j$ while using the result from column $i$ to perform the update. Because the elements in the matrix is generated from a uniform distribution from 0 to 1 , the magnitude are at the same order. Under this distribution, the frequency of dynamic scaling happening at a log function, which is more frequent at the beginning at the update. Another way to think of it is to accumulate ones $1+1+1+\ldots$ in binary. It would need one more bit when it is at the order of 2 . This frequency is of course depend of the distribution of elements in the matrix. But as we can see in column growth plot (figure 1.3), uniformly distributed elements is a bad case in terms of large column growth. Other matrices would generally have much mild column growth.

### 4.3 Iterative Refinement Results

Here we use the factorization results from integer LU as low precision approximation and try to refine the result in double-precision accuracy. We set our convergence goal to be the backward error smaller than $10^{-15}$.


Figure 1.4: Histogram of elements in residual matrix $R=P A-L U$


Figure 1.5: The frequency of dynamic scaling happening in the int16 with int32 accumulation with column scaling algorithm. The input matrix is A1 with size 1000. A dot at $(i, j)$ represents scaling (via integer shift) happened at column $j$ while using the result from column $i$ to perform the update.

First we show the convergence results from LU-IR using different precisions in figure 1.6. The input $1000 \times 1000$ matrices are pre-assigned geometrically distributed singular values with condition numbers varying from $10^{2}$ to $10^{9}$, shown in different colored lines. The drop of backward error over iterations is also plotted. For both single precision and int32, LU-IR can work with the matrices with a condition number up to $10^{7}$. They all converge very quickly, under 10 iterations, indicating that the factorization is accurate and contains most of the information from the original matrix. Half precision can only work with extremely well conditioned matrices, which is condition number $\operatorname{cond}(A)=100$. The pure int16 integer LU is similarly very limited, with the $\operatorname{cond}(A)=10^{3}$ case eventually converging in 50 iterations. But if using int16 input with int32 accumulation, results from the integer LU can be applied to matrices with condition numbers up to $10^{5}$.

Figure 1.7 shows a case in which LU-IR would fail but GMRES-IR can still converge to double-precision accuracy. The matrix is still size 1000 but the pre-assigned singular values are arithmetically distributed with $\operatorname{cond}(A)=10^{5}$. For this particular matrix, the backward error of LU-IR cannot improve over $10^{-5}$, but GMRES-IR successfully converges with 20 inner iterations.

### 4.4 Discussion

The proposed integer arithmetic-based LU factorization algorithm have been shown as a good low precision approximate for iterative refinement when matrix is well conditioned. The dynamic column scaling algorithm solves the issue of column growth from factorization and can work on a wider range of matrices. For different integer lengths, the behavior is similar to the floating-point format with the same bit width. And by paying a little additional cost to do accumulation in int32 while the inputs for multiplication are still in int16, the iterative refinement can work for matrices with condition numbers up to $10^{5}$.

Here we do not have any performance results because of the lack of an integer BLAS library. Deep learning inference uses a special matrix-matrix multiplication of unsigned int8 times signed int8, which is too short from the numerical linear algebra point of view. Intel MKL does provide gemm_s16s16s32 which does int16 matrix-matrix multiplication with


Figure 1.6: LU-IR convergence of $1000 \times 1000$ matrices with pre-assigned geometrically distributed singular values


Figure 1.7: Comparing LU-IR and GMRES-IR for a matrix of size 1000, arithmetically distributed singular values and $\operatorname{cond}(A)=10^{5}$.
int32 accumulation and output. But the triangular solve TRSM function is also needed to complete the LU factorization.

## 5 Conclusion and Future Work

This work has demonstrated the possibility of using fixed-point number representation and integer arithmetic to solve systems of linear equations. It has also shown the potential of using mixed-precision iterative refinement algorithms to refine the solution to doubleprecision accuracy, which is usually desired by applications. The proposed algorithm uses the property of LU factorization with partial pivoting and assigns two different fixed-point representation for matrices $L$ and $U$. Algorithm 3 has been shown to work for matrices for which column growth is bounded. And the backward errors for int32 and int16 are at the same order as single and half precision, respectively. Moreover, using int16 as inputs with int32 accumulators enables GMRES-IR to converge on matrices with greater condition numbers, up to $10^{4}$. For the unbounded column growth matrices, algorithm 4 based on left-looking LU factorization is proposed with dynamic column scaling.

In order for a performance comparison, there are a few issues that need to be solved. First is the blocking of algorithms. Blocking is needed to utilize the cache to store the panels and update via high-performance matrix-matrix multiplication routines. This would, however. cause some conflicts with the dynamic column scaling, as the update will be accumulated and applied at once. The selection of fixed-point representation for $U$ needs to be reconsidered to have more room for growth and needs to be tune according to the matrix property. Other than that, we also do not have a full set of integer BLAS like we do for floating-point numbers. There are some specialized int8 routines from deep learning libraries such as gemmlowp and FBGEMM. Intel MKL also provides some special functions for integer, and gemm_s 16 s 16 s 32 is the closest we could use. However, the integer triangular solve trsm needs to be implemented to fit the algorithm.

Another path is to implement the algorithms on FPGAs. The data representation on FPGAs can be customized, as we can have integers with arbitrary number bits. Also,
integer arithmetic on FPGAs usually requires much less resources compared to floating-point numbers. This makes FPGAs a perfect target for implementing the algorithms.

Also, LU factorization is just one of the fundamental factorizations. We would like also to consider extending the approach to other factorizations, like QR. To further generalize the algorithm, complex numbers can also be adapted. The division of complex would be tricky but other parts of the algorithm should be similar to real number cases. Sparse matrices is a whole other domain can be explored.

## Chapter 2

## Iterative Refinement Algorithm for Symmetric Eigenvalue Problem on Modern Hardware

This article and its research work is done by student Yaohung M. Tsai under the guidance from Professor Jack Dongarra and Assistant Professor Piotr Luszczek. It has not been published in any proceedings at the time the dissertation was submitted to the University of Tennessee, Knoxville.


#### 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 $A x=\lambda x$, we revisit the SICE algorithm from 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 $3.6 \times$ speedup using the mixed-precision eigensolver with the blocked SICESM algorithm for iterative refinement when compared with full double complex precision solvers for the cases with a portion of eigenvalues and eigenvectors requested.


## 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

$$
\begin{equation*}
A x=\lambda x \text { where } A=A^{\top}, A \in \mathbb{R}^{n \times n} \tag{2.1}
\end{equation*}
$$

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: $A=A^{\mathrm{H}}$ 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 [14, 15, 30]. 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 [29, 31], the multicore and multiGPU eigensolvers' algorithms for refining eigenvalues should be reviewed carefully in order to ascertain the possibility to improve the performance especially on this new hardware.

## 2 Literature Review and Background

### 2.1 Eigenvalue refinement

Symm and Wilkinson[63] 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[22, 23, 24] later improved the algorithm with reduced computational cost and provided additional error analysis, including the comparison to Newton's method[55, 71], numerical results, and discussion of extending the algorithm for ill-conditioned problems with multiple close eigenvalues. More detail will be reviewed in subsection 2.4 as it is also the core of the algorithm used in this work.

Other related work from Stewart[62] and Chatelin[17] answered the same question from the point of view of the invariant subspace problem. Demmel[20] later pointed out that these two methods and the one from Dongarra, Moler, and Wilkinson [22, 23, 24] can all be reduced to solving the same Riccati equation. He also extended the algorithm for the generalized eigenvalue problem of the form $A x=\lambda B x$.

Alefeld and Spreuer[2] followed the same approach but specifically targeted problems with doubly-repeated or numerically close eigenvalues. Tisseur[65] did the analysis of Newton's
method under floating-point arithmetic for generalized eigenvalue problems. Prikopa and Gansterer[56] used the symmetry of the matrix and Householder tridiagonalization $A=$ $Q T Q^{\top}$ to reduce the computational cost.

Ogita and Aishima[50] 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[51, 52].

### 2.2 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.

### 2.2.1 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.[25] 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.[10] 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 bulgechasing 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[29, 31]. 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 $L Q$ 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. The illustration of comparing one stage and two stages algorithm can be found at figure 2.1.

### 2.2.2 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[70] 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 $Q R$ factorization


Figure 2.1: Illustration of comparing one stage and two stages tridiagonalization algorithm. The one stage at the top uses Householder transformations to reduce the matrix directly into tridiagonal but will touch the whole matrix for each column. The two stages algorithm at the bottom will first reduce the matrix into band by performing QR factorization in a submatrix, which will open touch the whole matrix after each submatrix (block) is factored. Then the second stage bulge-chasing to further reduce it to tridiagonal.
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[18] 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 $L D L^{\top}$ factorization. The Bisection method [68] 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)[54] 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 $\left(L D L^{\top}\right.$ factorization).

### 2.3 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[61] 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)[44] 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 matrixmatrix. 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[40] and oneMKL, IBM ESSL[36], ARM Performance Libraries[7], NVIDIA cuBLAS[47], AMD AOCL[3] for CPUs and rocBLAS[4] for GPUs. The implementations from academia and open-source communities also exist and include BLIS[67] and OpenBLAS[53], both of which build on the success story of portable performance of GotoBLAS[27].

LAPACK[5] 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, leastsquares 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[11] 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[1] 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[66] 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)[26] 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.

### 2.4 The SICE Algorithm

In this section, we review the SICE algorithm by Dongarra el al. [22, 23, 24]. Given the base eigenpair $\lambda, x$ and its nearby eigenpair $\lambda+\mu, x+\tilde{y}$, then based on the original eigenproblem we have:

$$
\begin{equation*}
A(x+\tilde{y})=(\lambda+\mu)(x+\tilde{y}) \tag{2.2}
\end{equation*}
$$

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. (2.2) we get:

$$
\begin{equation*}
(A-\lambda I) \tilde{y}-\mu x=\lambda x-A x-\mu \tilde{y} \tag{2.3}
\end{equation*}
$$

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:

$$
y^{\text {def }}=\left(\tilde{y}_{1}, \tilde{y}_{2}, \ldots, \tilde{y}_{s-1}, \mu, \tilde{y}_{s+1}, \ldots, \tilde{y}_{n-1}, \tilde{y}_{n}\right)
$$

So $y$ would encode information from both $\tilde{y}$ and $\mu$ and thus Eq. (2.3) becomes:

$$
\begin{equation*}
B y=r+y_{s} \tilde{y}=r+\mu \tilde{y} \tag{2.5}
\end{equation*}
$$

where $r=\lambda x-A x$ 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:

$$
\begin{equation*}
f(v) \equiv\binom{A x-\lambda x}{e_{s}^{\top} x-1}=0 \tag{2.6}
\end{equation*}
$$

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:

$$
J\binom{\tilde{y}}{\mu}=\left(\begin{array}{cc}
A-\lambda I & -x  \tag{2.7}\\
e_{s}^{\top} & 0
\end{array}\right)\binom{\tilde{y}}{\mu}=\binom{r}{0}=f(v)
$$

Expanding it, we arrive at Eq. (2.3) without the second-order term:

$$
\begin{equation*}
(A-\lambda I) \tilde{y}-\mu x=r \tag{2.8}
\end{equation*}
$$

This is the basic idea of the SICE algorithm: by iteratively solving Eq. (2.5) 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. (2.5). For any real matrix $A$, there exists an orthogonal matrix $Q$ and an upper quasi-triangular matrix $T$, such that

$$
\begin{equation*}
A=Q U Q^{\top} \tag{2.9}
\end{equation*}
$$

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. (2.5), we get:

$$
\begin{equation*}
\left[A_{\lambda}-\left(x+a_{\lambda s}\right) e_{s}^{\top}\right] y=\left(A+c e_{s}^{\top}\right) y=r+y_{s} \tilde{y} \tag{2.10}
\end{equation*}
$$

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

$$
\begin{gather*}
Q\left(U_{\lambda}+Q^{\top} c e_{s}^{\top} Q\right) Q^{\top} y=r+y_{s} \tilde{y}  \tag{2.11}\\
\left(U_{\lambda}+d f^{\top}\right) Q^{\top} y=Q^{\top} g \tag{2.12}
\end{gather*}
$$

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

$$
\begin{equation*}
Q_{1} d=\left(P_{2} P_{3} \ldots P_{n}\right) d=\gamma e_{1} \text { where } \gamma=\|d\|_{2} \tag{2.13}
\end{equation*}
$$

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

$$
\begin{equation*}
Q_{1}\left(U_{\lambda}+d f^{\top}\right)=Q_{1} U_{\lambda}+\gamma e_{1} f^{\top} \tag{2.14}
\end{equation*}
$$

The transformation $Q_{1}$ introduces one more nonzero element in the subdiagonal direction of $U_{\lambda}$. The new rank-one update $\gamma e_{1} \times f^{\top}$ 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 $\bar{U}_{\lambda}=Q_{2} Q_{1}\left(U_{\lambda}+d \times f^{\top}\right)$ in

$$
\begin{equation*}
\bar{U}_{\lambda} Q^{\boldsymbol{\top}} y=Q_{2} Q_{1} Q^{\top} g \tag{2.15}
\end{equation*}
$$

The triangular solve requires $O\left(n^{2}\right)$ operations while the remaining steps of the iteration are only $O(n)$. This procedure is shown in Algorithm 5.

```
Algorithm 5 SICE algorithm
    Input: Matrix \(A \in \mathbb{R}^{n \times n}\). An approximate eigenvalue \(\lambda\) and the corresponding eigenvector \(x\).
    iter \({ }_{\text {max }}\) denotes the maximum number of iterations.
    Output: Refined eigenvalue \(\lambda\) and its eigenvector \(x\).
    function \([\lambda, x] \leftarrow \operatorname{SICE}(A, \lambda, x, i\) ter \()\)
        \([Q, U] \leftarrow \operatorname{schur}(A) \quad \triangleright\) obtain Schur decomposition \(A=Q U Q^{\top}, Q Q^{\top}=I\).
        \([m, s] \leftarrow \max (\operatorname{abs}(x)) ; x \leftarrow x / m \quad \triangleright\) Normalizing \(x\) so that \(\|x\|_{\infty}=s_{x}=1\).
        for \(i\) in 1: iter \({ }_{\text {max }}\) do
                    \(r \leftarrow \lambda x-A x\)
                    \(c \leftarrow-x-a_{\lambda s}\)
                    \(d \leftarrow Q^{\top} c\)
                    \(f^{\top} \leftarrow Q(s,:)=e_{s}^{\top} Q \quad \triangleright s\)-th row of \(Q\).
                    \(\bar{U}_{\lambda} \leftarrow Q_{1}(U-\lambda I) ; \bar{d} \leftarrow Q_{1} d=\|d\|_{2} e_{1} \quad \triangleright\) Givens rotations \(Q_{1}\) from Eq. (2.13)
                    \(\bar{U}_{\lambda} \leftarrow \bar{U}_{\lambda}+\bar{d}(1) f^{\top}\)
                \(\bar{U}_{\lambda} \leftarrow Q_{2} \bar{U}_{\lambda} \quad \triangleright\) Givens rotations \(Q_{2}\) to introduce upper triangular form.
                Solve the triangular system \(\bar{U}_{\lambda} z=Q_{2} Q_{1} Q^{\top} r\)
                \(y \leftarrow Q y\)
                \(\lambda \leftarrow \lambda+y(s) \quad \triangleright\) Update eigenvalue.
                \(y(s) \leftarrow 0 \quad \triangleright\) Set \(y(s)\) to 0 .
                \(x \leftarrow x+y \quad \triangleright\) Update eigenvector.
                if desired accuracy is reached then
                break
            end if
        end for
    end function
```


## 3 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.

### 3.1 SICE-SM Algorithm

For symmetric eigenvalue problems, the matrix $A$ is first reduced to tridiagonal through unitary similarity transformations: $T=Q^{\top} A Q$ where $Q Q^{\top}=I$ and $T$ is a symmetric tridiagonal matrix. This corresponds to LAPACK routines SSYTRD and DSYTRD for singleand double-precision arithmetic, respectively. In the same fashion as SICE algorithm in

Section 2.4, we start with Eq. (2.10) and apply the tridiagonal reduction to it. Eqs. (2.11) and (2.12) in this case become

$$
\begin{equation*}
Q\left(T_{\lambda}+Q^{\top} c e_{s}^{\top} Q\right) Q^{\top} y=r+y_{s} \tilde{y} \tag{2.16}
\end{equation*}
$$

and

$$
\begin{equation*}
\left(T_{\lambda}+d \times f^{\top}\right) Q^{\top} y=Q^{\top} g \tag{2.17}
\end{equation*}
$$

the same with $d=Q^{\top} c, f^{\top}=e_{s}^{\top} Q$ and $g=r+y_{s} \tilde{y}$. Dongarra[22] discussed the approach of using the Sherman-Morrison formula[60]

$$
\begin{equation*}
\left(A-u v^{\top}\right)^{-1}=A^{-1}-\frac{A^{-1} u v^{\top} A^{-1}}{1+v^{\top} A^{-1} u} \tag{2.18}
\end{equation*}
$$

for solving the rank-one updated system. Eq. (2.17) 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 $\delta$ 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 $\left\|\lambda_{T}-\lambda_{A}\right\|=O\left(\epsilon_{\text {double }}\right)$ : 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. (2.18) to Eq. (2.17) we get

$$
\begin{equation*}
Q^{\top} y=\left(T_{\lambda}^{-1}-\frac{T_{\lambda}^{-1} d \times f^{\top} T_{\lambda}^{-1}}{1+f^{\top} T_{\lambda}^{-1} d}\right) Q^{\top} g \tag{2.19}
\end{equation*}
$$

or

$$
\begin{equation*}
Q^{\top} y=T_{\lambda}^{-1} Q^{\top} g-\frac{f^{\top}\left(T_{\lambda}^{-1} Q^{\top} g\right)}{1+f^{\top}\left(T_{\lambda}^{-1} d\right)} T_{\lambda}^{-1} d \tag{2.20}
\end{equation*}
$$

These involve solving the tridiagonal system $T_{\lambda}$ with two different right hand sides $d$ and $Q^{\top} g$. It can be easily done with the Thomas algorithm which is a special case of Gaussian

```
Algorithm 6 SICE-SM algorithm: SICE algorithm with Sherman-Morrison formula
    Input: Matrix \(A=A^{\top} \in \mathbb{R}^{n \times n}\). An approximate eigenvalue \(\lambda\) and the corresponding
    eigenvector \(x\). iter \(_{\max }\) denotes the maximum number of iterations.
    Output: Refined eigenvalue \(\lambda\) and eigenvector \(x\).
    function \([\lambda, x] \leftarrow \mathbf{S I C E}\) _SM \((A, \lambda, x\), iter \()\)
        \([Q, T] \leftarrow \operatorname{tridiag}(A) \quad \triangleright\) Tridiagonalization \(A=Q T Q^{\top}, Q Q^{\top}=I\).
        \([m, s] \leftarrow \max (\operatorname{abs}(x)) ; x \leftarrow x / m \quad \triangleright\) Normalization of \(x\) so that \(\|x\|_{\infty}=s_{x}=1\).
        for \(i\) in 1: iter \({ }_{\text {max }}\) do
            \(r \leftarrow \lambda x-A x\)
            \(c \leftarrow-x-a_{\lambda s}\)
            \(d \leftarrow Q^{\top} c\)
            \(f^{\top} \leftarrow Q(s,:)=e_{s}^{\top} Q \quad \triangleright s\)-th row of \(Q\).
            \(r h s \leftarrow Q^{\top} r\)
            \(u \leftarrow(T-\lambda I)^{-1} d\)
            \(v \leftarrow(T-\lambda I)^{-1} r h s\)
            \(y \leftarrow v-\frac{f^{\top} v}{1+f \tau u} u \quad \triangleright\) Sherman-Morrison formula
            \(y \leftarrow Q y\)
            \(\lambda \leftarrow \lambda+y(s) \quad \triangleright\) Update eigenvalue.
            if \(i \neq 1\) then
                \(y(s) \leftarrow 0 \quad \triangleright \operatorname{Set} y(s)\) to 0.
                    \(x \leftarrow x+y \quad \triangleright\) Update eigenvector.
                end if
                if desired accuracy reached then
                break
                end if
        end for
    end function
```

elimination. There are other parallel tridiagonal solvers available and we will discuss them in Section 3.3.1. We outline the SICE algorithm with Sherman-Morrison formula in Algorithm 6.

The main difference between Algorithms 5 and 6 is the use of the Sherman-Morrison formula to solve the system from line 12 to 14 instead of using the Givens rotations for that purpose. It is applied to solving the same tridiagonal system $T_{\lambda}$ with two different right hand sides $d$ and $Q^{\top} g$. 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 $\frac{x^{\top} A x}{x \top x}$ as the starting point. Apart from tridiagonalization,
the computational cost for algorithm 6 is dominated by the matrix-vector multiplications which require $O\left(n^{2}\right)$ operations. The remaining steps of the algorithm are all order $O(n)$ including the tridiagonal solve.

Alternatively, as described in [56], one can also solve the Jacobian matrix with the special structure $J=\left(\begin{array}{cc}T-\lambda I & y \\ z^{\top} & 0\end{array}\right)$, a tridiagonal system with an extra filled row and column at the end. However, comparing to solving tridiagonal systems which is well studied and exists several parallel solvers targeting different environments, it is hard to parallelize a solver for special structure and even make it scalable.

### 3.2 Blocked SICE-SM Algorithm

The computational cost of Algorithm 6 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[5] 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 2.1, 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 $3.6 \times$.

Table 2.1: 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 |

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 $f^{\top}$. 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. (2.16) and expand it without canceling any terms we get

$$
\begin{equation*}
\left(Q T_{\lambda} Q^{\top}+Q Q^{\top} v e_{s}^{\top} Q Q^{\top}\right) y=r+y_{s} \tilde{y} \tag{2.21}
\end{equation*}
$$

Again, for our mixed-precision scheme, we would like to perform the tridiagonalization in single precision. Hence $Q T_{\lambda} Q^{\top}$ is only an approximation of $A$ with precision $\epsilon_{\text {single }}$, i.e. $\left\|A_{\lambda}-Q T_{\lambda} Q^{\top}\right\| \sim O\left(\epsilon_{\text {single }}\right)$. The same applies to $Q Q^{\top}$ which is only an approximation of $I$ with $\left\|Q Q^{\top}-I\right\| \sim O\left(\epsilon_{\text {single }}\right)$. 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

$$
\begin{equation*}
X^{\prime}=X+\frac{1}{2} X\left(I-X^{\top} X\right) \tag{2.22}
\end{equation*}
$$

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 $I-$ $X^{\top} X$ also lets us detect if they converged to the same eigenvector. By combining these considerations, we arrive at Algorithm 7.

```
Algorithm 7 Blocked SICE-SM algorithm
    Input: \(A=A^{T} \in \mathbb{R}^{n \times n}\), initial eigenvectors \(X=\left[x_{1}\left|x_{2}\right| \ldots \mid x_{\ell}\right] \in \mathbb{R}^{n \times \ell}\) and the corresponding
    initial eigenvalues \(\Lambda=\left(\lambda_{1}, \lambda_{2}, \ldots, \lambda_{\ell}\right)^{T} \in \mathbb{R}^{l}\). 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 \()\)
        \([Q, T] \leftarrow \operatorname{tridiag}(A) \quad \triangleright\) Tridiagonalization \(A=Q T Q^{\top}, Q Q^{\top}=I\).
        for \(i\) in 1 : iter \(_{\text {max }}\) do
            \(s \leftarrow i\)
            \(R \leftarrow X \times\) diag_matrix \((\Lambda)-A \times X \quad \triangleright\) Residual vectors need higher precision.
            for \(j\) in \(1: \ell\) do
                \(c_{j} \leftarrow-x_{j}-A(:, s)\)
            end for
            Compose matrix \(C=\left[c_{1}\left|c_{2}\right| \ldots \mid c_{\ell}\right]\) from column vectors \(c_{j}\)
            \(C(s,:) \leftarrow C(s,:)+\Lambda^{T}\)
            \(D=\left[d_{1}\left|d_{2}\right| \ldots \mid d_{\ell}\right] \leftarrow Q^{T} \times C \quad \triangleright\) Can be in lower precision.
            \(R H S=\left[r h s_{1}\left|r h s_{2}\right| \ldots \mid r h s_{\ell}\right] \leftarrow Q^{T} \times R \quad \triangleright\) Can be in lower precision.
            \(f \leftarrow Q(s,:) \quad \triangleright\) s-th row of Q .
            for \(j\) in \(1: \ell\) do
                    \(u_{i} \leftarrow(T-\lambda I)^{-1} d_{i}\)
                    \(v_{i} \leftarrow(T-\lambda I)^{-1} r h s_{i}\)
                \(y_{i} \leftarrow v_{i}-\frac{f \top v_{i}}{1+f \tau u_{i}} u_{i} \quad \triangleright\) Sherman-Morrison
            end for
            Compose matrix \(Y=\left[y_{1}\left|y_{2}\right| \ldots \mid y_{\ell}\right]\) from correction vectors \(y_{j}\)
            \(Y \leftarrow Q \times Y\)
            \(\Lambda \leftarrow \Lambda+Y(s,:)^{T} \quad \triangleright\) Update eigenvalues.
            if \(i \neq 1\) then
                \(Y(s,:) \leftarrow 0 \quad \triangleright\) Set \(y_{i}(s)\) to 0 .
                \(X \leftarrow X+Y \quad \triangleright\) Update eigenvectors.
                Normalize eigenvectors \(x_{i}\) in \(X\).
            end if
            if desired accuracy reached then
                break
            end if
        end for
        \(X \leftarrow X+\frac{1}{2} X\left(I-X^{\top} X\right) \quad \triangleright\) Orthogonalization.
    end function
```

Because a Hermitian matrix can also be tridiagonalized into real matrix, algorithm 7 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.

### 3.3 Implementation Details

In this section, we will describe some of the details of our implementation. We implemented the Blocked SICE-SM (Algorithm 7) in two software packages: PLASMA[1] and MAGMA[66].

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 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 7 for both solvers.

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

> Algorithm 8 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 $O\left(n^{3}\right)$ 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 $\epsilon_{\text {single }}$ but not closer than $\epsilon_{\text {double }}$. 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. (2.17). Then the Blocked SICE-SM (Algorithm 7) 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 3.3.1.

```
Algorithm 9 Mixed precision one stage symmetric eigensolver with iterative refinement
    1: SSYTRD: Tridiagonalization via Householder transformations in single precision.
    2: DSTEDC: Tridiagonal symmetric eigensolver (divide and conquer) in double precision.
    3: SORGTR: Generate the transformation matrix \(Q\) from elementary reflectors in single
        precision.
    4: Blocked SICE-SM (algorithm 7) 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:

[^0]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 11 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 7) for iterative refinement.

Mixed precision for a two-stage eigensolver is actually more problematic performancewise. 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 11. 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.

### 3.3.1 Batched Tridiagonal Solver

Line 16 in Algorithm 7 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[21] 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 $[72,19,16]$ 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:

$$
\left[\begin{array}{ccccccccc}
b_{1} & c_{1} & & & & & &  \tag{2.23}\\
a_{2} & b_{2} & c_{2} & & & & & \\
& a_{3} & b_{3} & c_{3} & & & & \\
& & a_{4} & b_{4} & c_{4} & & & \\
& & & a_{5} & b_{5} & c_{5} & & \\
& & & & a_{6} & b_{6} & c_{6} & \\
& & & & & a_{7} & b_{7} & c_{7} \\
& & & & & & a_{8} & b_{8}
\end{array}\right]\left[\begin{array}{l}
x_{1} \\
x_{2} \\
x_{3} \\
x_{4} \\
x_{5} \\
x_{6} \\
x_{7} \\
x_{8}
\end{array}\right]=\left[\begin{array}{l}
y_{1} \\
y_{2} \\
y_{3} \\
y_{4} \\
y_{5} \\
y_{6} \\
y_{7} \\
y_{8}
\end{array}\right]
$$

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

$$
\left[\begin{array}{cccc}
b_{1}^{\prime} & c_{1}^{\prime} & &  \tag{2.24}\\
a_{3}^{\prime} & b_{3}^{\prime} & c_{3}^{\prime} & \\
& a_{5}^{\prime} & b_{5}^{\prime} & c_{5}^{\prime} \\
& & a_{7}^{\prime} & b_{7}^{\prime}
\end{array}\right]\left[\begin{array}{l}
x_{1} \\
x_{3} \\
x_{5} \\
x_{7}
\end{array}\right]=\left[\begin{array}{c}
y_{1}^{\prime} \\
y_{3}^{\prime} \\
y_{5}^{\prime} \\
y_{7}^{\prime}
\end{array}\right]
$$

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

$$
\begin{align*}
k_{1}=\frac{a_{i}}{b_{i-1}}, k_{2} & =\frac{c_{i}}{b_{i+1}} \\
a_{i}^{\prime}=-a_{i-1} k_{1}, b_{i}^{\prime} & =b_{i}-c_{i-1} k_{1}-a_{i+1} k_{2}  \tag{2.25}\\
c_{i}^{\prime}=-c_{i+1} k_{2}, y_{i}^{\prime} & =y_{i}-y_{i-1} k_{1}-y_{i+1} k_{2}
\end{align*}
$$

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 $b_{i}^{\prime \prime} x_{i}=y_{i}^{\prime \prime}$ 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.

## 4 Numerical 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.

### 4.1 Numerical Convergence

The numerical experiments in this section were performed in MATLAB version R2020a with implementations of Algorithm 7 (blocked SICE-SM). The expression A = gallery('randsvd' , $\mathrm{n},-$ cond) was used to generate symmetric test matrices with a prescribed condition number
from random eigenvectors and geometrically distributed eigenvalues in range ( $\left.1, \frac{1}{\text { cond }}\right)$. The input matrix is first converted to single precision and subsequently tridiagonalized using $[\mathrm{Q}, \mathrm{T}]=$ hess (A) function in single precision. Then converted back to double precision for finding the eigenvalues and eigenvectors using expression $[\mathrm{V}, \mathrm{D}]=\mathrm{eig}(\mathrm{A})$. The eigenvectors in D and column eigenvectors in $\mathrm{Q} * \mathrm{~V}$ will be used as the starting point of our refinement algorithms.

Figure 2.2 shows the convergence of Algorithm 7: 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 $\|A x-\lambda x\|_{\infty}$ 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.

### 4.2 Performance Results

The system we are using has two sockets of $\operatorname{Intel}(\mathrm{R}) \mathrm{Xeon}(\mathrm{R}) \mathrm{CPU}$ 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 $7.8 \mathrm{TFLOP} / \mathrm{s}$ 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 $A \equiv\left[a_{i j}\right]$ was generated with random elements from a uniform distribution in range $(0,1): a_{i j} \sim \mathcal{U}(0,1)$ and $a_{i j}=a_{j i}$. 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 Figure 2.3. 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


Figure 2.2: Blocked SICE-SM convergence of a $100 \times 100$ matrix with geometrically distributed eigenvalues from 1 (blue) to $10^{-7}$ (red).
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 2.4 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 $1.7 \times$ faster than double precision and the mixed precision is about $1.3 \times$ faster. The dashed lines represent the two-stage algorithm. They are at least $2 \times$ faster than their corresponding single stage algorithm in general. The performance improvement over double precision is about $1.2 \times$. Figure 2.5 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 $2.44 \times$ faster than complex double and mixed precision solver is $1.45 \times$

Figure 2.6 and 2.7 shows the performance when requesting different numbers of eigenpairs with the input matrix size fixed at $n=20000$ of both real symmetric and complex Hermitian matrices. Fixed precision performance is not changing much as the number of requested eigenpairs increase. Implementation of the second stage back transformation on CPU is more optimized for certain sizes, causing some cases that less eigenpairs can be slightly slower then more eigenpairs. Mixed precision is noticeably faster than double precision if 128 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.

Figure 2.8 shows the detailed profile for matrix size $n=16000$ and 16 eigenvalues/eigenvectors requested. The details of computational components were explained in Section 3.3. 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


Figure 2.3: PLASMA execution times and their breakdowns for matrix of size $n=10000$ and with 32 eigenpairs requested.


Figure 2.4: 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 2.5: 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.


Figure 2.6: 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 $n=20000$.


Figure 2.7: Performance comparison of complex single, complex double, and complex mixed precision solvers on top of MAGMA on NVIDIA V100 GPU with varying number of requested eigenpairs and fixed matrix size $n=20000$.
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. Figure 2.9 gives us another view that separates CPU and GPU routines. The matrix size size $n=20000$ and 32 largest eigenpairs requested. For the bottom and middle rows, the single and double fixed precision implementations from MAGMA cannot overlap any CPU and GPU tasks. For the mixed precision implementation at the TOP row, we have an extra blue block from casting the input matrix from double to single. Then the back transformation of first stage on GPU is overlapping with the second stage band to tridiagonal on CPU. And the back transformation of second stage is overlapping with tridiagonal eigensolver on CPU. Without this overlapping, the performance would be slower than fixed double precision which is the case of the implementation in PLASMA.

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 2.10 shows the performance with different matrix sizes on GTX1060 when requesting the largest 32 eigenpairs. The performance of single precision is about $8 \times$ better than that of double precision and the mixed precision with refinement is about $2 \times$ better than double precision. Figure 2.11 is the complex Hermitian solver and the the speed up over complex double is $3.6 \times$ as the complex routines are more compute intense. In Figure 2.12 and 2.13 we show performance results when the matrix size was fixed at $n=10000$ but with varied number of requested eigenpairs for both real symmetric and complex Hermitian matrices. The mixed precision solver is still faster than double precision when 256 eigenpairs are


Figure 2.8: Breakdown of timings of two-stage eigensolvers based on MAGMA on the NVIDIA V100 GPU with size $n=16000$ and 16 largest eigenpairs requested.


Figure 2.9: Profiling of two-stage eigensolvers based on MAGMA on the NVIDIA V100 GPU with size $n=20000$ and 32 largest eigenpairs requested.
requested, but the time in iterative refinement became significant if more eigenvalues and eigenvectors were requested.

Figure 2.14 shows the profiling results with timing breakdown for matrix 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 firststage reduction in single precision and does not suffer from the same penalty. The backtransformation 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.

## 5 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 Sher-man-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 twostage 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 eigenvalues and corresponding eigenvectors


Figure 2.10: Performance of single, double, and mixed precision solvers for real symmetric matrix based on MAGMA two-stage algorithm on the NVIDIA GTX1060 GPU.


Figure 2.11: Performance of single, double, and mixed precision solvers for complex Hermitian matrix based on MAGMA two-stage algorithm on the NVIDIA GTX1060 GPU.


Figure 2.12: 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 $n=10000$.


Figure 2.13: Performance comparison of complex single, complex double, and complex mixed precision solvers on top of MAGMA on NVIDIA GTX1060 GPU with varying number of requested eigenpairs and fixed matrix size $n=10000$.


Figure 2.14: Breakdown of timings of two-stage eigensolvers based on MAGMA on the NVIDIA GTX1060 GPU with size $n=12000$ and 32 largest eigenpairs requested.
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.

## Bibliography

[1] Agullo, E., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Langou, J., Ltaief, H., Luszczek, P., and YarKhan, A. (2009). Plasma users guide. Technical report, Technical report, ICL, UTK. 36, 47
[2] Alefeld, G. and Spreuer, H. (1986). Iterative improvement of componentwise errorbounds for invariant subspaces belonging to a double or nearly double eigenvalue. Computing, 36(4):321-334. 31
[3] AMD AOCL (2020). Amd optimizing cpu libraries (aocl). https://developer.amd. com/amd-aocl/. [Online; accessed 2-August-2020]. 36
[4] AMD rocBLAS (2020). rocblas: Amd's library for blas on rocm. https://github.com/ ROCmSoftwarePlatform/rocBLAS. [Online; accessed 2-August-2020]. 36
[5] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., et al. (1999). LAPACK Users' guide. SIAM. 4, 8, 36, 43
[6] Anderson, E. and Dongarra, J. J. (1990). Evaluating block algorithm variants in LAPACK. Citeseer. 9
[7] ARM Performance Libraries (2020). Arm performance libraries. https://developer. arm.com/tools-and-software/server-and-hpc/compile/arm-compiler-for-linux/ arm-performance-libraries. [Online; accessed 2-August-2020]. 36
[8] Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., and Tomov, S. (2009). Accelerating scientific computations with mixed precision algorithms. Computer Physics Communications, 180(12):2526-2533. 5
[9] bfloat16 (2017). Google brain floating-point format bfloat16. https://cloud.google. com/tpu/docs/bfloat16. [Online; accessed 15-August-2020]. 7
[10] Bischof, C. H., Lang, B., and Sun, X. (2000). A framework for symmetric band reduction. ACM Transactions on Mathematical Software (TOMS), 26(4):581-601. 33
[11] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., et al. (1997). ScaLAPACK users' guide. SIAM. 36
[12] Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., and Tomov, S. (2008). Using mixed precision for sparse matrix computations to enhance the performance while achieving 64bit accuracy. ACM Transactions on Mathematical Software (TOMS), 34(4):1-22. 5
[13] Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., and Kurzak, J. (2007). Mixed precision iterative refinement techniques for the solution of dense linear systems. The International Journal of High Performance Computing Applications, 21(4):457-466. 4
[14] Carson, E. and Higham, N. J. (2017). A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. SIAM Journal on Scientific Computing, 39(6):A2834-A2856. 5, 11, 31
[15] Carson, E. and Higham, N. J. (2018). Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM Journal on Scientific Computing, 40(2):A817-A847. 5, 11, 12, 31
[16] Chang, L.-W., Stratton, J. A., Kim, H.-S., and Hwu, W.-M. W. (2012). 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, pages 1-11. IEEE. 50
[17] Chatelin, F. (1984). Simultaneous newton's iteration for the eigenproblem. In Defect correction methods, pages 67-74. Springer. 31
[18] Cuppen, J. J. (1980). A divide and conquer method for the symmetric tridiagonal eigenproblem. Numerische Mathematik, 36(2):177-195. 35
[19] Davidson, A., Zhang, Y., and Owens, J. D. (2011). An auto-tuned method for solving large tridiagonal systems on the gpu. In 2011 IEEE International Parallel $\mathcal{E}$ Distributed Processing Symposium, pages 956-965. IEEE. 50
[20] Demmel, J. W. (1987). Three methods for refining estimates of invariant subspaces. Computing, 38(1):43-57. 31
[21] Dongarra, J., Hammarling, S., Higham, N. J., Relton, S. D., Valero-Lara, P., and Zounon, M. (2017). The design and performance of batched blas on modern highperformance computing systems. Procedia Computer Science, 108:495-504. 50
[22] Dongarra, J. J. (1980). Improving the accuracy of computed matrix eigenvalues. Technical report, Argonne National Lab., IL (USA); Argonne National Lab.(ANL), Argonne, IL .... 31, 37, 41
[23] Dongarra, J. J. (1982). Algorithm 589: Sicedr: A fortran subroutine for improving the accuracy of computed matrix eigenvalues. ACM Transactions on Mathematical Software (TOMS), 8(4):371-375. 31, 37
[24] Dongarra, J. J., Moler, C. B., and Wilkinson, J. H. (1983). Improving the accuracy of computed eigenvalues and eigenvectors. SIAM Journal on Numerical Analysis, 20(1):2345. 31, 37
[25] Dongarra, J. J., Sorensen, D. C., and Hammarling, S. J. (1989). Block reduction of matrices to condensed forms for eigenvalue computations. Journal of Computational and Applied Mathematics, 27(1-2):215-227. 33
[26] Gates, M., Kurzak, J., Charara, A., YarKhan, A., and Dongarra, J. (2019). 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, pages 1-18. 37
[27] Goto, K. and Van De Geijn, R. (2008). High-performance implementation of the level-3 blas. ACM Transactions on Mathematical Software (TOMS), 35(1):1-14. 36
[28] Gustavson, F. G. (1997). Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development, 41(6):737-755. 9
[29] Haidar, A., Ltaief, H., and Dongarra, J. (2011). Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memoryaware kernels. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-11. 31, 33
[30] Haidar, A., Tomov, S., Dongarra, J., and Higham, N. J. (2018). 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, pages 603-613. IEEE. 5, 31
[31] Haidar, A., Tomov, S., Dongarra, J., Solcà, R., and Schulthess, T. (2014). A novel hybrid cpu-gpu generalized eigensolver for electronic structure calculations based on finegrained memory aware tasks. The International journal of high performance computing applications, 28(2):196-209. 31, 33
[32] High-Performance Linpack Benchmark (2018). Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. https://www. netlib.org/benchmark/hpl/. [Online; accessed 18-August-2020]. 19
[33] Higham, N. J. (1997). Iterative refinement for linear systems and lapack. IMA Journal of Numerical Analysis, 17(4):495-509. 4
[34] Higham, N. J., Pranesh, S., and Zounon, M. (2019). Squeezing a matrix into half precision, with an application to solving linear systems. SIAM Journal on Scientific Computing, 41(4):A2536-A2551. 5
[35] HPL-AI benchmark (2020). Hpl-ai mixed-precision benchmark. https://www.icl. utk. edu/hpl-ai. [Online; accessed 18-August-2020]. 19
[36] IBM ESSL (2020). Ibm engineering and scientific subroutine library (essl) version 6.3. https://www.ibm.com/support/knowledgecenter/SSFHY8_6.3/navigation/ welcome.html. [Online; accessed 2-August-2020]. 36
[37] IEEE (1985). Ieee standard for binary floating-point arithmetic. ANSI/IEEE Std 7541985, pages $1-20.5$
[38] IEEE (2008). Ieee standard for floating-point arithmetic. IEEE Std 754-2008, pages 1-70. 7
[39] Instruments, T. (2003). Tms320c64x dsp library programmer's reference. Literature Number: SPRU565B. 12
[40] Intel MKL (2020). Intel math kernel library. https://software.intel.com/content/ www/us/en/develop/tools/math-kernel-library.html. [Online; accessed 2-August2020]. 36
[41] Jankowski, M. and Woźniakowski, H. (1977). Iterative refinement implies numerical stability. BIT Numerical Mathematics, 17(3):303-311. 4
[42] Kurzak, J. and Dongarra, J. (2007). Implementation of mixed precision in solving systems of linear equations on the cell processor. Concurrency and Computation: Practice and Experience, 19(10):1371-1385. 5
[43] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., and Dongarra, J. (2006). Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, pages 50-50. IEEE. 4
[44] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. ACM Transactions on Mathematical Software (TOMS), 5(3):308-323. 35
[45] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. (2017). Mixed precision training. arXiv preprint arXiv:1710.03740. 5
[46] Moler, C. B. (1967). Iterative refinement in floating point. Journal of the ACM (JACM), 14(2):316-321. 4
[47] NVIDIA cuBLAS (2020). Nvidia cublas. https://docs.nvidia.com/cuda/cublas/ index.html. [Online; accessed 2-August-2020]. 36
[48] NVIDIA P100 (2017). Nvidia tesla p100 gpu pascal architecture whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/ volta-architecture-whitepaper.pdf. [Online; accessed 15-August-2020]. 5
[49] NVIDIA V100 (2017). Nvidia tesla v100 gpu volta architecture whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/ volta-architecture-whitepaper.pdf. [Online; accessed 15-August-2020]. 5, 16
[50] Ogita, T. and Aishima, K. (2018). Iterative refinement for symmetric eigenvalue decomposition. Japan Journal of Industrial and Applied Mathematics, 35(3):1007-1035. 32
[51] Ogita, T. and Aishima, K. (2019). Iterative refinement for symmetric eigenvalue decomposition ii: clustered eigenvalues. Japan Journal of Industrial and Applied Mathematics, 36(2):435-459. 32
[52] Ogita, T. and Aishima, K. (2020). Iterative refinement for singular value decomposition based on matrix multiplication. Journal of Computational and Applied Mathematics, 369:112512. 32
[53] OpenBLAS (2020). Openblas. https://github.com/xianyi/OpenBLAS. [Online; accessed 2-August-2020]. 36
[54] Parlett, B. N. and Dhillon, I. S. (2000). Relatively robust representations of symmetric tridiagonals. Linear Algebra and its applications, 309(1-3):121-151. 35
[55] Peters, G. and Wilkinson, J. H. (1979). Inverse iteration, ill-conditioned equations and newton's method. SIAM review, 21(3):339-360. 31
[56] Prikopa, K. E. and Gansterer, W. N. (2013). On mixed precision iterative refinement for eigenvalue problems. Procedia Computer Science, 18:2647-2650. 32, 43
[57] Pytorch (2020). Pytorch 1.4.0 documentation: Quantization. https://pytorch.org/ docs/stable/quantization.html. [Online; accessed 14-August-2020]. 3
[58] Saad, Y. (1993). A flexible inner-outer preconditioned gmres algorithm. SIAM Journal on Scientific Computing, 14(2):461-469. 4, 11
[59] Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on scientific and statistical computing, 7(3):856-869. 4
[60] Sherman, J. and Morrison, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. The Annals of Mathematical Statistics, 21(1):124-127. 41
[61] Smith, B. T., Boyle, J. M., Garbow, B., Ikebe, Y., Klema, V., and Moler, C. (2013). Matrix eigensystem routines-EISPACK guide, volume 6. Springer. 35
[62] Stewart, G. W. (1973). Error and perturbation bounds for subspaces associated with certain eigenvalue problems. SIAM review, 15(4):727-764. 31
[63] Symm, H. and Wilkinson, J. H. (1980). Realistic error bounds for a simple eigenvalue and its associated eigenvector. Numerische Mathematik, 35(2):113-126. 31
[64] TensorFlow (2020). Tensorflow lite 8-bit quantization specification. https://www. tensorflow.org/lite/performance/quantization_spec. [Online; accessed 14-August2020]. 3
[65] Tisseur, F. (2001). Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. SIAM Journal on Matrix Analysis and Applications, 22(4):1038-1057. 31
[66] Tomov, S., Dongarra, J., and Baboulin, M. (2010). Towards dense linear algebra for hybrid gpu accelerated manycore systems. Parallel Computing, 36(5-6):232-240. 36, 47
[67] Van Zee, F. G. and Van De Geijn, R. A. (2015). Blis: A framework for rapidly instantiating blas functionality. ACM Transactions on Mathematical Software (TOMS), 41(3):1-33. 36
[68] Wilkinson, J. (1962). Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection. Numerische Mathematik, 4(1):362-367. 35
[69] Wilkinson, J. (1963). Rounding errors in algebraic processes. 4
[70] Wilkinson, J. H. (1968). Global convergene of tridiagonal qr algorithm with origin shifts. Linear Algebra and its Applications, 1(3):409-420. 33
[71] Yamamoto, T. (1980). Error bounds for computed eigenvalues and eigenvectors. Numerische Mathematik, 34(2):189-199. 31
[72] Zhang, Y., Cohen, J., and Owens, J. D. (2010). Fast tridiagonal solvers on the gpu. ACM Sigplan Notices, 45(5):127-136. 50

## Vita

Yaohung "Mike" Tsai was born in Hsinchu, Taiwan, on February 27, 1986. He received his Bachelor of Science in Mathematics and Master of Science in Mathematics from National Taiwan University, Taipei, Taiwan in 2009 and 2012, respectively. In January 2015, he was enrolled in the University of Tennessee as a doctoral student in Computer Science and worked as a graduated research assistant in Innovative Computing Laboratory (ICL). His is involved and contributed to various projects, including recursive LU, QR, and Cholesky factorization in LAPACK, RaPyDLI project for deep learning performance optimization, BEAST project for GPU kernel performance autotuning, STALE and xSDK project for mixed-precision numerical linear algebra algorithms. His current research interests include high performance computing, numerical linear algebra, performance optimization, and mixed-precision algorithms. Yaohung Tsai is expected to receive a Doctor of Philosophy degree in Computer Science in December 2020.


[^0]:    Algorithm 10 Two stages symmetric eigensolver
    1: First stage symmetric to band via Householder transformations.
    2: Second stage band to tridiagonal via bulge chasing.
    3: Tridiagonal symmetric eigensolver (divide and conquer).
    4: back transformation for second stage on eigenvectors.
    5: back transformation for first stage on eigenvectors.

