Reducing Communication in the Solution of Linear Systems

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Neil Lindquist

August 2023

I thank Jack Dongarra and Piotr Luszczek for their excellent mentorship during my time at the University of Tennessee. I also thank Mike Heroux for the introduction to the world of numerical and high-performance computing and to subsequent guidance. Additionally, I thank Mike Berry for taking the time to serve on my committee.

Furthermore, I want to thank Mark Gates, Sébastien Cayrols, and the rest of the SLATE team for all the helpful and interesting conversations, as well as the entire Innovative Computing Laboratory. Finally, I thank my family and friends for supporting me in this endeavor, even when they didn't really understand what I was working on.

This research was financially supported by the National Science Foundation under grant no. 2004541, UT Battelle under subaward no. 4000123266, and the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and National Nuclear Security Administration. Additionally, this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC05-00OR22725.

Abstract

There is a growing performance gap between computation and communication on modern computers, making it crucial to develop algorithms with lower latency and bandwidth requirements. Because systems of linear equations are important for numerous scientific and engineering applications, I have studied several approaches for reducing communication in those problems. First, I developed optimizations to dense LU with partial pivoting, which downstream applications can adopt with little to no effort. Second, I consider two techniques to completely replace pivoting in dense LU, which can provide significantly higher speedups, albeit without the same numerical guarantees as partial pivoting. One technique uses randomized preprocessing, while the other is a novel combination of block factorization and additive perturbation. Finally, I investigate using mixed precision in GMRES for solving sparse systems, which reduces the volume of data movement, and thus, the pressure on the memory bandwidth.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction[1]

Large systems of linear equations arise in many scientific and engineering applications [31, 34, 50, 88]. The scale of these problems strains even leadership-class supercomputers, with dense problems containing millions of unknowns [34, 88], sparse problems containing tens of billions of unknowns [31, 50], and some applications desiring the ability to solve even larger problems. Thus, it is crucial to design and implement algorithms to solve these problems effectively on modern clusters and supercomputers.

Unfortunately, there is a growing imbalance in performance between data movement and arithmetic in modern hardware. For example, the new exascale Frontier supercomputer at the Oak Ridge National Laboratory can do more than 100 double-precision floating-point operations for every operand fetched from memory [85]. Future systems are expected to be even more imbalanced, as suggested by the trend in top supercomputers shown by Fig. 1.1. Additionally, the latency to access the main memory improves even slower than the bandwidth [99]. Thus, saturating the memory bandwidth requires speculative out-of-order execution that is expensive in terms of the transistor and energy budgets [59] and exposes new security vulnerabilities [84]. In distributed settings, communication between nodes is even more expensive. Thus, developing techniques to reduce the amount of communication and data movement is increasingly important, even in algorithms that have historically been compute-bound.

Because different algorithms provide significantly different performance characteristics, I consider only a few key algorithms. First, I consider direct methods for dense, non-symmetric systems, which are also often used for symmetric-indefinite systems. The primary workhorse for these problems is Gaussian elimination with partial pivoting (GEPP), with

[1] Section 1.3 of this chapter reuses material from four of my published papers [92] (© 2020 Springer), [94] (© 2021 IEEE), [91] (© 2022 IEEE), and [95]. Coauthors include Piotr Luszczek, Jack Dongarra, and Mark Gates. Reused coauthor contributions are limited to textual improvements.

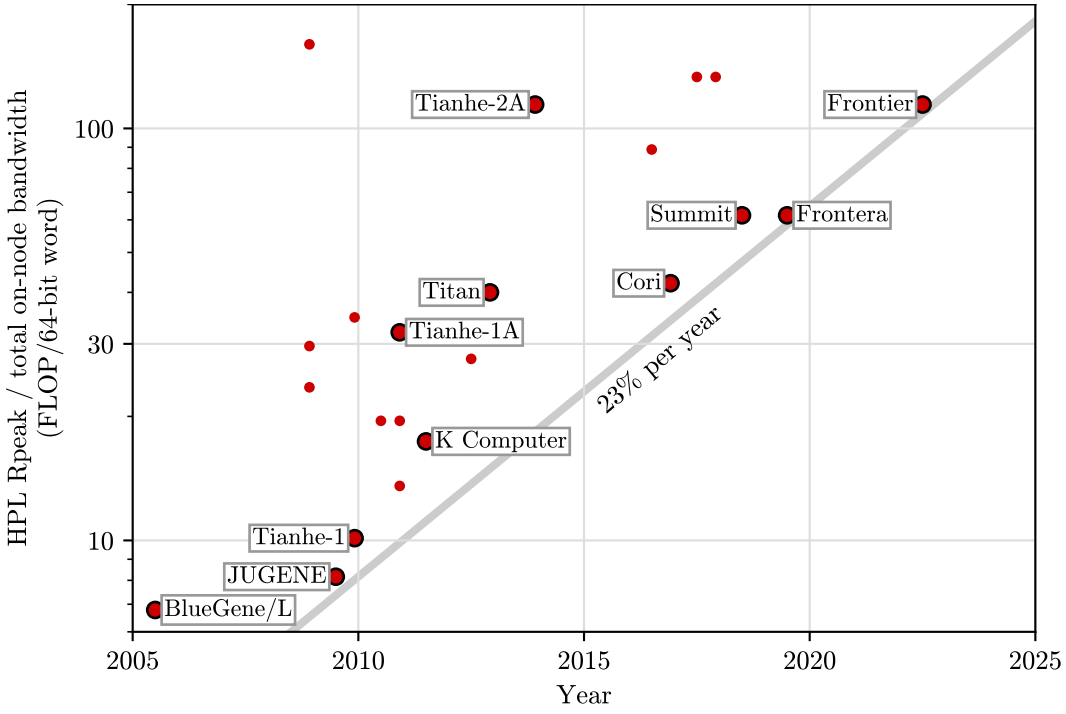


Figure 1.1: The machine balance of top supercomputers in recent years [83, 85].

the pivoting introducing a large amount of data movement. My work to improve dense, direct solvers consists of two parts. The first part is techniques to improve the efficiency of pivoting, including GPU-aware MPI and threshold pivoting. The second part is algorithmic variants that completely remove pivoting, including randomized preprocessing and additive modifications. Second, I consider iterative methods for sparse, non-symmetric systems. The generalized minimal residual method (GMRES) is a major method for this class of problems and also sees use in both dense [75] and symmetric [50] settings.

1.1 Notation

Most variable names follow Householder's convention: matrices are capital letters; vectors are lowercase Latin letters; scalars are lowercase Greek letters; and indices are lowercase Latin letters between i and p. A notable exception is u, the floating-point unit roundoff.

Generally, the variables A, x, and b correspond to the standard linear system, $Ax = b$. The real numbers are denoted $\mathbb{R}$, with $\mathbb{R}^n$ and $\mathbb{R}^{m \times n}$ being the corresponding n-dimensional column vectors and $m \times n$-dimensional matrices. The singular values of an $n \times n$ matrix A are denoted $\sigma_1(A) \geq \sigma_2(A) \geq \cdots \geq \sigma_n(A)$. Transposition and Hermitian-transposition of a matrix A are denoted A^T and A^H, respectively. For blocked and tiled algorithms, n_b represents the block or tile size, while n_t represents the number of blocks or tiles. All indices are 1-based and inclusive.

1.2 Dense LU experiments: SLATE and Summit

My research on dense LU factorization focuses on the Software for Linear Algebra Targeting Exascale (SLATE) library. SLATE is a dense linear algebra library targeting distributed, heterogeneous systems [53]. It is being developed as a successor to ScaLAPACK at the University of Tennessee as a part of the Exascale Computing Project. SLATE uses OpenMP tasking for node-level parallelism and MPI for inter-node parallelism.

I primarily tested SLATE on the Summit supercomputer at the Oak Ridge National Laboratory. The Summit supercomputer contains more than 4000 IBM Power System AC922 nodes, with each node containing two 22-core IBM POWER9 CPUs and six NVIDIA Volta V100 GPUs. Each socket has one CPU core reserved for system overheads, such as the operating system (Red Hat Enterprise Linux version 8.2). The GPUs provide most of the computational power, with 7.8 TFLOP/s, 16 GiB of high bandwidth memory (HBM2), and 900 GB/s memory bandwidth per GPU. The CPUs for a given socket provide 540 GFLOP/s,

256 GiB DDR4 memory, and 170 GB/s memory bandwidth (excluding the reserved core). NVIDIA NVLink provides a bidirectional 50 GB/s within a socket. A dual-rail EDR InfiniBand network connects the nodes with a non-blocking fat-tree topology. See the Summit user guide for more details.[2] In tests on Summit, the job launcher was always configured to assign all 21 cores and 3 GPUs of a socket to each process; threads were bound and distributed with `packed:21` and `packed`, respectively.

1.3 Literature Review

Because of the importance of communication in high-performance linear algebra codes and their performance, the entire literature on the subject is quite extensive. To keep the size of this review manageable, I limited its scope to pivoting schemes for dense LU factorization, approaches to replace pivoting in dense LU factorization, and mixed-precision GMRES. Other notable topics in the subject include 2.5D and 3D matrix algorithms [87], s-step methods [36], and low synchronization schemes for orthogonalization [26].

1.3.1 Pivoting in LU Factorization

Most high-performance implementations of dense LU factorization ensure numerical stability using partial pivoting. This scheme swaps rows during the factorization so that when a column is factored, its diagonal element is at least as large as each of the sub-diagonal elements. While examples with catastrophic errors have been known since Wilkinson's work in the 1960s [133], such problems are rare in practice [73, 125]. Because partial pivoting is usually robust, more expensive techniques, such as complete pivoting and rook pivoting, are rarely used for dense problems. Unfortunately, pivoting introduces several overheads. First, applying the pivots requires exchanging entire rows of the matrix across the network. Second, computing the pivots requires doing a latency-heavy reduction for each column of the matrix. Finally, the data dependencies introduced by pivoting reduce the available parallelism; computing the pivots is interspersed with the left-panel triangular solve, while the top-panel triangular solve is delayed until pivots have been applied to the corresponding columns.

The simplest way to avoid the cost of pivoting is to simply omit it, i.e., to use Gaussian elimination with no pivoting (GENP). Unfortunately, this is only numerically stable for specific classes of matrices, such as those that are diagonally dominant, totally nonnegative, or symmetric positive definite [71]. In general, the resulting error may be arbitrarily large due

[2] https://docs.olcf.ornl.gov/systems/summit_user_guide.html

to unchecked growth; thus, other techniques are needed. (See Section 3.1 for more discussion on element growth and its effect on accuracy.)

Various alternative pivoting schemes have been previously proposed to reduce the communication. The most well-known of these is Gaussian elimination with tournament pivoting (GETP) [65, 66]; its prominence is demonstrated by how frequently it is referred to as "Communication Avoiding LU" or CALU. Tournament pivoting divides the matrix into a 2d tiled layout and computes all of the pivots for a given block column in a single reduction. To find the pivots of a block column with n_b columns, the tiles are reduced in a binary fashion, like a tournament bracket. Each step of the reduction concatenates two tiles vertically, then applies a GEPP to find the best n_b pivot rows. This reduction requires significantly fewer messages than partial pivoting, which does a reduction for each column sequentially. Recently, tournament pivoting has been combined with a 2.5D matrix distribution and dynamic pivoting [54] in the COnfLUX algorithm [87].

Threshold pivoting is another strategy for reducing communication. Technically, it is not a pivoting scheme itself but a modification of other pivoting schemes; however, I have only considered it in conjunction with partial pivoting. For example, partial pivoting ensures that at the kth step,

$$|A[k,k]| \geq |A[j,k]| \qquad j = k, \ldots, n. \tag{1.1}$$

Threshold pivoting relaxes this constraint to

$$|A[k,k]| \geq \tau |A[j,k]| \qquad j = k, \ldots, n \tag{1.2}$$

where $0 \leq \tau \leq 1$. This allows the selection of a pivot that is not maximal in magnitude but is preferable in other regards, such as data movement. Using threshold pivoting in dense factorizations has received limited attention to date. The subject was primarily investigated by Malard in 1991 [98], followed by Hoffman and Potma a few years later [76]. Both works investigated the use of thresholded partial pivoting to reduce inter-process communication in LU factorization on distributed systems and experimentally demonstrated both limited loss of accuracy and significant performance improvements. Unfortunately, both works were limited to $\tau \geq 0.1$ and random matrices of size $n \leq 4096$. Malard also tested dynamic pivoting, which changes the matrix distribution instead of exchanging rows between processes [54]. Dynamic pivoting still must exchange rows for load-balancing, which Malard unsuccessfully tried to improve using threshold pivoting. The only other known uses of threshold pivoting for dense matrices are brief tests of either element growth [125] or accuracy [47]. Unfortunately, these other experiments do not consider performance or matrices larger than $n = 2048$.

In contrast to dense factorizations, threshold pivoting is heavily used in sparse factorizations [43]. The cost of a sparse factorization directly depends on the number of nonzero entries. Thus, it is beneficial to select pivots causing less fill-in even if (1.1) is slightly violated. The literature for sparse factorizations includes a wide range of threshold recommendations from 4^{-1} to 10^{-8}; however, a threshold of 10^{-1} or 10^{-2} is commonly recommended as a general-purpose guideline [43, 77].

Finally, there are other, less popular, techniques. Dynamic pivoting, mentioned above, discards the notion of a fixed matrix distribution [54]. Instead of exchanging rows across the network, the ownership of the rows is exchanged, saving significant network traffic. However, depending on the numerical structure, the trailing matrix may become imbalanced; this requires transferring rows to rebalance the remaining work. Another approach is pairwise pivoting which uses 2×2 transforms that combine pivoting with the elimination of subdiagonal elements [23, 117]. Extending this to a block-wise algorithm gives incremental pivoting, which eliminations one block of the column at a time [30]. Both pairwise and incremental pivoting were initially developed for out-of-core solvers and later adapted for parallel solvers. Their theoretical analysis and experimental results are worse than that of partial pivoting [66, 117, 125], but there is not a strong consensus on whether these schemes are robust enough in practice or should be avoided [41].

There are also several strategies to replace pivoting. Sections 1.3.2 and 1.3.3 go into detail on two such strategies. Another interesting strategy to replace pivoting is a hybrid of the LU and QR factorizations [48]. This algorithm attempts to factor each block column with GENP. It then tests the stability and, if instability is detected, re-factors the block column with the unequivocally stable QR factorization. Thus, in the best case, the factorization progresses as per GENP but provides more robust behavior when GENP struggles. Unfortunately, this has similar parallel dependences to GEPP in task-parallel implementations, which reduces the available parallelism.

1.3.2 Random Butterfly Transforms

Random butterfly transforms (RBTs) are structured sparse matrices with random coefficients that are closely related to the fast Fourier transform (FFT). Multiplying a matrix on both sides by such transforms usually allows LU without pivoting to factor the matrix without significant loss of accuracy. Loosely speaking, it "smears" the matrix's nonsingularity so that the leading principal submatrices are reasonably well conditioned. However, iterative refinement is still often needed to compute a solution accurate to double-precision. See Section 3.3 for a detailed description of this algorithm.

The RBT approach was first proposed in 1995 by a set of tech reports by Parker and Lê [107, 108, 109]. They outline the approach and provide basic theoretical and experimental analysis. They provided basic theoretical analysis showing that, in exact arithmetic, the transformed linear system can be factored without pivoting with a probability of 1. Unfortunately, they were unable to make any conclusions about the resulting element growth or the effects of floating-point arithmetic. They then tested 11 matrices for sizes from $n = 32$ to $n = 512$. The RBT solver provided similar solutions to LINPACK's GEPP, except for the matrices with large condition numbers. Due to fewer optimizations, their implementation performed worse than LINPACK.

The RBT idea was then developed for actual performance in a series of papers by Baboulin et al. The primary thrust of their work targets many-core and heterogeneous systems for both non-symmetric and symmetric-indefinite problems [16, 17, 19, 21, 24, 123]. This work included a number of modifications to improve performance, including

- the use of real-valued transforms instead of complex-valued ones,
- truncation of the transform recursion to a depth of 2,
- the use of iterative refinement, and
- the design of efficient RBT kernels.

Additionally, they explored the use of the RBT strategy for a distributed, symmetric-indefinite solver [14], for sparse factorization [20], and for incomplete sparse factorization [18].

Recent work by Shen et al. combines RBTs with a modified version of the adaptive cross approximation (ACA) algorithm [114]. The ACA algorithm takes advantage of low-rank properties in the matrix's off-diagonal to factor a dense matrix in $\mathcal{O}(n \log(n))$ time. Unfortunately, their tests are limited to matrices that can be factored without pivoting and they do not test the performance of a non-RBT, non-pivoted factorization (with or without ACA), limiting the strength of their conclusions.

Additionally, there has been recent work on the theoretical properties of RBTs [110, 111, 126]. Unfortunately, these works have focused on butterflies based on rotation matrices instead of the butterflies based on block-Hadamard matrices which have been used in performance-oriented works.

Butterfly matrices are not the only transform that has been proposed for replacing pivoting. Pan et al. have explored a number of transforms, including ones based on Gaussian matrices, circulant matrices, and subsampled random Fourier transforms (SRFTs) [103, 105]. Additionally, it has been proven that a matrix preconditioned with one or more Gaussian matrices can be factored with GENP and have a high probability of a limited growth factor. Furthermore, experimental results for various preconditioning matrices show numerical errors of 10^{-10} to 10^{-15} with GENP, one step of iterative refinement, and matrices of size

$n = 256$ to $n = 4096$. A related idea is to use additive preprocessing instead of multiplicative preprocessing [104].

1.3.3 Additive Modifications in LU Factorization

Another alternative to pivoting is perturbing the matrix to one with better numerical properties. Replacing pivoting with additive modifications was first suggested by Stewart for sparse matrices [119]. There, small diagonal elements were modified before factoring the corresponding column. When solving the system, the modifications were accounted for using the Sherman-Morrison formula in a recursive fashion. Unfortunately, this approach has seen limited use outside the optimization community [140]. An interesting variant of this approach is to apply additive modifications, then correct them by adding extra rows and columns to the matrix instead of using the Woodbury formula [8]. That is, the system $Ax = b$ is replaced with

$$\begin{bmatrix} A + F_1F_2 & F_2 \\ F_1 & I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

where F_1F_2 modifies the appropriate diagonal elements. This variant is closely related to the derivation of the Woodbury formula via the Schur complement [70]. Unfortunately, this idea has not been explored deeply as the focus was on avoiding fill-in for sparse, symmetric positive definite matrices with a few dense rows. Additionally, it complicates the matrix allocation because the dimension of the matrix grows with the number of perturbations.

Relatedly, static pivoting applies additive modifications without any form of the Woodbury formula and instead relies on iterative refinement to correct the perturbation [90]. As the name implies, static pivoting also pre-permutes the matrix, trying to place large elements on the diagonal before the factorization begins [44]. While iterative refinement can recover minor errors, large errors can slow or even prevent convergence, especially ill-conditioned matrices. Thus, there is a trade-off in the factorization's backward error between the direct perturbations to the matrix and the error induced by the growth factor.

1.3.4 Mixed-Precision Krylov Methods

The idea of using multiple precisions to improve performance, especially for systems of linear equations, has been long known in the linear algebra community, going back to Wilkinson's work in the 1960s [133]. Recent years have seen an increase in the popularity of such methods [1]. This has been driven, in part, by the increase in performance of reduced precision arithmetic, especially the 16-bit formats used in deep learning.

One approach to using multiple precisions in GMRES is to store the preconditioner in reduced precision and keep the rest of the computation in full precision [56]. The approximate nature of the preconditioner means that reducing the floating-point precision has a limited reduction in quality. However, the reduced precision round-off errors can make linear preconditioners act nonlinearly in full precision, making flexible GMRES (FGMRES) necessary. One interesting version of this is to precondition a full-precision GMRES with a reduced-precision GMRES (possibly with a preconditioner of its own) [15]. Reduced precision preconditioners for GMRES have recently become popular with the development of the GMRES-IR algorithm, which preconditions a restarted GMRES with an LU factorization computed in 16-bit precision [32].

A similar idea is to use a low-precision GMRES within mixed-precision iterative refinement [11, 12, 97, 128, 136, 139]. Such a scheme computes only the residual and solution in high precision. Unfortunately, this approach requires storing two copies of the matrix: one in low precision that is accessed frequently and one in high precision that is accessed infrequently. Recently, the GMRES-IR framework has been expanded in a way that includes this strategy [6]. Chapter 4 explores this approach, including justification of the division of precision and strategies to determine when to start the next outer iteration. Compared to previous works, my work puts more emphasis on the mixed-precision GMRES as a unified solver compared to the previous work, instead of merely viewing it as mixed-precision iterative refinement with a new inner-solver. The most notable manifestation of this perspective is my effort in finding effective restart strategies; other authors have limited themselves to restart strategies based on only the inner iteration count. Note that other authors have built on the work in Chapter 4 as they explored this strategy [97, 136, 139], resulting in a fuzzy definition of "previous work".

There have also been mixed-precision variants of GMRES based on reducing the precision of just the Krylov basis storage. The first reduces the precision of just the search space in FGMRES [3]. This has a limited effect on the accuracy because the inexactness of the preconditioner subsumes the added round-off error. A related variant for non-flexible GMRES is compressed basis GMRES (CB-GMRES) [4]. CB-GMRES successfully provided a speedup with the 32-bit floating-point version providing the best median speedup at 1.4 times. However, the scheme requires custom, high-performance, mixed-precision kernels, which increases the cost of implementation due to the limited availability of existing mixed-precision routines. This approach fits within the broader memory accessor paradigm promoted by the Ginkgo library [68].

There has recently been promising theoretical work for varying the working precision as the iteration progresses in a non-restarted GMRES. Notably, part of this work shows

that computing the Arnoldi process in finite precision still allows GMRES to converge at approximately the same rate as exact-precision GMRES [61]. There exists further theoretical work on reducing the accuracy of just the matrix-vector products in GMRES, and other Krylov solvers, as the iterations progress [28, 116, 129].

Finally, GMRES using integer arithmetic has been explored [80]. While integer GMRES did not involve a reduction in data movement, it does show GMRES achieving a full-precision solution with limited iteration overhead when the solver uses an alternative data format. Relatedly, there has been some work into storing parts of the solver in non-floating-point formats, particularly in FGMRES and CB-GMRES [3, 4]

Mixed precision approaches have also been used for other iterative solvers. Like GMRES, mixed precision approaches include a reduced precision preconditioner [29, 46] and using a reduced precision solver inside iterative refinement [11]. However, with Krylov methods, iterative refinement discards the subspace at each restart. So, the strategy of "reliable updates" has been proposed, which retains information about the Krylov subspace across restarts [37, 120]. Finally, there has been some experimental usage of alternate data representations, such as data compression, in iterative solvers [5, 10, 89].

Chapter 2

Reducing Pivoting Overheads[1]

Because of the long history of GEPP, application developers will be wary to use other methods until analysis and experimentation have proven they are reliable. So, optimizing partial pivoting is still important, even in light of the techniques in Chapter 3. I have considered two main strategies: optimizations for permuting the trailing matrix and threshold pivoting. The former are direct software optimizations, while the latter introduces a controlled relaxation of the pivoting constraints. These strategies can be combined, although they address the same overhead (the cost to exchange rows).

2.1 Efficiently Permuting the Trailing Matrix

One of the overheads in partial pivoting is applying the permutation to the trailing matrices. This is communication intensive in distributed memory environments and cannot be elided from the critical path. Furthermore, techniques such as tournament pivoting do not address these overheads since they still must pivot for most matrices. Because of its importance I have investigated three optimizations for SLATE's `permuteRows` routine: consolidating inter-node communication to reduce latency costs, task-level parallelism, and GPU-aware MPI. All three of these optimizations are part of the current version of SLATE.

The initial version of SLATE's `permuteRows` was implemented similarly to Netlib's ScaLAPACK with each swap being applied separately. Algorithm 2.1 shows this approach for a single block column. The main difference was that SLATE completely pivoted one block column before starting the next, while ScaLAPACK treats the entire trailing matrix

[1]Section 2.1 reuses material from an unpublished class report for COSC 594 and Section 2.2 reuses material from one of my published papers [91] (© 2022 IEEE). Coauthors for the published paper include Mark Gates, Piotr Luszczek, and Jack Dongarra. Reused coauthor contributions are limited to the idea of threshold pivoting, textual improvements, and improvements to Fig. 2.3.

```
1: C ← the block column
2: n_b ← the number of rows in the top tile
3: if C[1 : n_b, :] is local then
4:     for i = 1, ..., n_b do
5:         p ← pivot_list[i]
6:         if C[p, :] is local then
7:             Swap C[i, :] and C[p, :]
8:         else
9:             Copy C[i, :] to host
10:            Send C[i, :] to and Recv C[p, :] from the owner of C[p, :]
11:            Copy C[p, :] to GPU at memory location C[i, :]
12: else
13:    for i = 1, ..., n_b do
14:        p ← pivot_list[i]
15:        if C[p, :] is local then
16:            Copy C[p, :] to host
17:            Send C[p, :] to and Recv C[i, :] from owner of C[i, :]
18:            Copy C[i, :] to GPU at memory location C[p, :]
```

Algorithm 2.1: SLATE's original strategy for permuting rows in a block column on GPU

as one block. This design is latency bound due to the small, sequential point-to-point communications. Furthermore, when using GPUs, this latency also includes synchronizing the device stream and transferring data between host and device. My strategy gathers all rows touched by the set of pivots onto a single process, does all the swaps locally, then scatters the rows back to their owners. Algorithm 2.2 describes this in more detail; note that R should be stored sparsely since most rows are not involved in the permutation. This results in at most two MPI latencies per remote process and two transfers between host and device. Figure 2.1 illustrates this effect on two processes.

The second optimization was a straightforward addition of task-level parallelism. Each block column can be independently permuted, which provides a natural place to parallelize the routine. Particularly, SLATE's `getrf` routine already assumed that all tiles in the same column belong to the same GPU. Thus, I used a separate task for each device; more tasks are possible but require additional device streams.

The final optimization was the addition of GPU-aware MPI, which allows MPI to directly access GPU memory instead of requiring the user to manually copy the data to the host [132]. This provides a few advantages. First, it allows the same internal host buffer to be shared between the GPU API and MPI, reducing local data movement. Second, some hardware can transfer data from the GPU straight to the network, reducing the amount of data movement within a node. This is especially important on ORNL's new Frontier supercomputer, where the network cards are part of the GPUs; thus, sending data directly from the GPU is cheaper than sending data from the CPU, let alone doing a round trip from GPU to CPU and back.

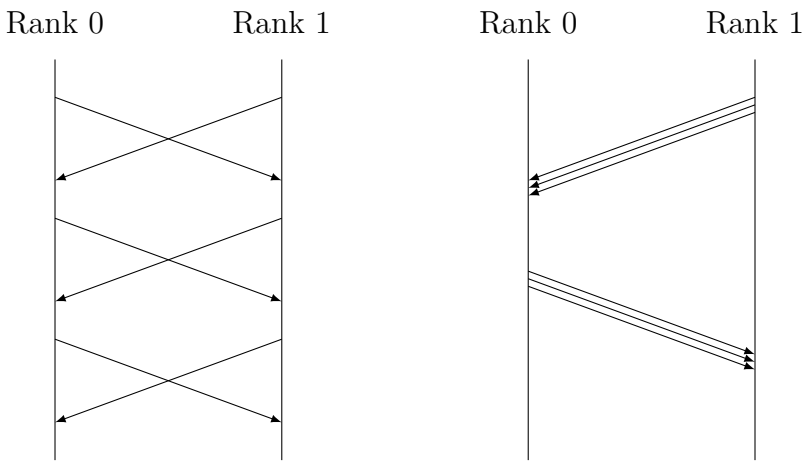
Experimental Results

To demonstrate the benefit of each of these optimizations, I tested them incrementally. These experiments were performed on OLCF's Frontier supercomputer since the GPU-aware MPI provided by Summit does not provide a performance benefit. Because Frontier's vendor MPI (Cray MPICH) does not support GPU-aware MPI with multiple GPUs per process, I could not test the second optimization (creating separate tasks for each GPU). Fortunately, this optimization is the most straightforward of those discussed in this section.

Each node of Frontier contains a single 64-core "Optimized 3rd Gen EPYC" AMD CPU and four AMD MI250X accelerators. Note that each MI250X accelerator contains two graphic compute dies (GCDs), which act as independent GPUs albeit with a higher bandwidth between the pair than with other accelerators. The GPUs provide most of the computational power with 26.5 TFLOP/s and 1.6 TB/s memory bandwidth per GPU. See the Frontier user

```
1: C ← the block column
2: n_b ← rows in the top tile
3: if C[1 : n_b, :] is local then
4:     Receive remote rows from their owners into R
5:     Copy R to GPU
6:     for i = 1, ..., n_b do
7:         p ← pivot_list[i]
8:         if C[p, :] is local then
9:             Swap C[i, :] and C[p, :]
10:        else
11:            Swap C[i, :] and R[p, :]
12:    Copy R to host
13:    Send remote rows to their owners from R
14: else
15:    for i = 1, ..., n_b do
16:        p ← pivot_list[i]
17:        if C[p, :] is local then
18:            Copy C[p, :] to R[p, :]
19:    Copy R to host
20:    Send R to the owner of C[0 : n_b, :]
21:    Receive R from the owner of C[0 : n_b, :]
22:    Copy R to GPU
23:    for i = 1, ..., n_b do
24:        p ← pivot_list[i]
25:        if C[p, :] is local then
26:            Copy R[p, :] to C[p, :]
```

Algorithm 2.2: Optimized strategy for permuting rows in a block column on GPU



(a) Sequential swapping **(b)** Combined swapping

Figure 2.1: Communication patterns for applying a series of pivots on two ranks.

guide for more details.[2] The code was compiled using version 15.0.0 of the Cray compiler, ROCm 5.3.0, Cray LibSci 22.12.1.1, and Cray MPICH 8.1.23. Eight cores were reserved for the OS (SUSE Linux Enterprise Server version 15.4). SLATE was configured with `-check n -ref n -origin d -target d -nb 640 -ib 32 -lookahead 1 -panel-threads 1`. The default values were used for the precision (double), the process grid (8×8), and the number of right-hand sides (10). Random matrices (`rand`) with fixed seeds (1 and 2, respectively) were used for both the matrix and right-hand side (respectively).

Figure 2.2 shows the effect of the optimizations on the performance to factor a matrix. Because it better shows the benefit of GPU-aware MPI in `permuteRows`, I tested the performance of SLATE's Gaussian elimination with tournament pivoting (GETP). Each configuration was run 3 times and plotted with 99 % confidence intervals. As shown by the plot, coalescing the communication improves the performance more than threefold. GPU-aware MPI also provides a useful speedup, but only of around 10 %. Remember that these results include the entire cost to factor the matrix, so `permuteRows` itself will have a significantly higher speedup.

2.2 Threshold Pivoting

Threshold pivoting is designed around relaxing the constraint on valid pivots so that it can choose pivots requiring less communication. Recall that before factoring the ith column, partial pivoting ensures that

$$|A[i,i]| \geq |A[j,i]| \qquad j = i, \ldots, n \tag{2.1}$$

by conditionally swapping the ith row with one below it. Threshold pivoting relaxes this to

$$|A[i,i]| \geq \tau |A[j,i]| \qquad j = i, \ldots, n \tag{2.2}$$

where $0 \leq \tau \leq 1$ is a fixed parameter. This relaxation allows the selection of non-maximal pivot elements that are otherwise preferable. When $\tau = 1$, threshold pivoting is equivalent to partial pivoting. On the other hand, when $\tau = 0$, no numerical pivoting is applied. Threshold pivoting is common in sparse factorizations, primarily to avoid fill-in [43]. I have instead investigated using it to reduce data movement in dense factorizations.

Usually in a dense factorization, the matrix is distributed by a 2D block-cyclic mapping. In such a distribution, the pivot belongs to the same process as the diagonal element if and

[2] https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

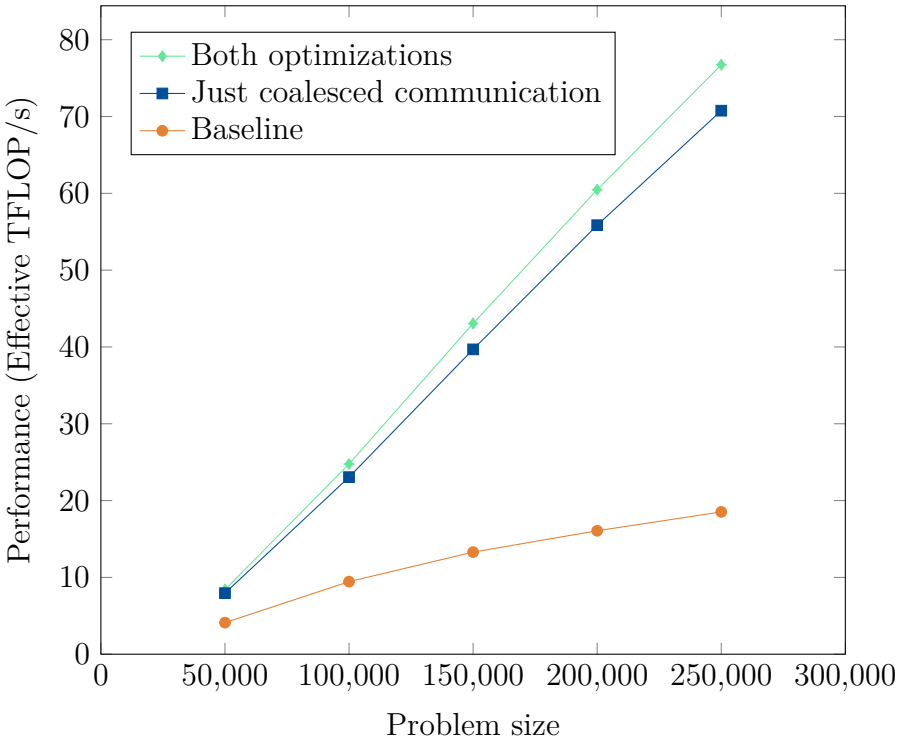


Figure 2.2: Effect of pivot optimizations on the performance of GETP. The performance rate is computed as if for $\frac{2}{3}n^3$ FLOP.

only if the corresponding rows can be exchanged without communicating between processes. So, the volume of communication can be fully determined using the distribution of just one column, in particular the column being searched. This equivalency, and thus the following theory, also holds for most other distributions; however, certain unusual distributions may require a different analysis.

Using threshold pivoting to reduce the inter-process communication modifies only the pivot search of partial pivoting. For clarity's sake, denote the process that owns the current diagonal element as the *root process*. First, each process finds its local maximum, as in partial pivoting. Then, in the global reduction, the candidates from non-root processes are reduced by a factor of τ, as shown in Fig. 2.3. Thus, the selected pivot is the largest element from the root process if and only if that element satisfies (2.2). And, if the selected pivot is not from the root process, then it is the global maximum.

Threshold pivoting can also reduce intra-process data movement by preferring the diagonal row over other rows on the root process. This extends the above procedure by adding another global reduction to check if the diagonal element already satisfies (2.2). A single `MPI_Allreduce` can do these reductions simultaneously, so the added overhead will be negligible [27]. Algorithm 2.3 shows this approach. The local panel search is unchanged from a regular code, as per lines 3–5. The best local pivot candidate is α and its index is ℓ. The threshold logic begins at line 6. On most processes, ℓ will be the argument for both reductions. However, if $A[j, j]$ satisfies (2.2) for the elements of the root process, j will be its argument for the first reduction. Both reductions scale the non-root values by τ. If the first reduction yields j, then row j is selected as the pivot. Otherwise, the result of the second reduction is selected as the pivot.

In Algorithm 2.3, $A[j, j]$ fulfills (2.2) if and only if it is the result of the first reduction because of lines 8 and 11, respectively, for elements on the root process and on the non-root processes. Similarly, the maximum element from the root process satisfies (2.2) if and only if it is the result of the second maximization. Therefore, the selected pivot satisfies (2.2) while minimizing data movement. Even though the reductions are identical unless the condition in line 8 is false, that value is known by only the root process. So, always computing both reductions minimizes the latency cost.

Finally, this approach can be further extended to deeper hierarchies, such as using the network topology or distributing a column across multiple accelerators within a process. As before, each communication layer has a corresponding maximization where the non-local entries are penalized by a multiple of τ. Then, consider the reductions in order of cost. If any maximization gave a local entry, take the first such entry. Otherwise, use the result of the final reduction. As before, this gives a pivot with minimal cost that satisfies (2.2).

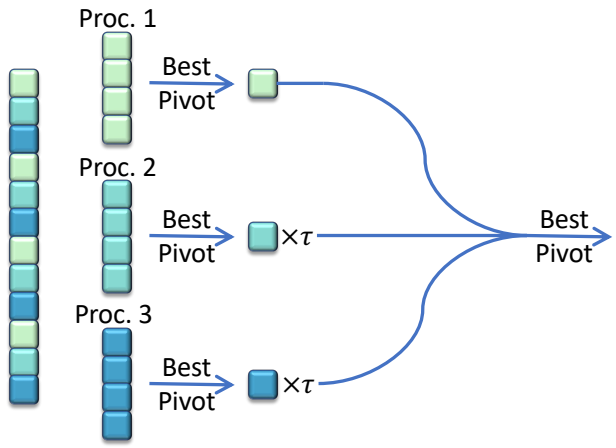


Figure 2.3: Threshold pivoting for reducing inter-process data movement. © 2022 IEEE

1: **procedure** PANEL_SEARCH(A, j, τ)
2: $\mathcal{I} \leftarrow$ local indices of $A[j{:}n, j]$
3: $\ell \leftarrow$ first entry in $\mathcal{I}$; $\alpha \leftarrow |A[\ell, j]|$
4: **for** $i \in \mathcal{I}$ **do**
5: **if** $|A[i, j]| > \alpha$ **then** $\ell \leftarrow i$; $\alpha \leftarrow |A[\ell, j]|$
6: $\ell_1 \leftarrow \ell$; $\ell_2 \leftarrow \ell$
7: **if** $j \in \mathcal{I}$ **then** ▷ If this is the root process
8: **if** $|A[j, j]| \geq \tau |A[\ell, j]|$ **then** $\ell_1 \leftarrow j$
9: $\alpha_1 \leftarrow |A[\ell_1, j]|$; $\alpha_2 \leftarrow |A[\ell_2, j]|$
10: **else**
11: $\alpha_1 \leftarrow \tau |A[\ell_1, j]|$; $\alpha_2 \leftarrow \tau |A[\ell_2, j]|$
12: $\ell_1 \leftarrow$ global_argmax(ℓ_1, α_1)
13: $\ell_2 \leftarrow$ global_argmax(ℓ_2, α_2)
14: **if** $\ell_1 = j$ **then return** ℓ_1
15: **else return** ℓ_2

Algorithm 2.3: Threshold Pivoting Panel Search © 2022 IEEE

Additionally, this can model the communication for complex matrix distributions by grouping rows into layers based on the amount of communication required.

2.2.1 Theoretical Bounds for Threshold Pivoting

Gaussian elimination computes a solution, $\widehat{x}$, to the equation $Ax = b$ such that

$$(A + \Delta A)\widehat{x} = b, \quad \|\Delta A\|_\infty \leq \tfrac{3nu}{1-3nu}(1 + 2(n^2 - n)\rho(A))\|A\|_\infty, \quad \rho(A) = \frac{\max_{i,j,k} |A^{(k)}[i,j]|}{\max_{i,j} |A[i,j]|}$$

where $A^{(k)}$ is the matrix after factoring k columns, $\rho(A)$ is called the *growth factor* and u is the *unit roundoff* for the floating-point format [71, Thm. 9.4, Lemma 9.6]. Because the n^3 factor is pessimistic in practice [73, 74], I focus on the growth factor.

The bound for the growth factor of threshold pivoting is similar to that of partial pivoting. At step k, let $\alpha = \max_{ij} |A^{(k)}[i,j]|$. Then, the values computed in the kth Schur complement have a magnitude of at most $\alpha + \tau^{-1}\alpha$. Applying this recursively gives

$$\rho(A) \leq (1 + \tau^{-1})^{n-1}. \tag{2.3}$$

This bound is tight and achieved with the matrix

$$\begin{bmatrix} \tau & 0 & \dots & 0 & 1 \\ -1 & \tau & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & \dots & \tau & 1 \\ -1 & -1 & \dots & -1 & 1 \end{bmatrix},$$

which is based on Wilkinson's matrix with exponential growth in partial pivoting [134]. For $\tau = 1$ and $\tau = 0$, (2.3) is a tight bound on the growth of partial pivoting and not pivoting, respectively.

Ideally, for a given matrix, the growth with threshold pivoting would be bounded in terms of the growth with partial pivoting. Unfortunately, this bound is exponential in the matrix size and thus not meaningfully better than (2.3). Minor variations of Wilkinson's matrix

demonstrate the exponential relation. Let

$$W_{\alpha\beta} = \begin{bmatrix} 1+\alpha & 0 & \dots & 0 & 1 \\ -1 & 1 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & \dots & 1 & 1 \\ -1-\beta & -1 & \dots & -1 & 1 \end{bmatrix} \quad \text{and} \quad \Omega_{\alpha\beta} = \begin{bmatrix} -1-\beta & -1 & \dots & -1 & 1 \\ -1 & 1 & \dots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & \dots & 1 & 1 \\ 1+\alpha & 0 & \dots & 0 & 1 \end{bmatrix}.$$

These matrices differ only in the exchange of the first and last rows. Let $0 < \tau < 1$ and $0 < \delta < \max(\tau^{-1} - 1, 1)$. Assume the diagonal element is selected as the pivot if it satisfies the appropriate pivoting constraint, i.e., (2.1) or (2.2). Then, apply partial pivoting and threshold pivoting to $W_{0\delta}$ and $\Omega_{\delta 0}$. For both matrices, partial pivoting will exchange the first and last rows at the first step (giving $\Omega_{0\delta}$ and $W_{\delta 0}$), while threshold pivoting will not (leaving $W_{\delta 0}$ and $\Omega_{\delta 0}$). As shown below, the growth is exponential in n for $W_{\delta 0}$ and $W_{0\delta}$ but constant for $\Omega_{\delta 0}$ and $\Omega_{0\delta}$. Hence, the growth of threshold pivoting is not meaningfully bounded by that of partial pivoting and vice versa. These drastic differences likely relate to Trefethen's conjecture that exponential growth is rare for partial pivoting because such growth "correspond[s] to unstable 'modes' that are themselves somehow unstable" [124].

Growth of $W_{\delta 0}$ and $W_{0\delta}$

First, consider $W_{\delta 0}$ and $W_{0\delta}$. Because the upper triangular parts of both are mostly zero, δ appears in only the last column after the first Schur complement. So, the factorization will not pivot, as per Wilkinson's matrix. Because the δ in $W_{0\delta}$ occurs in the last row, it increases only the (n, n) element by δ. Thus, the growth of $W_{0\delta}$ is

$$(1+\delta)^{-1}(2^{n-1} + \delta) \approx 2^{n-1}.$$

For $W_{\delta 0}$, all entries in the last column are $(2+\delta)/(1+\delta)$ after the first Schur complement. Then, the elements in the last column double at each step, as per Wilkinson's matrix. Thus, the growth of $W_{\delta 0}$ is

$$(1+\delta)^{-2}(2+\delta)2^{n-2} \approx 2^{n-1}.$$

Growth of $\Omega_{\delta 0}$

Next, consider $\Omega_{\delta 0}$. Let $\Omega_{\delta 0}^{(k)}$ be the matrix after k steps of elimination on $\Omega_{\delta 0}$. For notational simplicity, the indices are offset by k; i.e., the lower-rightmost element is always $\Omega_{\delta 0}^{(k)}[n, n]$. Note that all entries in the first row of U are trivially -1. Furthermore, all entries of the

first column of L are 1, except the last which is $-1-\delta$. The first Schur complement gives

$$\Omega_{\delta 0}^{(1)} = \begin{bmatrix} 2 & 1 & \dots & 1 & 0 \\ 0 & 2 & \dots & 1 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 2 & 0 \\ -1-\delta & -1-\delta & \dots & -1-\delta & 2+\delta \end{bmatrix}.$$

For $2 \le k < i, j \le n$, if no further pivoting occurs,

$$\Omega_{\delta 0}^{(k)}[i,j] = \frac{\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}]}{\det \Omega_{\delta 0}^{(1)}[2{:}k, 2{:}k]}$$

by Schur's identity [113].[3] Because $\Omega_{\delta 0}^{(1)}[2{:}k, 2{:}k]$ is upper triangular, its determinant is 2^{k-1}. The bound for the numerator is separated into six cases, depending on the position of $A[i,j]$ in the matrix. The first three cases correspond to on, below, and above the diagonal, respectively, in the leading principal part. The latter three cases correspond to on, below, and above the diagonal, respectively, in the last row or column.

Case 1: $i = j < n$. Since $\Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}]$ equals the $(k+1)$st leading principal submatrix,

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = \det \Omega_{\delta 0}^{(1)}[2{:}k+1, 2{:}k+1] = 2^k.$$

Case 2: $i < j < n$. The last row is zero for all but the last element, which is one. Expanding it gives

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = 1 \det \Omega_{\delta 0}^{(1)}[2{:}k, 2{:}k] = 2^{k-1}.$$

Case 3: $j < i < n$. The ith row is zero. So,

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = 0.$$

Case 4: $i = j = n$. The last column is zero except for the last element, which is $2+\delta$. Expanding it gives

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = (2+\delta) \det \Omega_{\delta 0}^{(1)}[2{:}k, 2{:}k] = (2+\delta)2^{k-1}.$$

[3] This use for computing intermediate values of Gaussian elimination seems to have originated from Grossman [67] with refinement from Gantmacher [52, p. 26].

Case 5: $i < j = n$. The last column is zero. So,

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = 0.$$

Case 6: $j < i = n$. Adding $(1+\delta)^{-1}$ times the last row to the first leaves the determinant unchanged but makes the first row equal to e_1^T. Expanding this row gives 1 times the determinant of a matrix with the same structure less the first row and column. Recursively applying this procedure gives

$$\det \Omega_{\delta 0}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = 1 \cdot \det[-1-\delta] = -1-\delta.$$

Thus, in all six cases, the diagonal elements are either 2 or $2+\delta$, and off-diagonal elements are either 0, $-2^{-k+1}(1+\delta)$, or ± 1. So, no pivoting will occur during the factorization. Hence, the maximum element during the process is $2+\delta$ and

$$\rho(\Omega_{\delta 0}) = \frac{2+\delta}{1+\delta} \leq 2. \qquad \square$$

Growth of $\Omega_{0\delta}$

Finally, consider $\Omega_{0\delta}$. Its growth can be bounded similarly to that of $\Omega_{\delta 0}$. Again, let $\Omega_{0\delta}^{(k)}$ be the matrix after k steps of elimination on $\Omega_{0\delta}$, and assume its indices are offset by k. The first Schur complement gives

$$\Omega_{0\delta}^{(1)} = (1+\delta)^{-1} \begin{bmatrix} 2+\delta & 1 & \dots & 1 & \delta \\ -\delta & 2+\delta & \dots & 1 & \delta \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -\delta & -\delta & \dots & 2+\delta & \delta \\ -1 & -1 & \dots & -1 & 2+\delta \end{bmatrix}.$$

Let $\Omega_{0\delta}^{(1*)} = (1+\delta)\Omega_{0\delta}^{(1)}$. For $2 \leq k < i, j \leq n$, if no further pivoting occurs,

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{\det \Omega_{0\delta}^{(1)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}]}{\det \Omega_{0\delta}^{(1)}[2{:}k, 2{:}k]} = \frac{\det \Omega_{0\delta}^{(1*)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}]}{(1+\delta)\det \Omega_{0\delta}^{(1*)}[2{:}k, 2{:}k]}. \tag{2.4}$$

by Schur's identity [113]. Unlike $\Omega_{\delta 0}$, there are no zeros in $\Omega_{0\delta}^{(1*)}$ that can be used to simplify the determinants. However, we can introduce zeros, without changing the determinant, by adding a scalar multiple of one row to another. The numerator is again divided into six cases, with the first case addressing the denominator.

Case 1: $i = j < n$. The sub-diagonal elements in the first column can be eliminated by adding the first row times $\chi_1 = \delta/(2+\delta)$ to the subsequent rows, giving

$$\begin{bmatrix} 2+\delta & 1 & 1 & \dots & 1 \\ 0 & 2+\delta+\gamma_1 & 1+\gamma_1 & \dots & 1+\gamma_1 \\ 0 & -\delta+\gamma_1 & 2+\delta+\gamma_1 & \dots & 1+\gamma_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -\delta+\gamma_1 & -\delta+\gamma_1 & \dots & 2+\delta+\gamma_1 \end{bmatrix}$$

where $\gamma_1 = \chi_1$. Then, the sub-diagonal elements in the second column can be eliminated by adding the second row times $\chi_2 = (\delta-\gamma_1)/(2+\delta+\gamma_1)$ to the subsequent rows, giving

$$\begin{bmatrix} 2+\delta & 1 & 1 & \dots & 1 \\ 0 & 2+\delta+\gamma_1 & 1+\gamma_1 & \dots & 1+\gamma_1 \\ 0 & 0 & 2+\delta+\gamma_2 & \dots & 1+\gamma_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & -\delta+\gamma_2 & \dots & 2+\delta+\gamma_2 \end{bmatrix}$$

where $\gamma_2 = \gamma_1 + \chi_2(1+\gamma_1)$. Continuing the procedure gives multiples defined recursively by

$$\chi_\ell = \frac{\delta-\gamma_{\ell-1}}{2+\delta+\gamma_{\ell-1}}, \qquad \text{and} \qquad \gamma_\ell = \gamma_{\ell-1} + \chi_\ell(1+\gamma_{\ell-1})$$

with $\chi_0 = 0$ and $\gamma_0 = 0$. Note that

$$\gamma_\ell = \frac{\gamma_{\ell-1}(2+\delta+\gamma_{\ell-1}) + (\delta-\gamma_{\ell-1})(1+\gamma_{\ell-1})}{2+\delta+\gamma_{\ell-1}} = \frac{\delta+\gamma_{\ell-1}+2\delta\gamma_{\ell-1}}{2+\delta+\gamma_{\ell-1}}.$$

If $\gamma_{\ell-1} = 0$, then $\gamma_\ell = \delta/(2+\delta)$. And if $\gamma_{\ell-1} = \delta$, then $\gamma_\ell = \delta$. Furthermore, it is straightforward to show that the partial derivative of γ_ℓ with respect to $\gamma_{\ell-1}$ is positive. Thus, if $\gamma_{\ell-1} \in [0,\delta]$, then $\gamma_\ell \in [0,\delta]$. Hence, $\gamma_\ell \in [0,\delta]$ for all ℓ.

Because this elimination to triangular form leaves the determinant unchanged, we have

$$\det \Omega_{0\delta}^{(1*)}[2{:}k \cup \{i\}, 2{:}k \cup \{j\}] = \prod_{\ell=0}^{k-1}(2+\delta+\gamma_\ell) \quad \text{and} \quad \det \Omega_{0\delta}^{(1*)}[2{:}k, 2{:}k] = \prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell).$$

Substituting these equalities into (2.4) gives

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{\prod_{\ell=0}^{k-1}(2+\delta+\gamma_\ell)}{(1+\delta)\prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell)} = \frac{2+\delta+\gamma_{k+1}}{1+\delta}, \quad \text{and} \quad \frac{2+\delta}{1+\delta} \le \Omega_{0\delta}^{(k)}[i,j] \le 2.$$

Case 2: $i < j < n$. The second case is similar to the first, except the lower-right element starts as 1 and becomes $1 + \gamma_{k-2}$ after eliminating the sub-diagonal elements. So,

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{1+\gamma_{k-2}}{1+\delta}, \qquad \text{and} \qquad \frac{1}{1+\delta} \leq \Omega_{0\delta}^{(k)}[i,j] \leq 1.$$

Case 3: $j < i < n$. The third case is like the first two, except the lower-right element starts as $-\delta$ and becomes $-\delta + \gamma_{k-2}$ after eliminating the sub-diagonal elements. So,

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{-\delta+\gamma_{k-2}}{1+\delta}, \qquad \text{and} \qquad \frac{-\delta}{1+\delta} \leq \Omega_{0\delta}^{(k)}[i,j] \leq 0.$$

Case 4: $i = j = n$. The same process can be applied, except with the multiples for the last row being scaled by δ^{-1}. Hence,

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{(2+\delta+\delta^{-1}\gamma_{k-1})\prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell)}{(1+\delta)\prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell)} = \frac{2+\delta+\delta^{-1}\gamma_{k-1}}{1+\delta},$$

and

$$\frac{2+\delta}{1+\delta} \leq \Omega_{0\delta}^{(k)}[i,j] \leq \frac{3+\delta}{1+\delta}$$

Case 5: $i < j = n$. The fifth case is identical to the second, except for the last column, and thus the numerator is scaled by δ. So,

$$\frac{\delta}{1+\delta} \leq \Omega_{0\delta}^{(k)}[i,j] \leq \delta.$$

Case 6: $j < i < n$. Sub-diagonal elements in all but the last row can be eliminated without changing the determinant similar to before. Because the final row is all -1, multiplying it by $1 + \gamma_\ell$ and adding it to the ℓth row zeros out the upper triangular part, but leaves $1 + \delta$ on the diagonal. So,

$$\Omega_{0\delta}^{(k)}[i,j] = \frac{(1+\delta)^{k-2}(-1)}{(1+\delta)\prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell)} = \frac{-(1+\delta)^{k-3}}{\prod_{\ell=0}^{k-2}(2+\delta+\gamma_\ell)},$$

and

$$\frac{-(1+\delta)^{k-3}}{(2+\delta)^{k-1}} \leq \Omega_{0\delta}^{(k)}[i,j] \leq \frac{-1}{(2+2\delta)^{k-1}}.$$

Hence, in all six cases, the off-diagonal elements are always strictly less than the diagonal elements, ensuring that no pivoting occurs after the first step. Therefore,

$$\rho(\Omega_{0\delta}) = \frac{(3+\delta)/(1+\delta)}{1+\delta} = \frac{3+\delta}{(1+\delta)^2} \leq 3. \qquad \square$$

2.2.2 Experimental Results

Both of the proposed thresholding strategies were implemented as modifications of SLATE's existing GEPP [86]. The existing `options` argument controls the threshold, demonstrating a backward-compatible addition of threshold pivoting to an existing LU factorization code.

Ten matrices were tested, five random and five structured, all of order $n = 225\,000$. The random matrices were: (1) `rand` (entries uniformly distributed on $[0, 1]$), (2) `rands` (entries uniformly distributed on $[-1, 1]$), (3) `randn` (entries normally distributed), (4) `randb` (entries randomly selected from $\{0, 1\}$), and (5) `rand+nI` which is `rand` plus n times the identity (making it diagonally dominant). The structured matrices are based on matrices from the MATLAB gallery: (6) `circul`, (7) `fiedler`, (8) `orthog`, (9) `riemann`, and (10) `ris`.

Experimental Setup

Threshold pivoting was tested on eight nodes of the Summit supercomputer. Recent releases of SLATE include the two-level threshold pivoting; the specific version used in these tests is available at https://zenodo.org/record/6972268. The software stack included GCC 9.1.0, CUDA 11.0.3, IBM Spectrum MPI 10.4.0.3-20210112, IBM ESSL 6.1.0-2, Netlib LAPACK 3.8.0, Netlib ScaLAPACK 2.1.0, and PAPI 6.0.0.1 [121].

Because MPI and BLAS libraries often initialize during the first call, warm-up tests of size 5000 were run before the reported tests. Hyperthreading was disabled with the `smt1` mode. Performance and accuracy were measured with SLATE's test code; however, the accuracy was scaled by n to compensate for a division by n in its calculation. The flags `-origin h -target d -seed 42 -seedB 24 -ref n -check y -nb 896 -ib 32 -panel-threads 18 -lookahead 3 -grid 4x4 -dim 5000,225000` were always used; the `-matrix` and `-piv-thresh` flags were set as appropriate. The PAPI event `ibmpowernv-isa-0000.System.energy11_input` was used to measure a node's cumulative energy usage in millijoules (mJ).

Effects on Performance and Accuracy

First, I reduced just the inter-process exchanges with the approach from Fig. 2.3. Various thresholds were tested on `rand+nI`, `rand`, and `orthog`. Each test ran three times, measuring the time to solve a double-precision system of equations with ten right-hand sides. Figure 2.4 summarizes the result with the mean and 95% confidence interval. First, the relative times for $\tau = 1$ of `rand` and `orthog` compared to `rand+nI` (which does not pivot) imply that slightly over half the time in the solve is spent exchanging rows. This supports the notion that pivoting is costly. Next, consider how changing τ affected the performance for each matrix. The `rand+nI` matrix was unaffected by the threshold; this is expected since partial

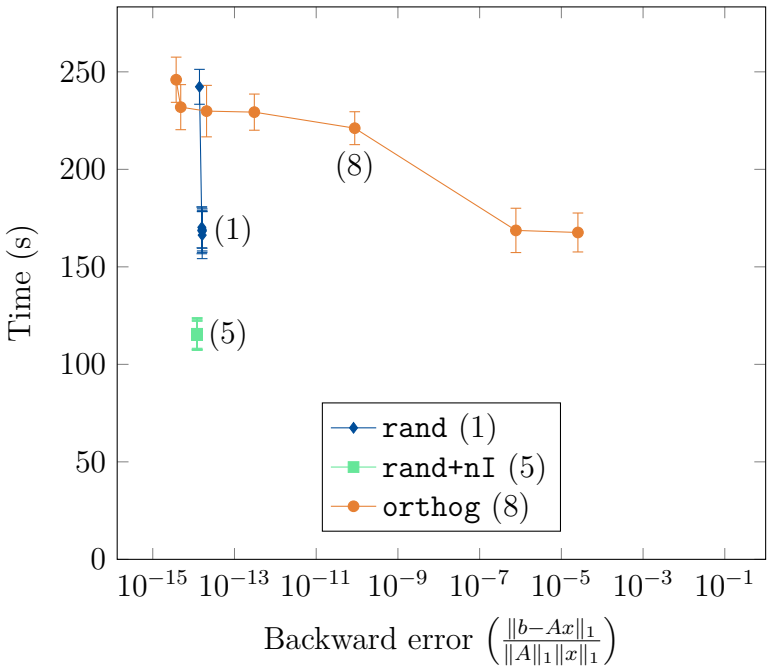


Figure 2.4: Tradeoffs between performance and accuracy when avoiding just inter-process row swaps. Each line has points for thresholds of 1 (slowest), 2^{-1}, 10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, and 0 (fastest) for that matrix. For `rand`, all but the first point overlap. © 2022 IEEE

pivoting selects diagonal pivots for diagonally dominant matrices. The `rand` matrix received a 46% speedup and a negligible effect on accuracy going from $\tau = 1$ to $\tau = 2^{-1}$, but further reductions in the threshold had little effect on either performance or accuracy. Even with $\tau = 0$, the time for the random matrix was 47% larger than for the diagonally dominant matrix; this indicates that intra-node row swaps contribute a significant overhead. The `orthog` matrix had significant reductions in accuracy as τ decreased and, for $\tau \geq 10^{-4}$, minimal increase in performance. However, the accuracy for $\tau = 2^{-1}$ was almost that of $\tau = 1$. So, for these matrices, $\tau = 2^{-1}$ does not degrade accuracy while sometimes improving performance.

Because the best performances achieved for `rand` and `orthog` were significantly below that of `rand+nI`, I repeated the previous experiment except avoiding both inter- and intra-process row swaps using Algorithm 2.3. Figure 2.5 shows the results. Avoiding both types of data movement allows the best-case performance to match that of the diagonally dominant case. However, the error increased as the tolerance decreased. For `rand`, $\tau = 2^{-1}$ provided the same accuracy and a 31% reduction in run time compared to partial pivoting, while $\tau = 10^{-2}$ provided the same performance and a 4-digit improvement in error compared to never exchanging rows. This provides a useful selection of τ, depending on the relative importance of performance and accuracy. On the other hand, `orthog` saw only a 6% improvement in performance compared to partial pivoting for $\tau = 2^{-1}$, although the effect on accuracy was still negligible. Even when the tolerance was still as low as $\tau = 10^{-4}$, the improvement was merely 12%, but the error increased to $1.6 \cdot 10^{-9}$. This demonstrates that threshold pivoting will not consistently improve performance, although high accuracy was still obtained with large tolerance values. Compared to Fig. 2.4, the results for $\tau = 1$ had a slight reduction in performance. However, because the corresponding confidence intervals overlap, this may stem from system noise. Furthermore, the difference was only a few percent, so even if the difference is entirely due to the increased complexity of pivot selection, the overhead is inconsequential.

With the successes of $\tau = 2^{-1}$ and $\tau = 10^{-1}$ in the previous experiment, I compared the performance and accuracy of the remaining matrices for those thresholds and $\tau = 1$ (i.e., partial pivoting). For reference, I also factored `rand+nI` with SLATE's non-pivoted LU code; this routine has increased parallelism and better GPU utilization but risks catastrophic numerical failure if the matrix is not diagonally dominant. Figure 2.6 shows the results. The added random matrices behave similarly to `rand`. However, three of the added structured matrices saw significant performance improvements with minimal reduction in accuracy, unlike `orthog`. Furthermore, two of the three (`circul` and `fiedler`) achieved performance equivalent to the diagonally dominant matrix for the smallest threshold. The last structured

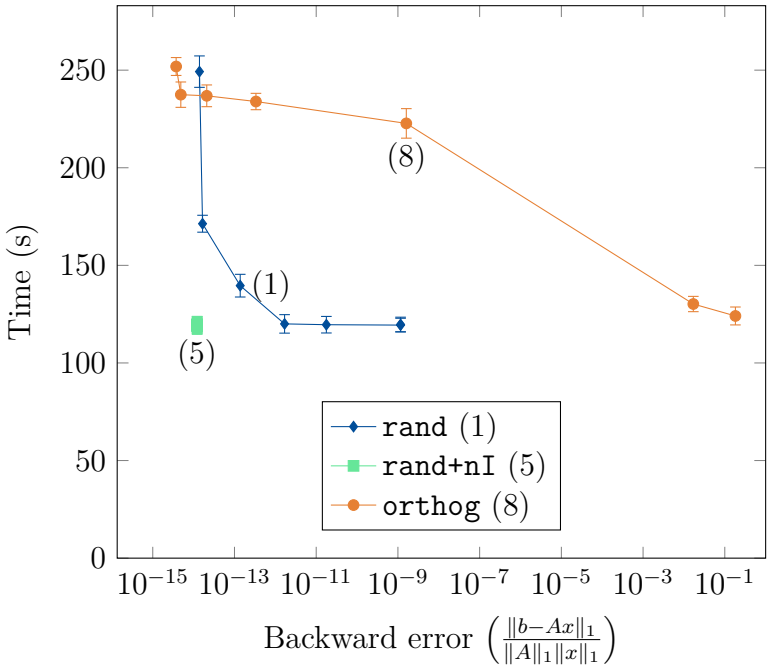


Figure 2.5: Tradeoff between performance and accuracy when avoiding inter- and intra-process row swaps. Each line has points for thresholds of 1 (slowest), 2^{-1}, 10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, and 0 (fastest) for that matrix. For `rand`, the last two points overlapped. © 2022 IEEE

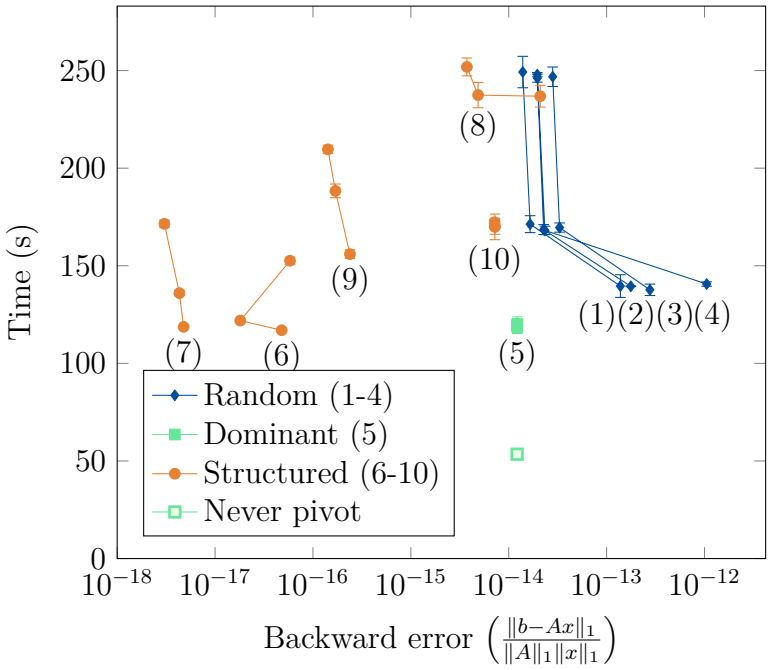


Figure 2.6: Tradeoffs between performance and accuracy when avoiding inter- and intra-process row swaps for a variety of matrices. Each line has points for thresholds of 1 (slowest), 2^{-1}, and 10^{-1} (fastest) for that matrix. © 2022 IEEE

matrix (`ris`) was unaffected by the tested tolerance values; this likely stems from the entries being very small except along the anti-diagonal, which limits the available pivots for the moderate thresholds tested here. Interestingly, one matrix (`circul`) had higher accuracy for $\tau = 2^{-1}$ than for $\tau = 1$; this reflects the observation in Section 2.2.1 that the accuracy of threshold pivoting can, on occasion, be better than that of partial pivoting.

These results indicate that a threshold of 2^{-1} or 10^{-1} is a reasonable, general-purpose default that consistently achieves an accuracy close to partial pivoting. This aligns with the conservative recommendations from the sparse-factorization literature. However, a threshold of 10^{-2} is also commonly recommended for the general case, and recommendations for specific applications can be much smaller, e.g., 10^{-8} [77]. This reflects a lower cost to pivot in dense factorizations due to the full nonzero structure.

Effects on Energy Consumption

In light of the performance improvements, I tested the effect of this strategy on energy consumption. Because the performance overhead of the intra-process exchanges was significant, only the two-layer formulation was considered for the energy consumption. I provide the results as the energy consumed to solve the system in megajoules (MJ). Another common metric is the "flops per watt," which is used by benchmarks such as the Green500 [49] list, as well as supercomputing power consumption research [78, 96]. This metric divides the performance in GFLOP/s by the average power usage in Watts, resulting in an efficiency metric measured in (GFLOP/s)/W. However, for a fixed n, this is inversely proportional to the total energy, so I provide only the latter. For the tested size, $n = 225\,000$, using 1 MJ to solve a system is equivalent to achieving 7.59 (GFLOP/s)/W.

Figure 2.7 shows the effect of varying τ on energy usage for the `rand+nI`, `rand`, and `orthog` matrices (cf. Fig. 2.5), while Fig. 2.8 shows the effect of threshold pivoting on energy for all ten matrices (cf. Fig. 2.6). The effect on energy consumption shows similar trends as the effect on run time for both experiments, albeit with less relative improvement. The lesser improvement likely stems from the inability to remove the energy usage of tasks by parallelizing them, unlike the time to solution.

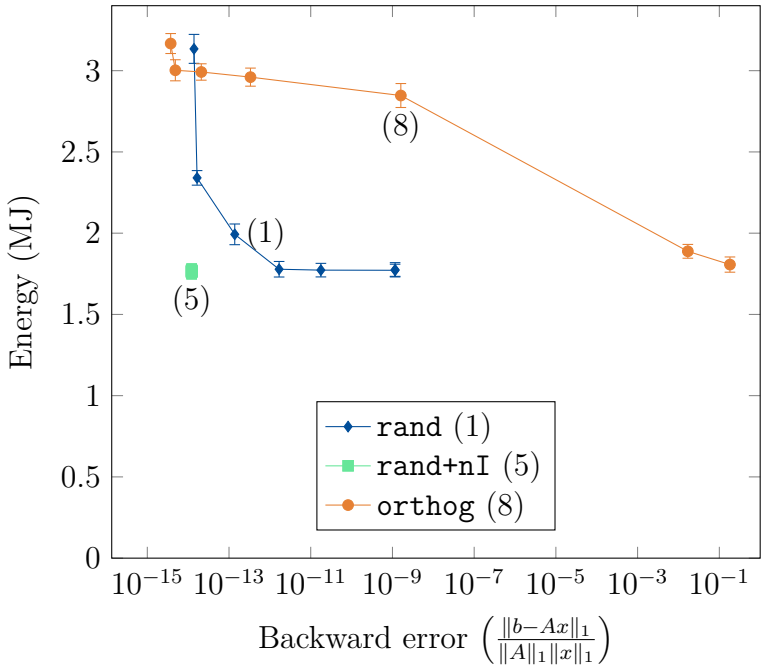


Figure 2.7: Tradeoffs between energy usage and accuracy when avoiding inter- and intra-process row swaps. Each line has points for thresholds of 1 (slowest), 2^{-1}, 10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}, and 0 (fastest) for that matrix. For `rand`, the last two points overlap. © 2022 IEEE

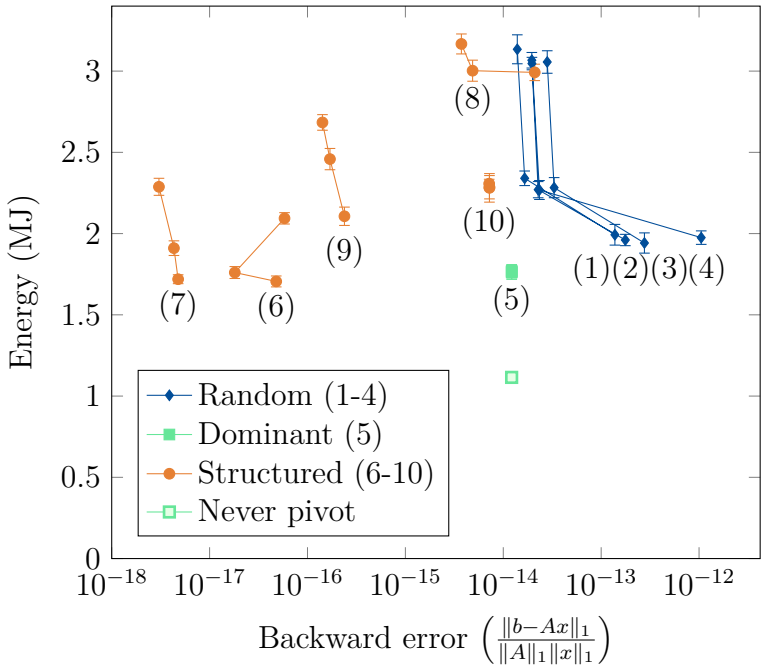


Figure 2.8: Tradeoffs between energy usage and accuracy of when avoiding inter- and intra-process row swaps for a variety of matrices. Each line has points for thresholds of 1 (slowest), 2^{-1}, and 10^{-1} (fastest) for that matrix. © 2022 IEEE

Chapter 3

Replacing Pivoting[1]

Even with optimizations like those in Chapter 2, partial pivoting will always introduce some overhead. First, even when no row exchanges are applied, detecting that requires a reduction between the processes owning elements in the panel. Techniques like tournament pivoting can help reduce the frequency, but they still must do a reduction for each block column. Second, there are always matrices that require pivoting between processes (unless a less-scalable 1d distribution is used). Dynamic pivoting can reduce the associated network communication, but balancing the remaining work between processes can still require communication. Third, the data dependencies from pivoting limit the available parallelism. Without pivoting, the triangular solves to compute the next block-column of L and the next block-row of U can be done simultaneously; pivoting forces them to be sequential. Furthermore, partial pivoting complicates the computation of the block-column of L by interleaving the pivot search with the triangular solve, resulting in memory-bound operations. Towards this end, I have also explored two techniques to replace pivoting. The first randomizes the matrix so that GENP can factor the matrix accurately. The second is a novel factorization that combines block LU with additive modifications.

3.1 The Growth Factor

Element growth in LU and LU-like methods often leads to problematic cancellation errors, and so it is crucial in understanding the backward error of these factorizations. Normally,

[1]Section 3.3 reuses material from one of my published papers [93] (© 2020 IEEE) and Section 3.4 reuses material from another [95] (although Sections 3.4.3 and 3.4.6 are entirely new material). Coauthors include Piotr Luszczek and Jack Dongarra. Reused coauthor contributions include high-level guidance, Fig. 3.6, and textual improvements.

growth is measured by Wilkinson's *growth factor*, which is defined as

$$\rho(A) = \frac{\max_{1\le k<i,j\le n} |A^{(k)}[i,j]|}{\max_{1\le i,j\le n} |A[i,j]|}. \tag{3.1}$$

Then, solving $Ax = b$ with Gaussian elimination gives a solution, $\widehat{x}$, such that

$$(A+\Delta A)\widehat{x} = b \qquad \text{and} \qquad \|\Delta A\|_\infty \le \tfrac{3nu}{1-3nu}(1+2(n^2-n)\rho(A))\|A\|_\infty. \tag{3.2}$$

where u is the floating-point *unit roundoff* [71, Thm. 9.4, Lemma 9.6]. In practice, the cubic polynomial in n is not achieved; this is supported by probabilistic analysis [74, Thm. 3.7] and analysis of other growth factors [7] (see Section 3.4.3). This leaves growth as the primary concern. While (3.1) is the most commonly used, there are other ways to measure element growth [7, 22]; a generalization of (3.1) for block LU is described in Section 3.4.3.

The Schur complement can be used to bound the growth. For any $1 \le k \le i, j \le n$,

$$A^{(k)}[i,j] = A[i,j] - A[i,1{:}k]A^{-1}[1{:}k,1{:}k]A[1{:}k,j].$$

Taking $\max_{ij} A^{(k)}[i,j]$ and several triangle inequalities gives

$$\rho(A) \le 1 + \max_k \|A\|_\infty \|A^{-1}[1{:}k,1{:}k]\|_\infty \tag{3.3}$$

which provides a condition number-like value (and likewise for the 1-norm). Thus, growth can only occur when leading principal submatrices have small singular values.

The bound (3.3) is reflected by the exponential growth of partial pivoting for Wilkinson's matrix. Wilkinson's matrix has the form

$$W = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ -1 & -1 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & -1 & \cdots & 1 & 1 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{bmatrix}.$$

Each leading principal submatrix is an ill-conditioned matrix of Turing [127, pg. 307]; the inverses of the leading submatrices are a unit lower-triangular with subdiagonal elements equal to 2^{i-j-1} [106]. Hence, $\|W[1{:}k,1{:}k]\|_\infty = 2^{k-1}$ and (3.3) implies $\rho(W) \le n2^{n-2}$, which is an upper bound of the true growth of 2^{n-1}.

3.2 Implementation of GENP in SLATE

For both of the strategies to replace pivoting, the design of an efficient Gaussian elimination with no pivoting (GENP) is critical for performance (GENP itself for the RBT solver and the parallel structure for the additive solver). Towards this end, it is worth considering the GENP code that I developed for the Software for Linear Algebra Targeting Exascale (SLATE) library. Its structure resembles that of SLATE's Cholesky factorization, except that both the upper and lower parts are explicitly computed. Like SLATE's other factorization, this implementation uses a right-looking formulation, which allows the Schur-complements to be computed as a series of highly parallel block outer products and allows a lookahead to reduce the amount of work on the critical path.

Keeping with SLATE's design, the algorithm is formulated using high-level blocks and implemented by mapping those blocks and their dependencies to OpenMP tasks. Internode communication is explicitly specified with MPI in the appropriate tasks. Figure 3.1 shows the general structure of the parallel dependencies when the lookahead is 1. Each of these large tasks calls MPI or other internal routines, which are parallelized in turn with dependency-less tasks.

3.3 Random Butterfly Transforms

A butterfly matrix is a matrix of the form

$$\mathcal{B}^{\langle n \rangle} = \frac{1}{\sqrt{2}} \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix},$$

where R_0 and R_1 are $n/2 \times n/2$, nonsingular, diagonal matrices [107]. Then, a random butterfly transform (RBT), $\mathcal{U}^{\langle n \rangle}$, with depth d is a matrix of the form

$$\mathcal{U}^{\langle n \rangle} = \begin{bmatrix} \mathcal{B}_1^{\langle n/2^{d-1} \rangle} & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & \cdots & \mathcal{B}_{2^{d-1}}^{\langle n/2^{d-1} \rangle} \end{bmatrix} \times \cdots \times \begin{bmatrix} \mathcal{B}_1^{\langle n/2 \rangle} & 0 \\ 0 & \mathcal{B}_2^{\langle n/2 \rangle} \end{bmatrix} \times \mathcal{B}^{\langle n \rangle}. \tag{3.4}$$

Note that n must be a multiple of 2^d; however, a linear system can be augmented with ones on the diagonal and zeros on the off diagonals to add the necessary rows and columns to the matrix. Figure 3.2 shows the sparsity pattern for butterfly matrices and the resulting depth-two RBT. Additionally, (3.4) is a generalized form of the original definition of RBTs, which is the case where $d = \log_2(n) + 1$ [107]. ($\mathcal{B}^{\langle 1 \rangle}$ is defined to be a nonzero scalar.)

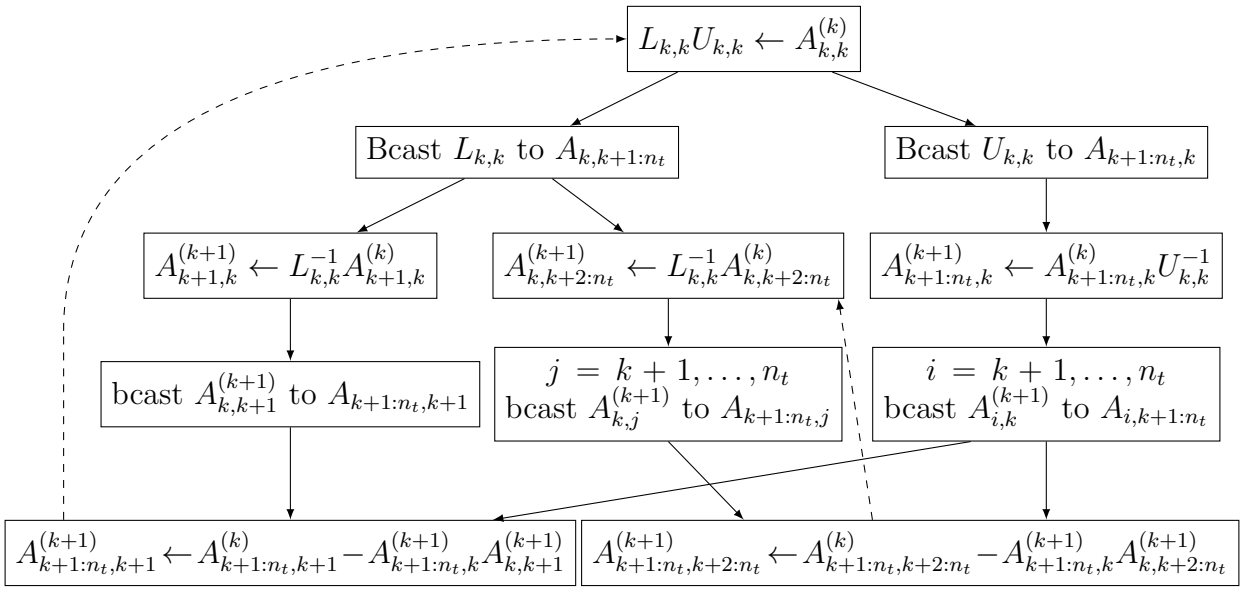
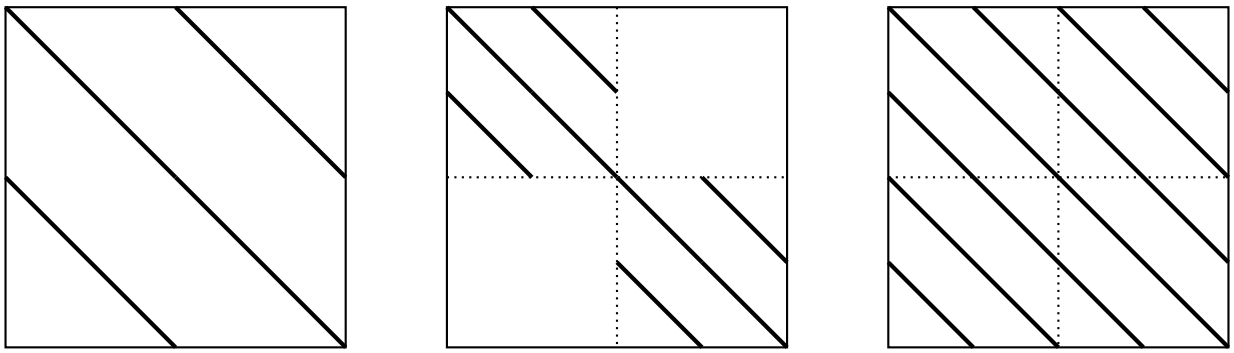


Figure 3.1: Task dependencies for GENP with a lookahead of 1. Dashed lines represent dependencies between iterations.



(a) Sparsity pattern of $\mathcal{B}^{\langle n \rangle}$.

(b) Sparsity pattern of $\operatorname{diag}\left(\mathcal{B}_1^{\langle n/2 \rangle}, \mathcal{B}_2^{\langle n/2 \rangle}\right)$.

(c) Sparsity pattern of a depth-two RBT.

Figure 3.2: Sparsity patterns of the last two terms of (3.4) and their product, a depth-two butterfly. © 2020 IEEE

For example, the structure of $\mathcal{U}^{\langle 4 \rangle}$ with a depth of 2 is

$$\mathcal{U}^{\langle 4 \rangle} = \frac{1}{2} \begin{bmatrix} r_{1,1} & r_{2,1} & & \\ r_{1,1} & -r_{2,1} & & \\ & & r_{3,1} & r_{4,1} \\ & & r_{3,1} & -r_{4,1} \end{bmatrix} \begin{bmatrix} r_{1,2} & & r_{3,2} & \\ & -r_{2,2} & & r_{4,2} \\ r_{1,2} & & -r_{3,2} & \\ & r_{2,2} & & -r_{4,2} \end{bmatrix},$$

where each $r_{i,j}$ is a scalar. The structure of RBT matrices is equivalent to the bit-shuffle permutations performed by FFT matrices [107] that bear the butterfly name due to the data transfer patterns they form [131]. This relation to FFT computational and communication processes makes RBT matrices efficient to apply in practice. Additionally, if each nonzero in the component butterfly matrices has magnitude one, the transform is unitary.

RBTs can be used to avoid pivoting by preconditioning the system. Let $\langle \mathcal{U} \rangle$ and $\langle \mathcal{V} \rangle$ be RBTs. Then, the linear system $Ax = b$ can be solved by the algorithm in Algorithm 3.1.[2] If a $2^d \times 2^d$, nonsingular matrix is preconditioned by a random, depth-d RBT, then GENP will succeed with probability 1 [107]. However, previous work has noted that most matrices can be preconditioned successfully with a depth-2 RBT if iterative refinement is also used [16].

It may appear that the RBTs could instead use butterflies of size $2^k \times 2^k$ for $k = 1, 2, \ldots, d$, which would reduce communication in distributed contexts. However, the conditions of the leading, principal submatrices with dimension divisible by 2^d do not change. Recall that GENP is numerically safe if and only if each leading, principal submatrix is nonsingular and well-conditioned [105]. So, this block diagonal form of RBT uses at most $(k + 2^d)^2$ elements to generate the leading $k \times k$ principal submatrix. On the other hand, the form of RBT described in (3.4) uses $\min(n^2, k^2 2^{2d})$ elements when generating the same submatrix. So, for a depth-2 RBT, about 16 times more elements affect moderately sized leading blocks, and leading blocks with $k \geq n/4$ are affected by all elements of the matrix. Thus, this later form of RBT can prevent ill-conditioned leading principal submatrices for a wider variety of matrices.

Packed Storage for Recursive Butterfly Transforms

Explicitly constructing the RBT matrices would triple the storage required for the linear system. However, the structured sparsity can be used to reduce storage costs. Note that a

[2]While this discussion uses matrix transposition, it is also valid for RBT matrices in the complex domain, either as written or after replacing the transpositions with conjugate transpositions.

1: $A' \leftarrow {\mathcal{U}^{\langle n \rangle}}^T \times A \times \mathcal{V}^{\langle n \rangle}$
2: $b' \leftarrow {\mathcal{U}^{\langle n \rangle}}^T \times b$
3: $L, U \leftarrow$ Apply GENP to A'
4: $x' \leftarrow U^{-1} \times L^{-1} \times b'$
5: $x \leftarrow \mathcal{V}^{\langle n \rangle} \times x'$

Algorithm 3.1: Solving $Ax = b$ with RBTs $\mathcal{U}^{\langle n \rangle}$ and $\mathcal{V}^{\langle n \rangle}$.

butterfly matrix is equivalent to

$$\begin{bmatrix} I & I \\ I & -I \end{bmatrix} \times R,$$

where R is diagonal. So, only the m diagonal elements of an $m \times m$ butterfly matrix need to be stored. Hence, a recursive butterfly transform of depth d and size n can be stored in an $n \times d$ dense matrix where each column stores one term of (3.4). Because $d \leq \log_2(n) + 1 \ll n$, recursive butterfly transforms introduce little additional storage when using packed storage.

Computation Cost of Recursive Butterfly Transforms

Similar to storing the matrix, utilizing the transform's structure provides significant benefits for RBT application. Let A be an $m \times m$ matrix, and let

$$\mathcal{B}_1^{\langle m \rangle} = \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix} \qquad \text{and} \qquad \mathcal{B}_2^{\langle m \rangle} = \begin{bmatrix} R_2 & R_3 \\ R_2 & -R_3 \end{bmatrix}$$

be butterfly matrices stored in vectors w_1 and w_2 using the packed format. Then,

$$(\mathcal{B}_1^{\langle m \rangle})^T A \mathcal{B}_2^{\langle m \rangle} = \begin{bmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} R_2 & R_3 \\ R_2 & -R_3 \end{bmatrix} = diag(w_1)\, C\, diag(w_2) \tag{3.5}$$

where

$$C = \begin{bmatrix} A_{11} + A_{12} + A_{21} + A_{22} & A_{11} - A_{12} + A_{21} - A_{22} \\ A_{11} + A_{12} - A_{21} - A_{22} & A_{11} - A_{12} - A_{21} + A_{22} \end{bmatrix}.$$

Thus, transforming an $m \times m$ matrix requires $4m^2$ FLOP. Applying RBTs to both sides of an $n \times n$ matrix can be broken down into $(n/m)^2$ butterfly matrix applications of size $m \times m$ and, thus, requires $4n^2$ FLOP. Hence, applying an RBT of depth d requires $4dn^2$ FLOP.

Similarly, an RBT in the packed format can be applied to a vector. Then, each transform only requires $\mathcal{O}(n)$ FLOP for a total of $\mathcal{O}(dn)$ FLOP to apply a recursive butterfly transform of depth d. Because $d \ll n$, the cost of transforming the vectors will be disregarded for the rest of the paper.

In distributed settings, it is also important to consider the amount of inter-process communication needed. Similar to computing the FLOP count, I start by considering the cost of applying a single butterfly matrix to each side of an $m \times m$ matrix, A. First, note that in the description above of applying a butterfly matrix to each side, each element, of $(\mathcal{B}_1^{\langle m \rangle})^T A \mathcal{B}_2^{\langle m \rangle} = (\alpha_{ij})$ depends on four elements in A: one element from either R_0 or R_1 and one element from either R_2 or R_3. Specifically, the dependencies of $\alpha_{1,1}$, $\alpha_{1+m/2,1}$, $\alpha_{1,1+m/2}$,

and $\alpha_{1+m/2,1+m/2}$ on $A = (a_{ij})$ are the elements in the same positions, i.e. $a_{1,1}$, $a_{1+m/2,1}$, $a_{1,1+m/2}$, and $a_{1+m/2,1+m/2}$. Furthermore, the elements in R_0 and R_1 are shared across the rows of the result, and the elements in R_2 and R_3 are shared across the columns. I only consider the case where $(\mathcal{B}_1^{\langle m \rangle})^T A \mathcal{B}_2^{\langle m \rangle}$ and A are distributed across the processes in the same manner, which occurs when reusing the storage of A to hold the resulting transformed matrix. Assume A is distributed in a 2D block-cyclic layout on a $p \times q$ process grid. Then by gathering each set of four interacting elements onto a single process where one of those elements already resides and returning them after doing the appropriate computation, up to $6m^2$ elements of the matrix, $mp/2$ elements of each R_0 and R_1, and $mq/2$ elements of each R_2 and R_3 must be transferred.

Then, to expand this analysis to the entire transform, simply combining the individual butterfly applications results in transferring $6dn^2$ elements of the matrix, $np/2$ elements of each R_0 and R_1, and $nq/2$ elements of each R_2 and R_3. Note that it is possible to merge the return of elements for one application with the gathering of elements for the next application. With this improvement, it would be possible to reduce the upper bound on matrix element transfers to $4dn^2 + 2n^2$. However, because of the increase in algorithm complexity, I have not implemented this improvement.

3.3.1 Implementing random butterfly transforms

I implemented and tested an RBT-based solver using SLATE. Given the recursive structure of RBT, I implemented the application by applying one term of the product in (3.4) at a time, albeit in a two-sided manner, to transform the matrix. Furthermore, the transformation is simplified by applying one pair of butterflies at a time, as described in Section 3.3. This is implemented as an elementwise operation on the four submatrices corresponding to each pair of left and right butterflies. Because the matrices are distributed, each set of four elements is sent to the node owning the upper left element to be transformed. Algorithms 3.2 and 3.3 show pseudocode for my two-sided RBT implementation.

The first procedure, RBT, breaks the transformation into individual butterfly applications. The second procedure, $\text{RBT2}_{\text{TILE}}$, applies a single pair of butterflies where the matrices and butterfly values are separated similar to the organization in (3.5). In a non-tiled code, $\text{RBT2}_{\text{TILE}}$ is equivalent to the omitted RBT2. However, as my implementation is tile-based and distributed, I have an additional step to group appropriate sets of elements into tiles on a single process before calling $\text{RBT2}_{\text{TILE}}$.

Unfortunately, the way the elements interact when applying an RBT rarely corresponds to the tiles used to store the matrix data. So, I implemented support for transmitting partial

```
1: procedure RBT(A, U, V)
2:     d ← depth(U)
3:     for k from d − 1 to 0 do
4:         b_n ← 2^k                                   ▷ Number of butterflies
5:         h ← n/2^{k+1}                               ▷ Half a butterfly's size
6:         for b_j from 0 to b_n do                    ▷ Right butterflies
7:             j_1 ← 2b_j h
8:             j_2 ← j_1 + h                           ▷ Column indices for b_j
9:             j_3 ← j_2 + h
10:            for b_i from 0 to b_n do                ▷ Left butterflies
11:                i_1 ← 2b_i h
12:                i_2 ← i_1 + h                       ▷ Row indices for b_i
13:                i_3 ← i_2 + h
14:                A_11 ← A[i_1:i_2, j_1:j_2]
15:                A_12 ← A[i_1:i_2, j_1:j_2]
16:                A_21 ← A[i_2:i_3, j_2:j_3]
17:                A_22 ← A[i_2:i_3, j_2:j_3]
18:                U_1 ← U[i_1:i_2, k]
19:                U_2 ← U[i_2:i_3, k]
20:                V_1 ← V[j_1:j_2, k]
21:                V_2 ← V[j_2:j_3, k]
22:                RBT2(A_11, A_12, A_21, A_22, U_1, U_2, V_1, V_2)
```

Algorithm 3.2: High-level algorithm for applying a two-sided RBT. © 2020 IEEE

1: **procedure** $\text{RBT2}_{\text{TILE}}(A_{11}, A_{12}, A_{21}, A_{22}, U_1, U_2, V_1, V_2)$
2: $m_b \times n_b \leftarrow \dim(A_{11})$ ▷ Tiles are all the same size
3: **for** j from 0 to n_b **do**
4: $v_1 \leftarrow V_1[j]$
5: $v_2 \leftarrow V_2[j]$
6: **for** i from 0 to m_b **do**
7: $u_1 \leftarrow U_1[j]$
8: $u_2 \leftarrow U_2[j]$
9: $a_{11} \leftarrow A_{11}[i, j]$
10: $a_{12} \leftarrow A_{12}[i, j]$
11: $a_{21} \leftarrow A_{21}[i, j]$
12: $a_{22} \leftarrow A_{22}[i, j]$
13: $A_{11}[i, j] \leftarrow u_1 v_1(a_{11} + a_{12} + a_{21} + a_{22})$
14: $A_{12}[i, j] \leftarrow u_1 v_2(a_{11} - a_{12} + a_{21} - a_{22})$
15: $A_{21}[i, j] \leftarrow u_2 v_1(a_{11} + a_{12} - a_{21} - a_{22})$
16: $A_{22}[i, j] \leftarrow u_2 v_2(a_{11} - a_{12} - a_{21} + a_{22})$

Algorithm 3.3: Applying a single, two-sided butterfly transform to a set of four tiles.

tiles in a manner similar to previous work on distributed, RBT-based solvers [14]; however, my approach differs in that I explicitly gather the elements into tiles defined by the size and process of the upper left submatrix. The gathered tiles can then be treated as the application of a single butterfly described in Section 3.3. Figure 3.3 shows an example of how elements are gathered for the two-sided transformation of A,

$$\begin{bmatrix} \mathcal{B}_1^{\langle n/2 \rangle} & 0 \\ 0 & \mathcal{B}_2^{\langle n/2 \rangle} \end{bmatrix}^T A \begin{bmatrix} \mathcal{B}_3^{\langle n/2 \rangle} & 0 \\ 0 & \mathcal{B}_4^{\langle n/2 \rangle} \end{bmatrix},$$

when A is tiled into a 5×5 grid of uniform size. Currently, my implementation does involve duplicate transfers of the butterfly nonzeros to reduce the complexity of computing the indices and managing storage lifetimes.

Note that if the size of the matrix can be controlled, it may be beneficial to adjust the matrix size so that the transformation can be efficiently applied. For example, a 2D block-cyclic distribution of an $n \times n$ matrix on a $p \times q$ process grid with tiles of size $b \times b$ does not need to subdivide tiles to apply RBTs when $2pb$ and $2qb$ divide n. Furthermore, for a butterfly depth of d, inter-process communication is unneeded when both $2^{d+1}pb$ and $2^{d+1}qb$ divide n.

Similar to previous work [16], I used a depth of two as the default, chose RBT elements of the form $\exp(r/10)$ with r chosen from the uniform distribution $[-\frac{1}{2}, \frac{1}{2}]$, and provide iterative refinement. While a depth of two does not provide the probabilistic guarantee of a depth of $\log_2(n)$, it does not require the matrix size to be a power of 2 and has lower computational overheads. The copy of the matrix for iterative refinement is always kept entirely on the CPUs to increase the maximum problem size that can be stored in GPU memory. Because a copy of the matrix is already needed to achieve the same accuracy as GEPP, there is limited overhead to check the backward error of the solution provided by the RBT-solver. So, problems that cannot be solved accurately with the RBT-solver can be re-solved with a regular GEPP implementation.

3.3.2 Experimental Results

Using the implementation of the RBT-based solver described in Section 3.3.1, I tested its accuracy and performance relative to SLATE's GEPP implementation for double precision on Summit. Because SLATE's GEPP and GENP have both been significantly optimized since this material was previously published [93], the experimental results were rerun for this dissertation; thus, the configuration differs in a few places.

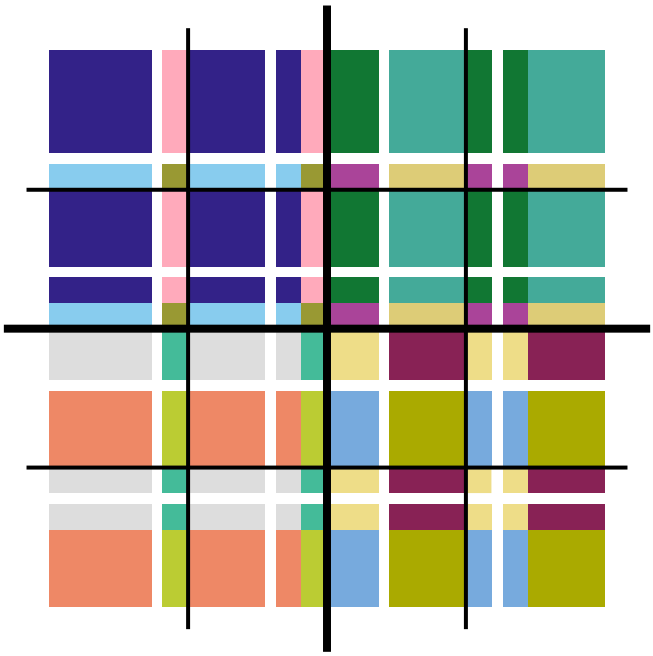


Figure 3.3: Subtiles gathered for computing a layer of an RBT for butterflies of size $n/2$. All submatrices of the same color are gathered onto the process owning the upper left entry of the color for the computation. © 2020 IEEE

First, consider the accuracy tests. The software stack included GCC 8.1.1, CUDA 10.1.243, IBM Spectrum MPI 10.3.1.2, IBM ESSL 6.1.0-2, and Netlib LAPACK 3.8.0 (to provide routines unsupported by ESSL). Summit was set to `smt2` mode. Results were measured using a modified version of SLATE's tester. Tests were run with the flags `--origin h --target d --ref n --nb 832 --seed 96 --seedB 42 --ib 64 --lookahead 0`. GEPP also used the flags `--panel-threads 20 --p 2 --q 8`, and the depth and maximum iterations of refinement were set as appropriate for the RBT solver. GEPP's process grid of 2×8 was allocated such that the processes sharing a column were located on the same node. The flags `--dim`, `--matrix`, and `--matrixB` were also used to control the test matrices.

Second, consider the performance tests. The software stack included GCC 9.1.0, CUDA 11.0.3, Spectrum MPI 10.4.0.3, ESSL 6.3.0, and Netlib LAPACK 3.9.1. Summit was run in `smt1` mode to disabling simultaneous multithreading. Results were measured using a modified version of SLATE's tester. The solvers' parameters were tuned for performance. Tests were run with the flags `--ref n --origin h --target d --seed 42 --seedB 64 --ib 64 --lookahead 1 --grid 4x4`. GENP also set `--nb 512`; the RBT solver was run with same the flags as GENP as well as the `--fallback n --depth 2 --itermax 0,1` flags. GEPP also used the flags `--nb 896 --panel-threads 16`. The flags `--dim`, `--matrix`, and `--matrixB` were also used to control the test matrices. A problem of size 10 000 was solved before running the actual performance tests to ensure that the initialization of BLAS and MPI was not included.

Accuracy Results

First, I tested the accuracy of the RBT solver in comparison to GEPP and GENP. I tested matrices of size 100 000 and compared the normwise backward error, $\|r\|_1/(\|A\|_1\|x\|_1)$, in Table 3.1. Both the RBT solver and GENP were run with and without one step of iterative refinement for the sake of comparison. The tested matrices are a subset of the matrices previously used to test RBT-based solvers for problems of size 1024 [16], with a minor adjustment to the last matrix. The elements of the first two matrices are uniformly selected $[0, 1]$ and $[-1, 1]$, respectively. The third matrix has elements selected from a normal distribution with mean 0 and standard deviation 1. The fourth matrix has elements selected from 0 and 1 with equal probability. The fifth and sixth matrices come from MATLAB's gallery function. Note that $|i - j|$ used in previous works is identical to `fielder`. The last matrix is based on the matrix in Higham's Matrix Computation Toolbox [72], except the elements below the diagonal are halved to ensure GEPP pivots the matrix as intended. All

Table 3.1: Normwise backward error of various solvers for matrices of size 100 000. NaN indicates the solution was invalid due to division by zero or overflow.

Matrix	GEPP	RBT Solver refined	RBT Solver	GENP refined	GENP
`rand`	1.23×10^{-15}	2.97×10^{-17}	6.43×10^{-12}	2.67×10^{-17}	4.10×10^{-12}
`rands`	2.39×10^{-15}	2.93×10^{-17}	1.53×10^{-11}	1.19×10^{-15}	8.76×10^{-11}
`randn`	1.77×10^{-15}	3.29×10^{-17}	6.68×10^{-12}	3.23×10^{-17}	1.71×10^{-11}
`randb`	1.82×10^{-15}	2.25×10^{-17}	6.15×10^{-12}	NaN	NaN
`fielder`	3.03×10^{-18}	2.92×10^{-19}	1.73×10^{-17}	NaN	NaN
`orthog`	2.29×10^{-16}	9.19×10^{-3}	1.00×10^{-2}	1.21×10^{-1}	1.30×10^{-1}
`gfpp`	NaN	2.79×10^{-19}	5.06×10^{-18}	NaN	NaN

problems had a right-hand side selected from a normal distribution with mean 0 and standard deviation 1.

As Table 3.1 shows, the RBT-based solver was able to solve all but one problem, including all of the problems solvable by GENP and the example of exponential growth for GEPP. The one matrix that the RBT-based solver had poor accuracy was the `orthog` matrix. This matrix is constructed by setting the i, j element to $\sqrt{2/(n+1)}\sin\left(ij\pi/(n+1)\right)$, which makes it orthogonal and symmetric. Previous work showed success on this matrix for a size of 1024 [16] but not for a size of 30 000 [41] when using an RBT depth of 2 and 1 step of iterative refinement. To understand the behavior of this matrix as the problem size grows, I plotted the backward error, $\|r\|_1/(\|A\|_1\|x\|_1)$, for varying problem sizes and RBT depths, but otherwise as the first test, in Fig. 3.4 For the smaller depths tested, there appears to be a problem size at which the depth loses effectiveness. However, depths of 4, 5, and 6 all started losing accuracy at a similar point, which complicates the situation. I believe the sine-based structure of this matrix causes the difficulties but have not been able to quantify the source of the issue.

Performance Results

Next, I tested the performance of the RBT solver in comparison to GEPP and GENP. Because of the increase in the complexity of inter-process communication, the RBT solver was tested on two sets of sizes. First are problem sizes that allow the RBT communication to be done as whole tiles, i.e., those that are multiples of $22\,528 = 512 \times 4 \times 11$. This first set of problem sizes was also used for the GEPP and GENP performance. Second is problem sizes that are smaller by 416 elements (half a tile width) than the first set of problem sizes. Figure 3.5 presents the performance in both TFLOP/s and seconds. This performance is computed from the mean time to solution of 3 executions and a flop count of $\frac{2}{3}n^3$. Additionally, 99.9 % confidence intervals are also included in the figure but are less than 1 TFLOP/s in each direction for all but one case. Finally, the RBT-based solver was also tested without iterative refinement to better understand the sources of overhead.

The RBT solver was successfully able to outperform GEPP. For problems of size greater than 200 000, the tile-aligned RBT solver was between 1.40 and 1.82 times faster than the best case of GEPP, and the non-tile-aligned solver was between 1.06 and 1.36 times faster. For reference, GENP was between 2.23 and 2.73 times faster than the best case of GEPP for those problem sizes, and the best case of GEPP was about twice as fast as GEPP on `rand`.

Next, consider the jagged performance of the RBT-based results. For the tile-aligned problems, the fourth, eighth, and twelfth sizes were all aligned such that no communication

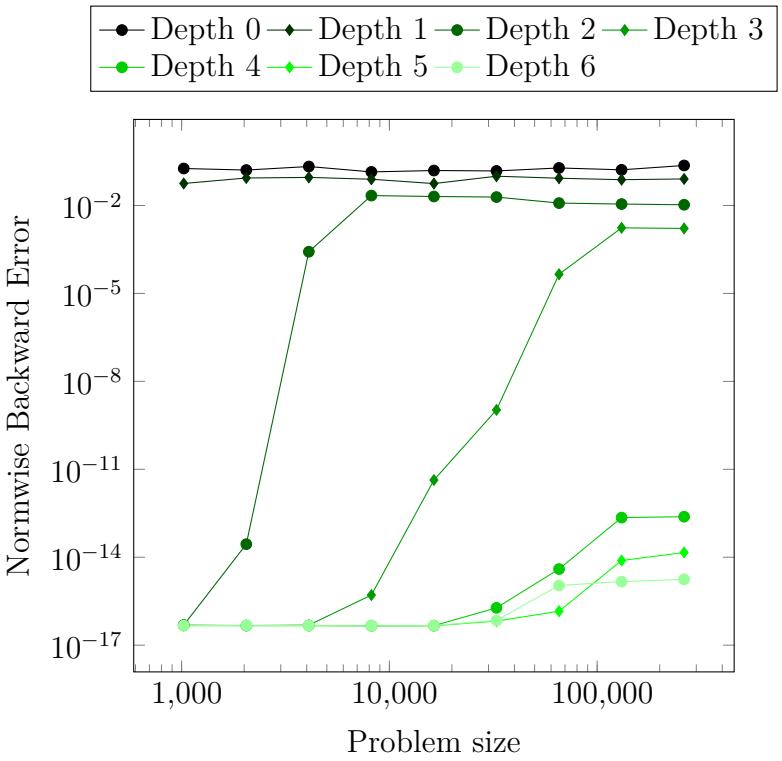


Figure 3.4: Accuracy of the RBT-based solver for various sizes of the `orthog` and various RBT depths. © 2020 IEEE

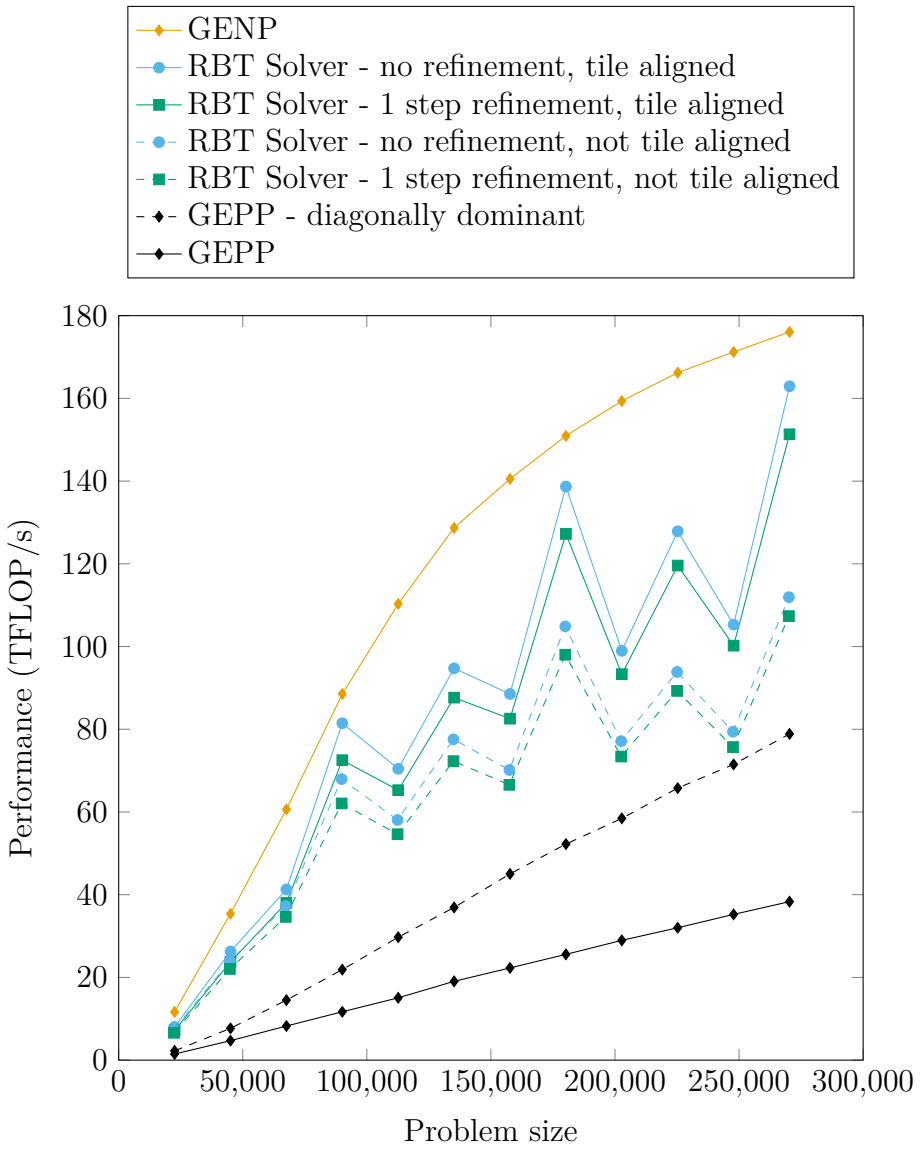


Figure 3.5: Performance of RBT compared to GENP and GEPP.

was needed for RBT application. The RBT-based solver without refinement almost reached the performance of GENP at these sizes with other sizes having worse performance. This indicates that inter-process communication causes most of the cost to apply the transforms. So, optimization efforts of the transform application should focus on inter-process communication. By comparison, iterative refinement introduced a much smaller overhead.

3.4 LU with Additive Modifications

The growth factor bound (3.3) suggests that is desirable to monitor the singular values of the leading principal submatrices to prevent large element growth. Unfortunately, computing these singular values is prohibitively expensive. So instead, consider the Schur complements of GENP. Factoring the kth diagonal, $A[k,k]$, updates each $A[i,j]$ in the trailing matrix by

$$A[i,j] \leftarrow A[i,j] - A[i,k]\,A[k,k]^{-1}\,A[k,j] \quad (k < i,j \leq n).$$

Thus, small $A[k,k]$ entries can result in significant element growth, which in turn can lead to a large backward error [71]. Diagonal blocks with small singular values behave analogously. To prevent this growth, I propose monitoring the singular values of the diagonal blocks and modifying those having values below a predefined threshold. These modifications give rise to a perturbed system with better numerical properties than the original one. The perturbation can then be corrected collectively with the Woodbury formula[3] [70,135] or iterative refinement.

3.4.1 Additive Modifications Algorithm

The core idea of this approach is to apply additive modifications during the factorization when small entries occur on the diagonal instead of exchanging rows. A straightforward way to do this is to perform the classic non-pivoted LU factorization and modify diagonal entries whenever they dip below a preset tolerance. However, this results in a myopic view of the matrix; the issues with one diagonal element can often be fixed using just the next row, for example in a matrix where the leading 2-by-2 diagonal submatrix is $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$. Thus, I use a block LU factorization where the diagonal blocks are factored with the singular value decomposition (SVD). Then, I modify the singular values that are too small. However, the SVD requires significantly more computation than an LU decomposition: $21n^3$ operations

[3] This formula has a variety of names, including the Bartlett-Sherman-Morrison-Woodbury formula and the Sherman-Morrison-Woodbury formula. Hagar's expository paper provides a history of the formula and its repeated discovery [70].

instead of $2/3n^3$—a $30\times$ difference [58]. This limits the block size, n_b, that can be used without introducing significant overhead. Other rank-revealing factorizations, such as QR with column pivoting, require significantly fewer operations; however, the SVD is a more robust factorization, which helped me focus on the effects of the overall block-wise factorization and the additive modifications. Section 3.4.6 explores the use of alternative factorizations. Note that by using the SVD in this way, the final decomposition is not a regular LU factorization (unless $n_b = 1$) but a decomposition into lower and upper *block*-triangular matrices.

Additive modifications commute naturally with the preceding Schur complement updates of the block LU factorization via the commutativity of matrix addition. Hence, the modified LU factors are equivalent to the factors produced by applying all modifications before beginning the factorization (ignoring the effects of numerical round-off). This is analogous to using multiplication by a permutation matrix, P, to represent row pivoting: $\widetilde{A} \equiv PA$. Thus, Block elimination with additive modifications (BEAM) factorizes A into

$$\widetilde{L}\widetilde{R} = \widetilde{A} \equiv A + M_U M_\Sigma M_V^T \tag{3.6}$$

where $\widetilde{L}$ and $\widetilde{R}$ are lower and upper block-triangular matrices, respectively, while M_Σ is a diagonal matrix containing the modifications. Note that I denote the upper block-triangular factor as $\widetilde{R}$ ("right") instead of the usual $\widetilde{U}$ ("upper") to avoid confusion with the U factor of the SVD. The columns of M_U and M_V are the left and right singular vectors corresponding to the modifications in M_Σ and padded with zeros to match the size of A. Thus, M_U and M_V are tall-and-narrow matrices whose columns are a subset of a block-diagonal matrix. Figure 3.6 visualizes these sub-matrix structures.

Because of the perturbations, the factored matrix $\widetilde{A}^{-1}$ often only provides the solution to a nearby system, so a correction is needed to obtain the solution to the original system. I considered two approaches for this correction: iterative refinement and the Woodbury formula. While the former has a well-established formulation, the latter can take various forms. The most general form of the Woodbury formula is

$$(A - BCD)^{-1} = A^{-1} + A^{-1}B(C^{-1} - DA^{-1}B)^{-1}DA^{-1}, \tag{3.7}$$

although a simplified form is often used where $C \equiv I$ [70]. The term $C^{-1} - DA^{-1}B$ is called a *capacitance matrix*, and its inverse is the centerpiece of the Woodbury formula. Here, the correction is formulated as

$$(A - M_U M_\Sigma M_V^T)^{-1} = A^{-1} + A^{-1}M_U(I - M_\Sigma M_V^T A^{-1} M_U)^{-1} M_\Sigma M_V^T A^{-1}$$

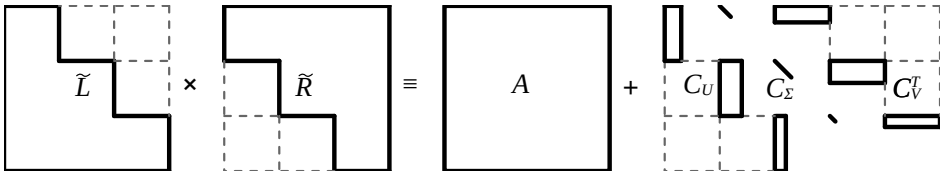


Figure 3.6: 3×3 block structure of BEAM factorization.

instead of the more obvious

$$(A - M_U M_\Sigma M_V^T)^{-1} = A^{-1} + A^{-1} M_U (M_\Sigma^{-1} - M_V^T A^{-1} M_U)^{-1} M_V^T A^{-1}$$

to avoid the need to invert the possibly ill-conditioned M_Σ and to improve the conditioning of the entire capacitance matrix. Section 3.4.2 discusses these numerical properties further.

Algorithm 3.4 outlines the BEAM method. While the algorithm is described with a fixed block size, n_b, it can easily be extended to a variable block size. In lines 6–15, BEAM decomposes the diagonal block and applies any necessary modifications. Then, lines 16–19 proceed as per a regular blocked, non-pivoted LU factorization. Finally, BEAM computes and factors the capacitance matrix if the Woodbury formula is needed. While computing the capacitance matrix, I form and save the $\mathcal{C}_R$ and $\mathcal{C}_L$ matrices; this reduces memory accesses at the cost of a slight increase in storage unless there are numerous modifications. In spite of the factored capacitance matrix being denoted $\mathcal{C}^{-1}$, the inverse should not be formed explicitly; instead, the factored form is preferable for numerical accuracy. I use GEPP to factor this second matrix, but other methods could also work. The solve step simply applies the block-triangular factors and possibly the Woodbury formula.

A key advantage of Algorithm 3.4 comes from the fact that it has the high-level structure of a non-pivoted, block LU. Such structure provides more parallelism than partial pivoting because the panel of L and panel of R can be updated simultaneously [41, 93]. Furthermore, it allows the trailing matrix update from one iteration to overlap with the panel updates from a subsequent iteration.

To outperform GEPP, the overhead introduced by BEAM must be lower than that of pivoting. To that end, I count the number of arithmetic operations used in the modifications and Woodbury formula. Let n be the size of the system, m be the rank of the Woodbury correction, n_b be the size of the diagonal blocks, and ℓ_{rhs} be the number of right-hand sides. (If the Woodbury formula is not applied, $m = 0$.) Because the factors' diagonal blocks are full instead of triangular, computing the Schur complement takes an extra $n^2 n_b + \mathcal{O}(n n_b^2)$ FLOP. Thus, BEAM without the Woodbury correction takes

$$\tfrac{2}{3}n^3 + 2n^2\ell_{\text{rhs}} + n^2 n_b + \mathcal{O}(n n_b^2 + n n_b \ell_{\text{rhs}}) \text{ FLOP.}$$

Next, building and factoring the capacitance matrix (via GEPP) takes an additional $2n^2 m + 2nm^2 + \frac{2}{3}m^3 + \mathcal{O}(nm)$ FLOP. Finally, the Woodbury formula requires two triangular solves and two matrix multiplies. So, the Woodbury formula adds an extra

$$2n^2 m + 2nm^2 + \tfrac{2}{3}m^3 + 4nm\ell_{\text{rhs}} + 2m^2\ell_{\text{rhs}} + \mathcal{O}(n^2 + n\ell_{\text{rhs}}) \text{ FLOP.}$$

procedure FACTORBEAM(A, τ)
  $n_t \leftarrow n/n_b$ ▷ number of blocks in A
  $m \leftarrow 0$ ▷ number of modifications applied
  $A^{(0)} \leftarrow A$
  for $k = 1 : n_t$ **do**
    $U_k, \Sigma_k, V_k^T \leftarrow \mathsf{SVD}(A_{k,k}^{(k-1)})$
    for $i = 1 : n_b$ **do**
      if $\Sigma_k[i] \leq \tau$ **then** ▷ is σ_i below tolerance τ
        $m \leftarrow m + 1$ ▷ Record modification
        $M_\Sigma[m, m] \leftarrow \tau - \Sigma_k[i]$
        $M_U[:, m] \leftarrow [0, U_k[:, i]^T, 0]^T$
        $M_V[:, m] \leftarrow [0, V_k[:, i]^T, 0]^T$
        $\Sigma_k[i] \leftarrow \tau$ ▷ Apply modification
    $\widetilde{L}_{k,k} \leftarrow U_k$
    $\widetilde{R}_{k,k} \leftarrow \Sigma_k V_k^T$
    $\mathcal{I} \leftarrow \{k+1, k+2, \ldots, n_t\}$ ▷ trailing matrix indices
    $\widetilde{L}_{\mathcal{I},k} \leftarrow A_{\mathcal{I},k}^{(k-1)} \widetilde{R}_{k,k}^{-1}$
    $\widetilde{R}_{k,\mathcal{I}} \leftarrow \widetilde{L}_{k,k}^{-1} A_{k,\mathcal{I}}^{(k-1)}$
    $A_{\mathcal{I},\mathcal{I}}^{(k)} \leftarrow A_{\mathcal{I},\mathcal{I}}^{(k-1)} - \widetilde{L}_{\mathcal{I},k} \widetilde{R}_{k,\mathcal{I}}$
  if $m > 0$ and using Woodbury formula **then**
    $\mathcal{C}_R \leftarrow M_\Sigma M_V^T \widetilde{R}^{-1}$
    $\mathcal{C}_L \leftarrow \widetilde{L}^{-1} M_U$
    $\mathcal{C} \leftarrow I - \mathcal{C}_R \mathcal{C}_L$
    $\mathcal{C}^{-1} \leftarrow \mathsf{FACTOR}(\mathcal{C})$ ▷ Using, e.g., GEPP
procedure SOLVEBEAM(b)
  $x \leftarrow \widetilde{L}^{-1} b$
  if $m > 0$ and using Woodbury formula **then**
    $x \leftarrow (I + \mathcal{C}_L \mathcal{C}^{-1} \mathcal{C}_R) x$
  $x \leftarrow \widetilde{R}^{-1} x$

Algorithm 3.4: BEAM algorithm's factor and solve steps. Subscripts for $A, \widetilde{L}, \widetilde{R}$ denote submatrices in terms of matrix blocks, and n_b denotes the block size.

Hence, if $n_b, m \ll n$, the arithmetic overhead compared to GENP should be negligible. While this does not measure the cost of data movement, most of the added computations have high data locality, especially compared to pivoting.

3.4.2 Theoretical Analysis of Key Condition Numbers

Because BEAM inverts $\widetilde{A}$ instead of A, it is crucial to understand the errors that arise when solving $\widetilde{A}x = b$ in finite precision. First, the additive modifications guarantee that $\widetilde{A}$ and its block principal leading submatrices are nonsingular, a prerequisite for the success of non-pivoted block-LU factorizations. Thus, this algorithm (without the Woodbury formula) computes a solution, $\widehat{x}$, that satisfies a nearby system,

$$(\widetilde{A} + \Delta\widetilde{A})\widehat{x} = (b + \Delta b), \tag{3.8}$$

with

$$\|\Delta\widetilde{A}\|_2 \leq \widetilde{\eta_2}(\widehat{x})\|\widetilde{A}\|_2 \quad \text{and} \quad \|\Delta b\|_2 \leq \widetilde{\eta_2}(\widehat{x})\|b\|_2; \tag{3.9}$$

that is, $\widetilde{\eta_2}(\widehat{x})$ is the normwise backward error for the spectral norm. Then, combining (3.8) with the definition of $\widetilde{A}$ from (3.6) gives

$$(A + M_U M_\Sigma M_V^T + \Delta\widetilde{A})\widehat{x} = (b + \Delta b).$$

Thus, the backward error of $\widetilde{x}$ for the original system $Ax = b$ is

$$\eta_2(\widehat{x}) \leq \max\left(\frac{\|M_U M_\Sigma M_V^T + \Delta\widetilde{A}\|_2}{\|A\|_2}, \frac{\|\Delta b\|_2}{\|b\|_2}\right) \leq \hat{\tau} + \widetilde{\eta_2}(\widehat{x}). \tag{3.10}$$

Hence, the forward error of $\widetilde{x}$ can be bounded with only $\hat{\tau}$, the backward stability of the block factorization, and the condition number of A. Importantly, $\widetilde{A}$ need not be well conditioned. Furthermore, the convergence of iterative refinement is also ensured when those three values are sufficiently small [33]. Note that (3.10) implies that $\hat{\tau}$ may directly contribute to the backward error when there are modifications but the Woodbury formula is not applied.

Unfortunately, an ill-conditioned $\widetilde{A}$ can still be problematic when using the Woodbury formula because its forward error directly perturbs the capacitance matrix and, thus, the correction. To help understand this condition number, I provide the following theorem, which states that if the tolerance, $\hat{\tau}$, is small relative to the reciprocal condition number of A, the conditioning of $\widetilde{A}$ will be close to that of A. Interestingly, the condition of this theorem

$(\hat{\tau}\kappa_2(A) \ll 1)$ appears related to the requirement implied by (3.10) for the solution to have any digits of accuracy $(\hat{\tau}\kappa_2(A) + \widetilde{\eta_2}\kappa_2(A) \ll 1)$.

Theorem 3.1. *Let $\widetilde{A} = A + M_U M_\Sigma M_V^T$ where M_U, M_V each have orthonormal columns, and M_Σ is a diagonal matrix with positive entries of at most $\tau = \hat{\tau}\|A\|_2$. If $\hat{\tau}\kappa_2(A) < 1$, then*

$$\kappa_2(\widetilde{A}) \leq \frac{\sigma_1(A) + \tau}{\sigma_n(A) - \tau} = \kappa_2(A)\frac{1 + \hat{\tau}}{1 - \hat{\tau}\kappa_2(A)}$$

where $\sigma_1(A)$ and $\sigma_n(A)$ denote the largest and smallest singular values, respectively, and $\kappa_2(A) = \sigma_1(A)/\sigma_n(A)$.

Proof. By the triangle inequality,

$$\sigma_1(\widetilde{A}) \leq \sigma_1(A) + \tau \quad \text{and} \quad \sigma_n(\widetilde{A}) \geq \sigma_n(A) - \tau.$$

Suppose $\hat{\tau}\kappa_2(A) < 1$, which implies $\sigma_n(A) - \tau > 0$. Hence,

$$\kappa_2(\widetilde{A}) \leq \frac{\sigma_1(A) + \tau}{\sigma_n(A) - \tau} = \frac{\sigma_1(A) + \hat{\tau}\sigma_1(A)}{\sigma_n(A) - \hat{\tau}\sigma_1(A)} = \kappa_2(A)\frac{1 + \hat{\tau}}{1 - \hat{\tau}\kappa_2(A)}. \qquad \square$$

When applying the Woodbury formula, BEAM must also invert the capacitance matrix as per (3.7). Thus, its condition number is also crucial in the analysis of this method. I start by generalizing a lemma of Yip [137] to the full version of the Woodbury formula.

Lemma 3.2. *Let $\|\cdot\|_p$ be any sub-multiplicative matrix norm. Denote the condition number with respect to the Moore-Penrose pseudoinverse by $\kappa_p^+(A) = \|A\|_p\|A^+\|_p$. Suppose that $\widetilde{A} = A + U\Sigma V^T$ is nonsingular with U, V having full column rank and Σ being nonsingular. Then,*

$$\begin{aligned}\kappa_p(\Sigma^{-1} - V^T\widetilde{A}^{-1}U) &\leq \min\left(\kappa_p^+(U)^2, \kappa_p^+(V^T)^2\right)\kappa_p(\Sigma)\kappa_p(A\widetilde{A}^{-1}) \\ &\leq \min\left(\kappa_p^+(U)^2, \kappa_p^+(V^T)^2\right)\kappa_p(\Sigma)\kappa_p(A)\kappa_p(\widetilde{A}).\end{aligned}$$

Proof. To bound the norm of the capacitance matrix, we start by rewriting its expression. Multiplying $\widetilde{A} - U\Sigma V^T = A$ on the left by $\Sigma^{-1}U^+$ and on the right by $\widetilde{A}^{-1}U$ gives

$$(\Sigma^{-1} - V^T\widetilde{A}^{-1}U) = \Sigma^{-1}U^+A\widetilde{A}^{-1}U. \tag{3.11}$$

We next seek a similar expression for its inverse. Note that,

$$A\widetilde{A}^{-1}U = (\widetilde{A} - U\Sigma V^T)\widetilde{A}^{-1}U = U - U\Sigma V^T\widetilde{A}^{-1}U.$$

So, the columns of $A\widetilde{A}^{-1}U$ are within the column space of U. Since UU^+ is an orthogonal projector onto that space [64], we have $(UU^+)A\widetilde{A}^{-1}U = A\widetilde{A}^{-1}U$. Using this, we can verify that

$$(U^+\widetilde{A}A^{-1}U\Sigma)(\Sigma^{-1}U^+A\widetilde{A}^{-1}U) = I.$$

Combining this with (3.11) gives the desired inverse:

$$(\Sigma^{-1} - V^T\widetilde{A}^{-1}U)^{-1} = U^+\widetilde{A}A^{-1}U\Sigma.$$

Hence, the condition number can be bounded as

$$\begin{aligned}\kappa_p(\Sigma^{-1} - V^T\widetilde{A}^{-1}U) &= \|\Sigma^{-1}U^+A\widetilde{A}^{-1}U\|_p\|U^+\widetilde{A}A^{-1}U\Sigma\|_p \\ &\leq \kappa_p^+(U)^2\kappa_p(\Sigma)\|A\widetilde{A}^{-1}\|_p\|\widetilde{A}A^{-1}\|_p.\end{aligned}$$

A similar argument shows that

$$\kappa_p(\Sigma - U\widetilde{A}^{-1}V^T) \leq \kappa_p^+(V^T)^2\kappa_p(\Sigma)\|A\widetilde{A}^{-1}\|_p\|\widetilde{A}A^{-1}\|_p. \qquad \square$$

Using this lemma, the condition number for the capacitance matrix in the obvious form of the Woodbury formula is bounded by

$$\kappa_2(M_\Sigma^{-1} - M_V^T\widetilde{A}^{-1}M_U) \leq \kappa_2(M_\Sigma)\kappa_2(A\widetilde{A}^{-1}).$$

As mentioned in Section 3.4.1, I instead formulate the Woodbury correction to get a tighter bound on conditioning:

$$\kappa_2(I - M_\Sigma M_V^T\widetilde{A}^{-1}M_U) \leq \kappa_2(A\widetilde{A}^{-1}).$$

The conditioning of this latter matrix can be further improved, particularly for the 2-norm. The following theorem shows that if neither A nor $\widetilde{A}$ is ill-conditioned, the capacitance matrix will have an excellent condition number.

Theorem 3.3. *Suppose $\widetilde{A} = A + M_U M_\Sigma M_V^T$ where M_U and M_V each have orthonormal columns, and $\|M_\Sigma\|_2 = \tau$. Additionally, let $\mathcal{C} = I - M_\Sigma M_V^T\widetilde{A}^{-1}M_U$. Then,*

$$\kappa_2(\mathcal{C}) \leq (1 + \tau\|\widetilde{A}^{-1}\|_2)(1 + \tau\|A^{-1}\|_2).$$

If $\tau = \hat{\tau}\|A\|_2$ with $\hat{\tau} < 1$, then the bound simplifies to

$$\kappa_2(\mathcal{C}) \leq (1 + \tfrac{\hat{\tau}}{1-\hat{\tau}}\kappa_2(\widetilde{A}))(1 + \hat{\tau}\kappa_2(A)).$$

Proof. After substituting $U = M_U$, $\Sigma = I$, and $V^T = M_\Sigma M_V^T$, Theorem 3.2 gives the bound $\kappa_2(\mathcal{C}) \leq \|A\widetilde{A}^{-1}\|_2\|\widetilde{A}A^{-1}\|_2$. Since $A = \widetilde{A} - M_U M_\Sigma M_V^T$ and $\|M_U M_\Sigma M_V^T\|_2 = \tau$, a little algebra shows that

$$\kappa_2(\mathcal{C}) \leq (1 + \tau\|\widetilde{A}^{-1}\|_2)(1 + \tau\|A^{-1}\|_2).$$

Suppose $\tau = \hat{\tau}\|A\|_2$ and $\hat{\tau} < 1$. Then,

$$\|A\|_2 = \|\widetilde{A} - M_U M_\Sigma M_V^T\|_2 \leq \|\widetilde{A}\|_2 + \hat{\tau}\|A\|_2,$$

and so $\|A\|_2 \leq (1-\hat{\tau})^{-1}\|\widetilde{A}\|_2$. Therefore,

$$\begin{aligned}\kappa_2(\mathcal{C}) &\leq (1 + \hat{\tau}\|A\|_2\|\widetilde{A}^{-1}\|_2)(1 + \hat{\tau}\|A\|_2\|A^{-1}\|_2) \\ &\leq (1 + \tfrac{\hat{\tau}}{1-\hat{\tau}}\kappa_2(\widetilde{A}))(1 + \hat{\tau}\kappa_2(A)).\end{aligned}$$

□

3.4.3 Error Analysis of Block LU

After the analysis of Section 3.4.2, one critical concern remains: how backward stable is the factorization of $\widetilde{A}$? To my knowledge, no existing analysis of block LU applies to Algorithm 3.4. The closest is by Demmel, Higham, and Schreiber [39], but the block LU they analyzed differs from my factorization in two primary ways. First, the diagonal blocks are factored with GEPP instead of the SVD. Second, the diagonal blocks of the lower block-triangular factor are identity matrices instead of singular vectors. Towards that end, I provide a new analysis of block LU general enough to apply to BEAM.

Because the analysis of block LU relies heavily on norm properties, I first review some key definitions. While matrix norms are (strictly speaking) only defined for a single matrix size, they are often treated as being independent of the matrix size, so that padding a matrix with zeros does not affect its norm. For clarity, I refer to this as a *family of norms*. Formally, for any partitioning of the matrix with $\mathcal{I}, \mathcal{J}$ being the row and column indices, respectively, of the blocks, let the norms of a given family satisfy

$$\max_{\substack{i\in\mathcal{I}\\ j\in\mathcal{J}}}\|A_{i,j}\| \leq \|A\| \leq \sum_{\substack{i\in\mathcal{I}\\ j\in\mathcal{J}}}\|A_{i,j}\|. \tag{3.12}$$

This is satisfied by all of the usual norms, including those induced by vector ℓ_p norms, the Schatten norms, and the element-wise norms. A matrix norm, $\|\cdot\|_\alpha$, is submultiplicative if $\|AB\|_\alpha \leq \|A\|_\alpha \|B\|_\alpha$ for all conformal matrices A and B. While some authors limit the term "matrix norm" to submultiplicative norms, I make no such limitation, due to the use of the max-norm in previous analyses [39, 133]. Submultiplicative norms include all operator norms and the Frobenius norm. A matrix norm, $\|\cdot\|_\alpha$, is absolute if $\|A\|_\alpha = \||A|\|_\alpha$ for any matrix A. Absolute norms include the max-norm, the 1- and ∞-operator norms, and the Frobenius norm.

For the sake of generality, I define the blocks based on the list of the global (pointwise) row/column for which each block starts, called $\mathcal{I}$. For notational convenience, $n+1$ is added as the last element. For example, a fixed block size, n_b, with $n_t = \lceil n/n_b \rceil$ blocks gives

$$\mathcal{I} = [1,\, n_b + 1,\, 2n_b + 1,\, \ldots,\, (n_t - 1)n_b + 1,\, n + 1].$$

For simplicity, I index blocks of a matrix with subscripts, as exemplified by

$$A_{i,j:k} = A[\mathcal{I}_i : \mathcal{I}_{i+1} - 1,\, \mathcal{I}_j : \mathcal{I}_{k-1} - 1]$$

where $\mathcal{I}_i$ is the ith element of $\mathcal{I}$. Then, block LU factorization is described by Algorithm 3.5; this is the core factorization of Algorithm 3.4 but is extracted for clarity. Note that to simplify notation, I index the blocks of $A^{(k)}$ from k to n_t instead of 1 to $n_t - k + 1$. Many basic properties of pointwise LU have blockwise analogs, as outlined by Theorem 3.4.

Lemma 3.4. *Consider the execution of Algorithm 3.5, assuming that a diagonal block can be factored if and only if the block is nonsingular. Note that $A^{(k+1)}$ is only defined if the first k iterations succeed. Then:*

1. $A^{(k+1)} = A/A_{1:k,1:k}$.
2. *The first k iterations of the factorization succeed if and only if the first k block leading principle submatrices $(A_{1,1}, A_{1:2,1:2}, \ldots, A_{1:k,1:k})$ are all nonsingular.*

Note that / (when applied to matrices) denotes the Schur complement.

Proof. The proofs all follow their pointwise equivalents. First, note that combining the steps of iteration k gives

$$A^{(k+1)} = A^{(k)}_{k+1:n_t,k+1:n_t} - A^{(k)}_{k+1:n_t,k}\left(A^{(k)}_{1:k,1:k}\right)^{-1} A^{(k)}_{k,k+1:n_t} = A^{(k)}/A^{(k)}_{k,k}$$

which is useful for proving both results.

1: $n_t \leftarrow$ the size of $\mathcal{I}$
2: $A^{(1)} \leftarrow A$
3: **for** $k = 1 : n_t$ **do**
4: $\quad L_{k,k} R_{k,k} \leftarrow A^{(k)}_{k,k}$
5: $\quad L_{k+1:n_t,k} \leftarrow A^{(k)}_{k+1:n_t,k}\, R^{-1}_{k,k}$
6: $\quad R_{k,k+1:n_t} \leftarrow L^{-1}_{k,k}\, A^{(k)}_{k,k+1:n_t}$
7: $\quad A^{(k+1)}_{k+1:n_t,k+1:n_t} \leftarrow A^{(k)}_{k+1:n_t,k+1:n_t} - L_{k+1:n_t,k}\, R_{k,k+1:n_t}$

Algorithm 3.5: Block LU factorization of A.

1. Assuming that the first k iterations of Algorithm 3.5 succeeded. Iteratively applying the quotient formula [138, Thm. 1.4] gives

$$A^{(k+1)} = A/A_{1:k,1:k}.$$

2. By assumption, the first diagonal block can be factored if and only if it is nonsingular. For the sake of induction, let $k > 1$ and assume that the first k diagonal blocks were factored. Then, $\det(A_{1:k-1,1:k-1}) \neq 0$. The first result implies $A^{(k)}_{k,k} = A_{1:k,1:k}/A_{1:k-1,1:k-1}$. By Schur's determinant formula, $\det(A^{(k)}_{k,k}) = \det(A_{1:k,1:k})/\det(A_{1:k-1,1:k-1})$ [113, pg. 217]. Hence, $A^{(k)}_{k,k}$ is nonsingular if and only if $A_{1:k,1:k}$ is nonsingular. □

A Generalized Measure of Growth

Because of the block structure of the factorization, I found it useful to generalize Wilkinson's traditional growth factor. For any matrix norm, $\|\cdot\|_\alpha$, define

$$\mathrm{P}^{\mathcal{I}}_{\alpha} = \frac{\max_{1\leq k\leq |\mathcal{I}|}\|A^{(k)}\|_\alpha}{\|A\|_\alpha}. \tag{3.13}$$

For simplicity, when every block has size n_b, denote the growth as $\mathrm{P}^{\langle n_b\rangle}_{\alpha}$. Interestingly, the growth is independent of how the diagonal blocks are factored (in exact arithmetic) since it depends only on the Schur complements. Note that Wilkinson's growth factor (i.e., (3.1)) equals $\mathrm{P}^{\langle 1\rangle}_{\max}$ [133] and that Amodio and Mazzia's growth factor equals $\mathrm{P}^{\langle 1\rangle}_{\infty}$ [7]. Barlow and Zha's growth factor is related to, but strictly greater than, $\mathrm{P}^{\langle 1\rangle}_{2}$ [22]; the difference arises from their inclusion of elements from U in the numerator's norm.

Because (3.13) is parameterized for both the blocking and the norm, I bound the relation between different versions of this measure. Importantly, this allows relating the new measurements by that of Wilkinson's classic growth factor:

$$\tfrac{1}{n}\mathrm{P}^{\langle 1\rangle}_{\max} \leq \mathrm{P}^{(n_b)}_{\alpha} \leq n\mathrm{P}^{\langle 1\rangle}_{\max} \qquad \alpha \in \{1, 2, \infty, F\}. \tag{3.14}$$

Theorem 3.5. *Let $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ be matrix norms. Then, the following relations between growth factors hold:*

1. *If $\mathcal{J} \subseteq \mathcal{I}$, then $\mathrm{P}^{\mathcal{J}}_{\alpha} \leq \mathrm{P}^{\mathcal{I}}_{\alpha}$.*
2. *If $\mu^{-1}\|A\|_\alpha \leq \|A\|_\beta \leq \nu\|A\|_\alpha$ for any square matrix A, then $(\mu\nu)^{-1}\mathrm{P}^{\mathcal{I}}_{\alpha} \leq \mathrm{P}^{\mathcal{I}}_{\beta} \leq \mu\nu\mathrm{P}^{\mathcal{I}}_{\alpha}$.*

Proof. The first result follows from the observation that, because all indices in $\mathcal{J}$ also occur in $\mathcal{I}$, all of the trailing matrices produced by the former must also occur by the latter. In

other words,

$$\mathrm{P}_\alpha^{\mathcal{J}} = \frac{\max_{j\in\mathcal{J}}\|A/A[1:j-1,1:j-1]\|_\alpha}{\|A\|_\alpha} \leq \frac{\max_{i\in\mathcal{I}}\|A/A[1:i-1,1:i-1]\|_\alpha}{\|A\|_\alpha} = \mathrm{P}_\alpha^{\mathcal{I}}$$

The second result is a straightforward substitution of norm equivalencies. □

However, (3.14) may be quite pessimistic for some matrices. So, it is also useful to directly study the growth of (3.13). Similar to (3.3) for the pointwise case, the growth can be bounded relative to the smallest singular values of the leading principal submatrices:

$$\begin{aligned}\mathrm{P}_\alpha &\leq 1 + \max_k \left[\|A_{1:k,1:k}^{-1}\|_\alpha \min\left(\|A_{k+1:n_t,1:k}\|_\alpha, \|A_{1:k,k+1:n_t}\|_\alpha\right)\right] \\ &\leq 1 + \max_k \|A\|_\alpha \|A_{1:k,1:k}^{-1}\|_\alpha. \end{aligned} \tag{3.15}$$

Backward Error

I begin by analyzing the backward error of the factorization itself. Because different inner factorizations may be preferable to the SVD (see Sections 3.4.1 and 3.4.6), parameterized the error of the block-wise operations.

Theorem 3.6. *Let $\|\cdot\|_\alpha$ be a submultiplicative norm. Apply Algorithm 3.5 such that for all $1 \leq k \leq n_t$,*

$$A_{kk}^{(k)} = \widehat{L}_{kk}\widehat{R}_{kk} + E_{11}^{(k)}, \qquad \|E_{11}^{(k)}\|_\alpha \leq c_{11}^{(k)} u \|A_{kk}^{(k)}\|_\alpha, \tag{3.16}$$

$$A_{k\mathcal{T}_k}^{(k)} = \widehat{L}_{kk}\widehat{R}_{k\mathcal{T}_k} + E_{12}^{(k)}, \qquad \|E_{12}^{(k)}\|_\alpha \leq c_{12}^{(k)} u \|\widehat{L}_{kk}\|_\alpha \|\widehat{R}_{k\mathcal{T}_k}\|_\alpha, \tag{3.17}$$

$$A_{\mathcal{T}_k k}^{(k)} = \widehat{L}_{\mathcal{T}_k k}\widehat{R}_{kk} + E_{21}^{(k)}, \qquad \|E_{21}^{(k)}\|_\alpha \leq c_{21}^{(k)} u \|\widehat{L}_{\mathcal{T}_k k}\|_\alpha \|\widehat{R}_{kk}\|_\alpha, \tag{3.18}$$

$$\begin{aligned} A_{\mathcal{T}_k\mathcal{T}_k}^{(k+1)} = A_{\mathcal{T}_k\mathcal{T}_k}^{(k)} - \widehat{L}_{\mathcal{T}_k k}\widehat{R}_{k\mathcal{T}_k} + E_{22}^{(k)}, \quad &\|E_{22}^{(k)}\|_\alpha \leq c_{22A}^{(k)} u \|A_{\mathcal{T}_k\mathcal{T}_k}^{(k)}\|_\alpha \\ &+ c_{22LU}^{(k)} u \|\widehat{L}_{\mathcal{T}_k k}\|_\alpha \|\widehat{R}_{k\mathcal{T}_k}\|_\alpha, \end{aligned} \tag{3.19}$$

where $\mathcal{T}_k = k+1 : n_t$ are the trailing blocks. Then, the computed factorization satisfies

$$\|A - \widehat{L}\widehat{R}\|_\alpha \leq C_A u \mathrm{P}_\alpha \|A\|_\alpha + C_{LU} u \|\widehat{L}\|_\alpha \|\widehat{R}\|_\alpha$$

where

$$C_A \leq \sum_{k=1}^{n_t} c_{11}^{(k)} + \sum_{k=1}^{n_t-1} c_{22A}^{(k)} \qquad \text{and} \qquad C_{LU} \leq \sum_{k=1}^{n_t-1} \left(c_{21}^{(k)} + c_{12}^{(k)} + c_{22LU}^{(k)}\right).$$

Proof. When A is a single block, $\|E\|_\alpha \le c_{11} u \mathrm{P}_\alpha \|A\|_\alpha$. So, assume there are multiple blocks. Writing out the product $A - \widehat{L}R$ block-wise gives

$$\begin{aligned}\|A - \widehat{L}R\|_\alpha &= \|E_{11}^{(1)}\|_\alpha + \|E_{12}^{(1)}\|_\alpha + \|E_{21}^{(1)}\|_\alpha + \|E_{22}^{(1)}\|_\alpha + \|A^{(2)} - \widehat{L}_{\mathcal{T}_1\mathcal{T}_1}\widehat{R}_{\mathcal{T}_1\mathcal{T}_1}\|_\alpha \\ &\le (c_{11}^{(1)} + c_{22A}^{(1)}) u \mathrm{P}_\alpha \|A\|_\alpha + (c_{12}^{(1)} + c_{21}^{(1)} + c_{22LU}^{(1)}) u \|\widehat{L}\|_\alpha \|\widehat{R}\|_\alpha \\ &\quad + \|A^{(2)} - \widehat{L}_{\mathcal{T}_1\mathcal{T}_1}\widehat{R}_{\mathcal{T}_1\mathcal{T}_1}\|_\alpha.\end{aligned}$$

Continuing the iteration gives

$$C_A \le \sum_{k=1}^{n_t} c_{11}^{(k)} + \sum_{k=1}^{n_t-1} c_{22A}^{(k)} \qquad \text{and} \qquad C_{LU} \le \sum_{k=1}^{n_t-1} (c_{12}^{(k)} + c_{21}^{(k)} + c_{22LU}^{(k)}). \qquad \square$$

This theorem can reproduce Amodio and Mazzia's backward error analysis of pointwise, partial pivoted LU [7, (4.1)]. Using the ∞-norm and a block size of 1, we have $c_{11} = c_{12} = 0$, $c_{21} = c_{22A} = c_{22LU} = 1$, and $\mathrm{P}_\infty^{\langle 1 \rangle}$ equaling their measure of growth. Furthermore, we have $\|\widehat{L}\|_\infty \le n$ and $\|\widehat{R}\|_\infty \le \mathrm{P}_\infty^{\langle 1 \rangle} \|A\|_\infty$. Thus, Theorem 3.6 gives

$$A = \widehat{L}\widehat{R} + E \qquad \text{where} \qquad \|E\|_\infty \le (2n^2 - n - 1) u \mathrm{P}_\infty^{\langle 1 \rangle} \|A\|_\infty + \mathcal{O}(u^2), \tag{3.20}$$

which is within a factor of 4 of their bound. Furthermore, because $\mathrm{P}_\infty^{\langle 1 \rangle} \le n \mathrm{P}_{\max}^{\langle 1 \rangle}$ (by Theorem 3.5), (3.20) also gives Wilkinson's classic normwise backward error bound [133, Chap 3, (16.13)]. Note that this change of norms explains why the polynomial is a factor of n smaller in Amodio and Mazzia's error analysis compared to Wilkinson's. Furthermore, redoing Theorem 3.6 and the analysis of pointwise LU for the (non-submultiplicative) max-norm gives

$$\|E\|_{\max} \le \mathcal{O}(n^2) u \mathrm{P}_{\max}^{\langle 1 \rangle} \|A\|_{\max}.$$

which indicates that one factor of n in the cubic polynomial from the classic analysis is due to mixing the ∞-norm with the implicit max-norm. This also helps explain why the cubic polynomial is pessimistic in practice: If $|E|$ has a similar numerical structure to $|A|$, then $\|E\|_\infty / \|E\|_{\max}$ should be close to $\|A\|_\infty / \|A\|_{\max}$, so the conversion factor will usually be less than or approximately 1. Combining this with recent probabilistic analysis [74, Thm. 3.7] provides some justification of the long-known fact that the polynomial is closer to n in practice [133, pg. 108].

Next, I bound the backward error of block-triangular solves. To ensure the bound has the correct structure for any submultiplicative norm, $\|\cdot\|_\alpha$, I introduce the quantity μ_α. This quantity is the smallest value such that for any $u, v \in \mathbb{R}^n$ there is a matrix, M, such that

$u = Mv$ and $\|M\|_\alpha = \|u\|_\alpha / \|v\|_\alpha$. This value trivially equals 1 for Schatten norms (notably the spectral norm and the Frobenius norm). Furthermore, it can be shown that this quantity is 1 for the max-norm and for operator norms induced by ℓ_p vector norms (notably the operator 1- and ∞-norms). However, for some norms, the quantity can be larger than 1. For example, consider the elementwise 1-norm, $\|\cdot\|_{\text{sum}}$, which equals the sum of element magnitudes and is submultiplicative. Then, $\mu_{\text{sum}} = n$.

Theorem 3.7. *Let $\|\cdot\|_\alpha$ be a submultiplicative norm and let*

$$\mu_\alpha = \max_{\substack{u,v \in \mathbb{R}^n \\ v \neq 0}} \min_{\substack{M \in \mathbb{R}^{n \times n} \\ u = Mv}} \frac{\|Mv\|_\alpha}{\|M\|_\alpha \|v\|_\alpha}$$

bound the norm of a mapping between an arbitrary pair of vectors. Assume that $\widehat{L}$ is a lower block-triangular matrix and that

$$(\widehat{L}_{kk} + E_{L1}^{(k)})\widehat{y}_k = b_k^{(k)} \qquad \|E_{L1}^{(k)}\|_\alpha \leq c_{L1}^{(k)} u \|\widehat{L}_{kk}\|_\alpha \tag{3.21}$$

$$b_{\mathcal{T}_k}^{(k+1)} + E_{L2}^{(k)} = b_{\mathcal{T}_k}^{(k)} - \widehat{L}_{\mathcal{T}_k k}\widehat{y}_k \qquad \|E_{L2}^{(k)}\|_\alpha \leq c_{L2}^{(k)} u\Big(\|b_{\mathcal{T}_k}^{(k)}\|_\alpha + \|\widehat{L}_{\mathcal{T}_k k}\|_\alpha \|\widehat{y}_k\|_\alpha\Big) \tag{3.22}$$

with $\mathcal{T}_k = k+1 : n_t$ being the indices of the trailing blocks. Then,

$$(\widehat{L} + E)\widehat{y} = b \qquad \textit{where} \qquad \|E\|_\alpha \leq \mu_\alpha \left(\sum_{k=1}^{n_t} c_{L1}^{(k)} + 2 \sum_{k=1}^{n_t - 1} c_{L2}^{(k)} \right) u \|\widehat{L}\|_\alpha + \mathcal{O}(u^2).$$

An analogous result holds for upper block-triangular matrices.

Proof. Writing out the product $b - \widehat{L}\widehat{y}$ block-wise gives

$$\begin{aligned} \|b - \widehat{L}\widehat{y}\|_\alpha &\leq \|E_1^{(1)}\|_\alpha \|\widehat{y}_1\|_\alpha + \|E_{L2}^{(1)}\|_\alpha + \|b_{2:n_t}^{(2)} - \widehat{L}_{2:n_t, 2:n_t}\widehat{y}_{2:n_t}\|_\alpha \\ &\leq c_{L1}^{(k)} u \|\widehat{L}_{11}\|_\alpha \|\widehat{y}_1\|_\alpha + c_{L2}^{(k)} u (\|b_{\mathcal{T}_1}^{(2)}\|_\alpha + \|\widehat{L}_{\mathcal{T}_1 1}\|_\alpha \|\widehat{y}_k\|_\alpha) + \|b_{\mathcal{T}_1}^{(2)} - \widehat{L}_{\mathcal{T}_1 \mathcal{T}_1}\widehat{y}_{\mathcal{T}_1}\|_\alpha \end{aligned} \tag{3.23}$$

Note that $\|b_{\mathcal{T}_1}^{(2)}\|_\alpha \leq \|b_{\mathcal{T}_1}^{(2)} - \widehat{L}_{\mathcal{T}_1 \mathcal{T}_1}\widehat{y}_{\mathcal{T}_1}\|_\alpha + \|\widehat{L}_{\mathcal{T}_1 \mathcal{T}_1}\widehat{y}_{\mathcal{T}_1}\|_\alpha$. Thus, (3.23) can be simplified to

$$\|b - \widehat{L}\widehat{y}\|_\alpha \leq (c_{L1}^{(k)} + 2c_{L2}^{(k)}) u \|\widehat{L}\|_\alpha \|\widehat{y}\|_\alpha + (1 + c_{L2}^{(k)} u)\|b_{2:n_t}^{(2)} - \widehat{L}_{22}\widehat{y}_{2:n_t}\|_\alpha.$$

Continuing the iteration gives

$$\|b - \widehat{L}\widehat{y}\|_\alpha \leq \left(\sum_{k=1}^{n_t} c_{L1}^{(k)} + 2 \sum_{k=1}^{n_t - 1} c_{L2}^{(k)} \right) u \|\widehat{L}\|_\alpha \|\widehat{y}\|_\alpha + \mathcal{O}(u^2).$$

By the definition of μ_α, there exists an error matrix E such that $E\widehat{y} = b - \widehat{L}\widehat{y}$ and $\|E\|_\alpha\|\widehat{y}\|_\alpha \leq \mu_\alpha\|b - \widehat{L}\widehat{y}\|_\alpha$. Therefore,

$$\|E\|_\alpha \leq \mu_\alpha \left(\sum_{k=1}^{n_t-1} c_{L1}^{(k)} + 2\sum_{k=1}^{n_t} c_{L2}^{(k)}\right) u\|\widehat{L}\|_\alpha + \mathcal{O}(u^2). \qquad \square$$

As with the factorization, Theorem 3.7 is able to recreate the backward error analysis of pointwise triangular solves, albeit with a larger coefficient.

Theorems 3.6 and 3.7 can be combined to bound the backward error of a full, block-wise solve. While Theorem 3.6 can be tightened to use $\||\widehat{L}||\widehat{R}|\|_\alpha$ instead of $\|\widehat{L}\|_\alpha\|\widehat{R}\|_\alpha$, the use of the triangular solves in the following corollary cannot be similarly tightened.

Corollary 3.8. *Let $\|\cdot\|_\alpha$ be a submultiplicative norm Apply Algorithm 3.5 with the assumptions of Theorem 3.6 and Theorem 3.7. Then, there exists a matrix E such that $(A+E)\widehat{x} = b$ with*

$$\begin{aligned}
\|E\|_\alpha &\leq C_A u \mathrm{P}_\alpha\|A\|_\alpha + C_{LU} u\|\widehat{L}\|_\alpha\|\widehat{R}\|_\alpha, \\
C_A &\leq \sum_{k=1}^{n_t} c_{11}^{(k)} + \sum_{k=1}^{n_t-1} c_{22A}^{(k)}, && \textit{and} \\
C_{LU} &\leq \mu_\alpha \sum_{k=1}^{n_t}\left[c_{R1}^{(k)} + c_{L1}^{(k)}\right] + \sum_{k=1}^{n_t-1}\left[c_{21}^{(k)} + c_{12}^{(k)} + c_{22LU}^{(k)} + 2\mu_\alpha\left(c_{L2}^{(k)} + c_{R2}^{(k)}\right)\right].
\end{aligned}$$

These theorems raise the question of whether $\|\widehat{L}\|_\alpha\|\widehat{R}\|_\alpha$ can be bound in terms of $\|A\|_\alpha$. A simple bound is

$$\|\widehat{L}\|_\alpha\|\widehat{R}\|_\alpha \leq \min\left[\kappa_\alpha(L), \kappa_\alpha(R)\right]\|A\|_\alpha + \|E\|_\alpha$$

when the norm is submultiplicative and E is the error from Theorem 3.6. However, the condition numbers of the computed factors provide minimal insight into how the algorithm is affected by its inputs. To improve that bound, I focus on the case where the diagonal blocks of L are orthogonal and the norm is unitarily invariant since that is the primary interest for BEAM. An analogous theorem holds for, e.g., permutation matrices with the 1- or ∞-norms. Note that this bound improves on the bound of Demmel et al. [39, pg. 182], replacing the pointwise growth factor cubed with the blockwise growth factor squared (in addition to applying to a more general formulation of block LU).

Theorem 3.9. *Let $\|\cdot\|_\alpha$ be submultiplicative and unitarily invariant and assume the diagonal blocks of L are all unitary. Then,*

$$\|L\|_\alpha\|R\|_\alpha \leq \left(n + n_t\mathrm{P}_\alpha\kappa_\alpha(A)\right)n_t\mathrm{P}_\alpha\|A\|_\alpha.$$

Proof. Bounds for both of the factors follow the work of Demmel et al. [39, pg. 182]. First, consider $\|R\|_\alpha$. Note that $\|R_{1,2:n_t}\|_\alpha = \|L_{11}^{-1}A_{1,2:n_t}\|_\alpha \leq \mathrm{P}_\alpha\|A\|_\alpha$. Then,

$$\|R\|_\alpha \leq \|R_{1,1:n_t}\|_\alpha + \|R_{2:n_t,2:n_t}\|_\alpha = \mathrm{P}_\alpha\|A\|_\alpha + \|R_{2:n_t,2:n_t}\|_\alpha. \tag{3.24}$$

Iterating this process on the trailing matrices gives $\|R\|_\alpha \leq n_t\mathrm{P}_\alpha\|A\|_\alpha$.

Next, consider $\|L\|_\alpha$. Starting similarly, we have

$$\|L_{k+1:n_t,k}\|_\alpha = \|A^{(k)}_{k+1:n_t,k}R_{kk}^{-1}\|_\alpha = \|A^{(k)}_{k+1:n_t,k}(A^{(k)}_{kk})^{-1}L_{kk}\|_\alpha = \|A^{(k)}_{k+1:n_t,k}(A^{(k)}_{kk})^{-1}\|_\alpha.$$

Blockwise inversion shows that $-(A^{(k+1)})^{-1}A^{(k)}_{k+1:n_t,k}(A^{(k)}_{kk})^{-1} = (A^{(k)})^{-1}_{k+1:n_t,k}$. So,

$$\|A^{(k)}_{k+1:n_t,k}(A^{(k)}_{kk})^{-1}\|_\alpha \leq \|A^{(k+1)}\|_\alpha\|(A^{(k)})^{-1}_{k+1:n_t,k}\|_\alpha \leq \mathrm{P}_\alpha\|A\|_\alpha\|A^{-1}\|_\alpha \leq \mathrm{P}_\alpha\kappa_\alpha(A).$$

Using the triangle inequality and unitary invariance, the norm of any of the diagonal blocks of L is bounded by the block's size. Thus,

$$\|L\|_\alpha \leq \sum_{k=1}^{n_t}\|L_{k,k}\|_\alpha + \sum_{k=1}^{n_t}\|L_{k+1:n_t,k}\|_\alpha \leq n + n_t\mathrm{P}_\alpha\kappa_\alpha(A)$$

Therefore, $\|L\|_\alpha\|R\|_\alpha \leq \Big(n + n_t\mathrm{P}_\alpha\kappa_\alpha(A)\Big)n_t\mathrm{P}_\alpha\|A\|_\alpha$. □

Theorems 3.6 to 3.8 can also be redone in a component-wise manner to replace the difficult $\|L\|_\alpha\|R\|_\alpha$ term with an easier $|L||R|$ term and to avoid the measure μ_α from Theorem 3.7. Unfortunately, component-wise theory cannot be used when the inner factorization is based on the SVD or QR factorization because those factorizations are not component-wise stable [71, Sec. 19.7].

This analysis shows that block LU is backward stable if the growth is limited and the component block routines are also adequately backward stable. Because the SVD and the QR routines discussed in Section 3.4.6 are backward stable, the growth factor is the primary concern, as in pointwise LU. More analysis is needed to understand how the additive modifications of BEAM affect the growth factor.

3.4.4 Experimental Results

To investigate the numerical stability, scalability, and performance of BEAM, I implemented it in SLATE and evaluated it on the Summit supercomputer. My implementation follows Algorithm 3.4 and uses a high-level structure based on that of SLATE's GENP routine (see Section 3.2). However, I separated BEAM's algorithmic block size from the distribution tile

size (the former always being smaller than the latter in these experiments). For simplicity, my code does not support an algorithmic block to be split across multiple tiles of SLATE and will truncate the last block in a tile to fit. But all the experiments align the block and tile sizes so that truncation only happens in the last tile. After the factorization is complete, the capacitance matrix is built and factored. While my theory defines τ in terms of $\|A\|_2$, this is expensive to compute in practice. So, my experiments instead used the Frobenius norm, $\tau = \hat{\tau}\|A\|_F$, which is closely related.

Because BEAM's factors are blockwise triangular instead of pointwise triangular, I had to implement a GPU routine for batched, block-triangular solves to apply the diagonal tiles of the factors. I used a recursive formulation similar to the MAGMA [2] and KBLAS [35] libraries. Because the diagonal blocks come from the SVD, these inverses can be realized by a matrix multiplication and sometimes a diagonal scaling. While cuBLAS's batched GEMM routine was effective for the trailing-matrix updates, its performance was lacking for small block sizes due to the subsequent copy or scale operation. For such cases, I implemented a custom routine that combined the multiplication and the copy to improve cache reuse and avoid extra kernel launch overheads. To reduce the effort in performance tuning, I used part of MAGMA's matrix-multiplication routine in my kernel.

Experimental Setup

The tester was compiled with GCC 9.1.0, CUDA 11.0.3, IBM Spectrum MPI 10.4.0.3, IBM ESSL 6.1.0, Netlib LAPACK 3.8.0, and Netlib ScaLAPACK 2.1.0. I set the `smt1` flag (disabling simultaneous multithreading) and started MPI with `jsrun -n 16 -a 1 -c 21 -g 3 -b packed:21 -d packed` which allocates 16 processes, each bound to a single socket and its GPUs. For all experiments, I configured the tester with `--origin h --target d --ref n --grid 4x4 --panel-threads 20 --seed 1 --seedB 2 --matrixB randn --nrhs 1`. I also set the `--matrix`, `--dim`, and `--check` flags as appropriate for the experiment. For GEPP, I also set `--nb 768 --ib 64 --lookahead 1`. For GENP and BEAM, I set `--nb 512 --lookahead 2 --ib 64`, except for the experiment described in Section 3.4.4 which changed the last argument as appropriate for BEAM. This configuration gives a 2D block-cyclic distribution with a 4×4 process grid and blocks of size 512 or 768, as indicated by `--nb`. Note that SLATE's `ib` parameter corresponds to the n_b value discussed in this paper; SLATE's `nb` corresponds to the larger blocks used to distribute the matrix.

All tests were preceded by extra tests of size $n = 5000$ (with the otherwise identical configuration) to ensure that the results were not influenced by software initialization costs. To measure the effects of system noise, I ran each performance test three times and computed the

mean and 95% confidence interval. Except for `svd_geo`, the error and number of modifications were the same between the different runs. Due to minor non-determinism in SLATE's QR factorization, there is slight variability between runs in the entries of `svd_geo`. However, this variability is small and does not affect my conclusions or analysis, so I just present the error values and number of modifications from the first run.

To understand how the BEAM algorithm behaves across various linear systems, I used seven random and eight structured matrices in these tests. Table 3.2 describes these matrices. I chose a right-hand side with each element randomly taken from the normal distribution. The matrix generator was always seeded with 1 for the matrices and with 2 for the right-hand sides so that the test problems can be reproduced.

Accuracy was measured with the infinity-norm backward error:

$$\eta_\infty(x) = \frac{\|b - Ax\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty}. \tag{3.25}$$

Correspondingly, iterative refinement was terminated when this error was less than or equal to $\sqrt{n}$ times the unit roundoff ($\sim 3.5 \times 10^{-14}$ when $n = 10^5$) or after 30 iterations. I selected this criterion based on the accuracy of GEPP (see Table 3.3).

Baseline Accuracy and Performance Experiments

First, Table 3.3 compares the accuracy of BEAM against GEPP and GENP for varying values of tolerance $\hat{\tau}$. The matrices were of size 10^5, with a blocking factor of 64 for BEAM. The reported error is the infinity-norm backward error of (3.25). Most importantly, the error of BEAM with the Woodbury correction is smaller than or approximately equal to that of GENP for all but one case (`orthog` with $\hat{\tau} = 10^{-10}$). Furthermore, BEAM with Woodbury correction has a significantly smaller error than GENP for most matrices and only incurs NaN values for one matrix, `zielkeNS`. (Those NaN values resulting from growth-induced overflow and are easily detected in the residual of iterative refinement.) These results demonstrate the ability of BEAM to provide better numerical stability than GENP. Moreover, the error was smaller than 10^{-10} for many of the matrices. This implies that the iterative refinement should often converge quickly to double-precision accuracy [33]. While $\hat{\tau} = 10^{-6}$ leads to modifications for most matrices, only five of the fifteen matrices required more than ten modifications when $\hat{\tau} \leq 10^{-8}$ (one of which was accurately solved without any modifications when $\hat{\tau} = 10^{-10}$). This indicates that the proposed approach is likely effective for a large class of matrices. Additionally, many linear systems saw a significant improvement in accuracy compared to GENP, even without modification. Thus, even just applying an SVD factorization to invert the diagonal blocks increases the stability of a non-pivoted factorization.

Table 3.2: Tested Matrices

Name	Description
`rand`	Random elements uniform on $[0, 1]$
`rands`	Random elements uniform on $[-1, 1]$
`randn`	Random elements normally distributed
`randb`	Random elements of 0 or 1
`randr`	Random elements of -1 or 1
`rand+nI`	`rand` plus nI (called `rand_dominant` in SLATE)
`svd_geo`	Random matrix with singular values geometrically spaced from 10^{-8} to 1
`chebspec`	From MATLAB's `gallery` function
`circul`	From MATLAB's `gallery` function
`fiedler`	From MATLAB's `gallery` function
`kms`	From MATLAB's `gallery` function
`orthog`	From MATLAB's `gallery` function
`riemann`	From MATLAB's `gallery` function
`ris`	From MATLAB's `gallery` function
`zielkeNS`	Zielke's non-symmetric matrix ($a = 1$) [141]

Table 3.3: Accuracy of BEAM without iterative refinement compared to GEPP and GENP.

			$\hat{\tau} = 10^{-6}$			$\hat{\tau} = 10^{-8}$			$\hat{\tau} = 10^{-10}$		
Matrix	GEPP Error	GENP Error	# Mods.	Corr. Error	Uncorr. Error	# Mods.	Corr. Error	Uncorr. Error	# Mods.	Corr. Error	Uncorr. Error
`rand`	2×10^{-14}	3×10^{-9}	126	2×10^{-13}	2×10^{-7}	2	2×10^{-12}	4×10^{-11}	0	5×10^{-12}	5×10^{-12}
`rands`	3×10^{-14}	3×10^{-10}	59	7×10^{-13}	2×10^{-7}	0	3×10^{-12}	3×10^{-12}	0	3×10^{-12}	3×10^{-12}
`randn`	4×10^{-14}	3×10^{-10}	57	7×10^{-13}	2×10^{-7}	0	2×10^{-12}	2×10^{-12}	0	2×10^{-12}	2×10^{-12}
`randb`	3×10^{-14}	NaN	89	4×10^{-13}	2×10^{-7}	0	2×10^{-12}	2×10^{-12}	0	2×10^{-12}	2×10^{-12}
`randr`	3×10^{-14}	NaN	60	5×10^{-13}	1×10^{-7}	0	1×10^{-12}	1×10^{-12}	0	1×10^{-12}	1×10^{-12}
`rand+nI`	2×10^{-14}	1×10^{-14}	0	1×10^{-14}	1×10^{-14}	0	1×10^{-14}	1×10^{-14}	0	1×10^{-14}	1×10^{-14}
`svd_geo`	5×10^{-15}	1×10^{-10}	47 424	8×10^{-14}	5×10^{-7}	9 072	2×10^{-13}	4×10^{-9}	127	2×10^{-12}	2×10^{-11}
`chebspec`	3×10^{-16}	8×10^{-10}	3 198	3×10^{-16}	5×10^{-7}	0	2×10^{-16}	2×10^{-16}	0	2×10^{-16}	2×10^{-16}
`circul`	2×10^{-17}	1×10^{-14}	2	9×10^{-16}	5×10^{-7}	0	1×10^{-15}	1×10^{-15}	0	1×10^{-15}	1×10^{-15}
`fiedler`	2×10^{-18}	NaN	98 440	3×10^{-15}	7×10^{-7}	92 188	4×10^{-15}	5×10^{-9}	0	4×10^{-15}	4×10^{-15}
`kms`	2×10^{-16}	2×10^{-16}	0	5×10^{-16}	5×10^{-16}	0	5×10^{-16}	5×10^{-16}	0	5×10^{-16}	5×10^{-16}
`orthog`	3×10^{-15}	5×10^{-5}	47 216	4×10^{-7}	1×10^{-6}	21 420	2×10^{-5}	2×10^{-5}	1 022	6×10^{-4}	3×10^{-4}
`riemann`	2×10^{-14}	5×10^{-13}	43	6×10^{-16}	8×10^{-7}	0	1×10^{-14}	1×10^{-14}	0	1×10^{-14}	1×10^{-14}
`ris`	3×10^{-15}	1×10^{-1}	49 980	3×10^{-9}	3×10^{-5}	49 977	3×10^{-6}	2×10^{-6}	49 973	7×10^{-5}	5×10^{-5}
`zielkeNS`	2×10^{-19}	NaN	1 594	NaN	NaN	1 594	NaN	NaN	1 594	NaN	NaN

Table 3.4: Performance of BEAM (using iterative refinement) compared to GEPP and GENP with 95% confidence intervals.

			$\hat{\tau} = 10^{-6}$				$\hat{\tau} = 10^{-8}$				$\hat{\tau} = 10^{-10}$			
Matrix	GEPP (s)	GENP (s)	Itr.	Corr. (s)	Itr.	Uncorr. (s)	Itr.	Corr. (s)	Itr.	Uncorr. (s)	Itr.	Corr. (s)	Itr.	Uncorr. (s)
`rand`	49.6±0.8	6.5±0.8*	1	13.7±0.8	3	10.9±0.8	1	10.7±0.8	1	10.2±0.8	1	10.2±0.8	1	10.3±0.8
`rands`	49.4±1.4	6.6±1.4*	1	12.2±1.4	3	10.7±1.4	1	10.3±1.4	1	10.3±1.4	1	10.2±1.4	1	10.2±1.4
`randn`	49.6±1.4*	6.5±1.4*	1	12.2±1.4	3	10.7±1.4	1	10.2±1.4	1	10.2±1.4	1	10.2±1.4	1	10.2±1.4
`randb`	49.6±1.1	6.5±1.1*	1	12.8±1.1	4	11.0±1.1	1	10.2±1.1	1	10.2±1.1	1	10.2±1.1	1	10.2±1.1
`randr`	50.0±2.4	6.6±2.4*	1	12.1±2.4	5	11.3±2.4	1	10.2±2.4	1	10.3±2.4	1	10.1±2.4	1	10.2±2.4
`rand+nI`	24.9±0.2	6.5±0.2	0	10.0±0.2	0	10.0±0.2	0	10.0±0.2	0	10.0±0.2	0	10.1±0.2	0	10.0±0.2
`svd_geo`	49.1±2.6	6.6±2.6*	1	44.2±2.6	30	17.9±2.6*	1	24.9±2.6	30	17.9±2.6*	1	13.6±2.6	1	10.2±2.6
`chebspec`	31.3±0.7	6.5±0.7*	0	21.2±0.7	30	17.8±0.7*	0	9.8±0.7	0	9.9±0.7	0	9.9±0.7	0	9.9±0.7
`circul`	31.4±0.9	6.5±0.9	0	10.3±0.9	3	10.5±0.9	0	9.8±0.9	0	9.8±0.9	0	9.8±0.9	0	9.9±0.9
`fiedler`	39.1±0.6	6.6±0.6*	0	76.3±0.6	30	17.7±0.6*	0	71.9±0.6	30	17.7±0.6*	0	9.8±0.6	0	9.8±0.6
`kms`	24.0±0.4	6.6±0.4	0	9.7±0.4	0	9.8±0.4	0	9.7±0.4	0	9.8±0.4	0	9.8±0.4	0	9.8±0.4
`orthog`	50.5±1.2	6.5±1.2*	2	43.9±1.2	2	10.5±1.2	3	30.6±1.2	2	10.5±1.2	24	25.6±1.2	24	16.1±1.2
`riemann`	42.2±0.7	6.6±0.7*	0	11.6±0.7	30	17.8±0.7*	0	9.9±0.7	0	10.0±0.7	0	10.0±0.7	0	10.0±0.7
`ris`	39.0±0.8	6.6±0.8*	1	44.5±0.8	3	10.6±0.8	1	44.6±0.8	2	10.5±0.8	2	45.2±0.8	2	10.5±0.8
`zielkeNS`	38.5±0.6	6.6±0.6*	0	19.8±0.6*	0	9.8±0.6*	0	19.8±0.6*	0	9.8±0.6*	0	19.7±0.6*	0	9.8±0.6*

*Error larger than $2^{-53}\sqrt{n} \approx 3.5 \times 10^{-14}$.

The results become more nuanced when considering BEAM without Woodbury correction. For $\hat{\tau} = 10^{-10}$, the uncorrected solver behaved similarly to the corrected one. For larger tolerances, however, there is a significant gap between the two, particularly for $\hat{\tau} = 10^{-6}$. This indicates that the perturbation of the uncorrected modification becomes the dominant source of error when $\hat{\tau} \gtrsim 10^{-8}$. This aligns with both the $\hat{\tau}$ term in the normwise backward error bound of (3.10) and the recommended tolerance when applying scalar updates without correction [90]. In contrast, when the Woodbury correction was applied, increasing the tolerances always saw similar or better accuracies. This suggests that the error in the corrected case comes from the presence of small diagonal singular values and the resulting growth and not from applying the modifications or the Woodbury correction process.

Table 3.4 augments Table 3.3 by showing the time to solve the linear systems of equations (again with $n = 10^5$). Up to 30 steps of iterative refinement were used for BEAM but not for GEPP or GENP. To clarify where iterative refinement was unsuccessful, I marked the cases which failed to achieve convergence criterion for iterative refinement ($\eta_\infty(x) \lesssim 3.5 \times 10^{-14}$). Furthermore, I provide the number of refinement iterations, with 30 being the limit.

First, note that by using iterative refinement, BEAM achieved an error of less than $2^{-53}\sqrt{n}$ for almost all cases. As above, `zielkeNS`'s failure involved excessive growth generating NaN values. For the remaining failures, BEAM produced a non-NaN solution, but iterative refinement failed to converge to full accuracy. These cases included `svd_geo`, `chebspec`, `fiedler`, and `riemann` with larger tolerances (and many modifications) but without Woodbury correction. These matrices are all ill-conditioned, which limits the ability of iterative refinement to converge when the inner solution is only moderately accurate [33]. For example, $\kappa_\infty(\texttt{fiedler}) = 2n(n-1) \approx 2 \times 10^{10}$ [122, pg. 159], so iterative refinement can only be expected to converge to full accuracy when $\eta_\infty(x) \lesssim 5 \times 10^{-11}$. Furthermore, this further supports the implication of both (3.10) and Theorem 3.1 that $\hat{\tau}$ should be chosen such that $\hat{\tau}\kappa_2(A) \ll 1$. Interestingly, these systems were successfully solved when using the Woodbury formula, despite the dire implication of Theorems 3.1 and 3.3 that $\hat{\tau}\kappa_2(A) \lesssim 1$ can lead to a large forward error. The only other cases with a high iteration count were `orthog` and $\hat{\tau} = 10^{-10}$, with and without Woodbury correction. While this matrix is orthogonal and perfectly conditioned, the factorization is of very low quality, almost certainly due to a large growth factor.

BEAM outperformed GEPP in all cases except `fiedler` and `ris` with many modifications and the Woodbury formula. Furthermore, most cases show a large speedup, particularly when the Woodbury formula was not applied. (Although, for cases that failed to converge, the speedup is, of course, a moot point.) Unfortunately, BEAM had, at best, about two-thirds the performance of GENP. The block factorization seems to be the predominant source of

errors, with iterative refinement only adding a significant overhead when many iterations are applied (cf. Section 3.4.4 and Tables 3.5 and 3.6).

Interestingly, in cases for which it converged, BEAM without Woodbury correction outperformed the corrected version in all cases. Furthermore, when $\hat{\tau} = 10^{-10}$, BEAM without Woodbury correction converged in all but the `zielkeNS` case. Combining this observation with (3.10) and Table 3.3 suggests that the Woodbury formula is unnecessary for small tolerances. Furthermore, comparing the $\hat{\tau} = 10^{-6}$ and $\hat{\tau} = 10^{-10}$ columns of Table 3.4 shows that in all but one case, smaller tolerances give similar or better performance than larger tolerances. The one exception is `orthog` without the Woodbury formula, likely due to the excessive growth.

Effect of Tolerance Choice

I next investigated the tradeoff in performance and accuracy for a larger variety of tolerance values and blocking sizes on select matrices without iterative refinement. Tables 3.5 and 3.6 show the results. Matching Tables 3.3 and 3.4, the matrix sizes are all $n = 10^5$. Furthermore, the number of modifications and error in Tables 3.5 and 3.6 for $\hat{\tau} = 10^{-6}, 10^{-8}, 10^{-10}$ correspond to the values in Table 3.3; however, the run times differ from Table 3.4 due to the omission of iterative refinement. While GEPP and GENP do not have the algorithmic blocking parameter n_b of BEAM, they implement cache-blocking with a similar structure and a block size of 64.

Both `rand+nI` and `rand` matrix types saw BEAM significantly outperform GEPP for all configurations. However, as mentioned earlier, BEAM performed worse than GENP, even when no corrections were applied. Moreover, smaller block sizes performed slightly better, likely due to increased arithmetic for the SVDs and block-triangular solves. For `rand+nI`, all configurations resulted in no modifications and the same accuracy. For `rand`, on the other hand, increasing the tolerance above 10^{-10} increased the accuracy when the Woodbury correction was applied but decreased the accuracy when it was not, with the number of modifications increasing in both cases. Given the number of modifications introduced when $\hat{\tau} \leq 10^{-6}$, a tolerance of 10^{-8} or 10^{-10} is a better choice, particularly when not applying the Woodbury correction. Finally, increasing the block size reduced the number of modifications and increased the accuracy in all but one case.

The structured matrices provided more interesting results. As in the previous tables, BEAM applied numerous modifications to the `orthog` matrix for all of the tested configurations. Increasing the blocking factor helped the accuracy, although it also increased the number of modifications. The best tradeoff between performance and accuracy seems

Table 3.5: Tradeoffs between performance and accuracy on select random matrices for tolerance values of $\{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\}$ and block sizes of $\{32, 64, 128\}$ without iterative refinement.

			$n_b = 32$			$n_b = 64$			$n_b = 128$		
		Corr.	# Mods.	Time (s)	Error	# Mods.	Time (s)	Error	# Mods.	Time (s)	Error
rand_dominant	GEPP	-	-	-	-	-	25.1±0.5	2×10^{-14}	-	-	-
	GENP	-	-	-	-	-	6.7±0.4	1×10^{-14}	-	-	-
	$\hat{\tau} = 10^{-4}$	Y	0	9.8±0.2	1×10^{-14}	0	9.8±0.1	1×10^{-14}	0	10.9±0.3	1×10^{-14}
		N		9.6±0.1	1×10^{-14}		9.6±0.3	1×10^{-14}		10.8±0.4	1×10^{-14}
	$\hat{\tau} = 10^{-6}$	Y	0	9.5±0.4	1×10^{-14}	0	9.7±0.1	1×10^{-14}	0	10.9±0.2	1×10^{-14}
		N		9.5±0.2	1×10^{-14}		9.7±0.4	1×10^{-14}		10.8±0.1	1×10^{-14}
	$\hat{\tau} = 10^{-8}$	Y	0	9.5±0.2	1×10^{-14}	0	9.8±0.1	1×10^{-14}	0	10.9±0.2	1×10^{-14}
		N		9.5±0.3	1×10^{-14}		9.7±0.1	1×10^{-14}		10.9±0.2	1×10^{-14}
	$\hat{\tau} = 10^{-10}$	Y	0	9.6±0.3	1×10^{-14}	0	9.7±0.4	1×10^{-14}	0	11.0±0.3	1×10^{-14}
		N		9.4±0.2	1×10^{-14}		9.6±0.2	1×10^{-14}		10.8±0.1	1×10^{-14}
	$\hat{\tau} = 10^{-12}$	Y	0	9.4±0.4	1×10^{-14}	0	9.7±0.2	1×10^{-14}	0	10.8±0.2	1×10^{-14}
		N		9.5±0.2	1×10^{-14}		9.8±0.2	1×10^{-14}		10.8±0.2	1×10^{-14}
rand	GEPP	-	-	-	-	-	50.0±0.6	2×10^{-14}	-	-	-
	GENP	-	-	-	-	-	6.7±0.4	3×10^{-9}	-	-	-
	$\hat{\tau} = 10^{-4}$	Y	8 798	24.5±1.2	4×10^{-14}	8 397	23.7±0.9	2×10^{-14}	8 206	24.7±0.8	9×10^{-15}
		N		9.5±0.2	6×10^{-5}		9.7±0.3	5×10^{-5}		10.9±0.2	4×10^{-5}
	$\hat{\tau} = 10^{-6}$	Y	138	13.0±0.3	5×10^{-13}	126	12.9±0.3	2×10^{-13}	122	14.0±0.1	1×10^{-13}
		N		9.5±0.2	2×10^{-7}		9.7±0.3	2×10^{-7}		10.7±0.3	2×10^{-7}
	$\hat{\tau} = 10^{-8}$	Y	2	10.1±0.2	3×10^{-12}	2	10.2±0.1	2×10^{-12}	1	11.2±0.2	1×10^{-12}
		N		9.5±0.3	1×10^{-9}		9.7±0.5	4×10^{-11}		10.8±0.2	2×10^{-10}
	$\hat{\tau} = 10^{-10}$	Y	0	9.6±0.5	7×10^{-11}	0	9.8±0.2	5×10^{-12}	0	10.7±0.4	1×10^{-12}
		N		9.5±0.2	7×10^{-11}		9.8±0.2	5×10^{-12}		10.8±0.2	1×10^{-12}
	$\hat{\tau} = 10^{-12}$	Y	0	9.4±0.3	7×10^{-11}	0	9.7±0.2	5×10^{-12}	0	10.7±0.1	1×10^{-12}
		N		9.5±0.3	7×10^{-11}		9.7±0.3	5×10^{-12}		10.7±0.2	1×10^{-12}

Table 3.6: Tradeoffs between performance and accuracy on select structured matrices for tolerance values of $\{10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}\}$ and block sizes of $\{32, 64, 128\}$ without iterative refinement.

			$n_b = 32$			$n_b = 64$			$n_b = 128$		
		Corr.	# Mods.	Time (s)	Error	# Mods.	Time (s)	Error	# Mods.	Time (s)	Error
orthog	GEPP	-	-	-	-	-	50.2±1.2	3×10^{-15}	-	-	-
	GENP	-	-	-	-	-	6.6±0.3	5×10^{-5}	-	-	-
	$\hat{\tau} = 10^{-4}$	Y	48 493	43.6±2.3	6×10^{-8}	49 065	44.0±2.5	5×10^{-11}	49 314	45.3±1.7	2×10^{-11}
		N		9.6±0.4	1×10^{-4}		9.8±0.1	1×10^{-4}		10.5±0.3	1×10^{-4}
	$\hat{\tau} = 10^{-6}$	Y	44 980	41.8±2.0	1×10^{-6}	47 216	43.1±1.7	4×10^{-7}	48 470	44.8±2.0	8×10^{-8}
		N		9.6±0.2	1×10^{-6}		9.7±0.2	1×10^{-6}		10.7±0.3	1×10^{-6}
	$\hat{\tau} = 10^{-8}$	Y	1 239	18.8±0.3	4×10^{-4}	21 420	29.6±1.1	2×10^{-5}	46 159	43.5±2.2	2×10^{-7}
		N		9.5±0.2	4×10^{-4}		9.7±0.3	2×10^{-5}		10.7±0.3	2×10^{-7}
	$\hat{\tau} = 10^{-10}$	Y	987	18.1±0.4	4×10^{-4}	1 022	18.2±0.2	6×10^{-4}	1 121	19.9±0.4	4×10^{-4}
		N		9.5±0.1	5×10^{-4}		9.7±0.2	3×10^{-4}		10.6±0.0	5×10^{-4}
	$\hat{\tau} = 10^{-12}$	Y	846	18.0±0.7	5×10^{-4}	873	18.2±0.3	2×10^{-4}	892	19.2±0.5	5×10^{-4}
		N		9.5±0.2	1×10^{-4}		9.6±0.6	3×10^{-4}		10.5±0.2	5×10^{-4}
zielkeNS	GEPP	-	-	-	-	-	39.5±0.3	2×10^{-19}	-	-	-
	GENP	-	-	-	-	-	6.7±0.3	NaN	-	-	-
	$\hat{\tau} = 10^{-4}$	Y	96 875	89.4±3.7	NaN	95 313	89.1±4.2	NaN	95 313	90.3±1.8	NaN
		N		9.3±0.5	7×10^{-5}		9.7±0.2	7×10^{-5}		10.8±0.2	7×10^{-5}
	$\hat{\tau} = 10^{-6}$	Y	3 156	21.2±0.9	NaN	1 594	19.4±0.7	NaN	813	19.1±0.3	NaN
		N		9.3±0.2	NaN		9.6±0.1	NaN		10.7±0.3	NaN
	$\hat{\tau} = 10^{-8}$	Y	3 156	21.0±0.9	NaN	1 594	19.5±0.7	NaN	813	19.1±0.3	NaN
		N		9.3±0.4	NaN		9.5±0.3	NaN		10.7±0.3	NaN
	$\hat{\tau} = 10^{-10}$	Y	3 156	20.9±0.7	NaN	1 594	19.4±0.4	NaN	813	19.2±0.0	NaN
		N		9.4±0.5	NaN		9.6±0.1	NaN		10.6±0.2	NaN
	$\hat{\tau} = 10^{-12}$	Y	3 156	20.9±0.5	NaN	1 594	19.5±0.7	NaN	813	19.2±0.3	NaN
		N		9.3±0.6	NaN		9.5±0.2	NaN		10.6±0.3	NaN

to be for tolerances of 10^{-6} or 10^{-8} (depending on the block size) without the Woodbury correction. Unexpectedly, a smaller block size led to fewer modifications; I suspect this is due to element growth in the later diagonals. For `zielkeNS`, only $\hat{\tau} = 10^{-4}$ without the Woodbury formula produced a non-NaN solution and the block size had little effect on the performance or the accuracy. For $\hat{\tau} = 10^{-4}$, all three block factor sizes resulted in the modification of about 95% of the diagonal singular values, whereas for smaller tolerance values, the number of modifications was just slightly larger than the number of blocks.

Scaling Results

Finally, Fig. 3.7 compares the performance of the different solvers as the size varies for the `rand+nI`, `rand`, and `orthog` matrices. BEAM achieved speedups from 4× to almost 5× for the three matrices compared to GEPP applied to `rand`. BEAM was configured with an algorithmic blocking factor size of $n_b = 64$ and a tolerance of $\hat{\tau} = 10^{-8}$. BEAM with iterative refinement ran out of GPU memory for $n = 250\,000$ due to the extra copy of the system matrix. Note that a diagonally dominant matrix, such as the `rand+nI`, is the best-case scenario for the performance of GEPP because the selected pivots already reside on the diagonal and the memory traffic of exchanging rows is avoided (though I still perform the pivot search for each column). For the large matrices, BEAM reached 80% of GENP's performance for `rand+nI` and `rand`, as the former required no modifications and the latter required just a few modifications. Without the Woodbury correction, BEAM also performed similarly on `orthog`. However, with the correction, the performance dropped to approximately that of the best-case for GEPP. Adding iterative refinement slightly reduced the overall performance, but BEAM still outperformed the best-case scenario of GEPP by 84% to 162% on the `rand+nI` and `rand` matrices. Without the Woodbury formula, BEAM performed almost as well on `orthog` as on `rand` with speedups of 70% to 144%. With the Woodbury formula, BEAM performed in the range of GEPP, between 40% and 112% faster than the GEPP's performance on `rand` (which is close to its performance on `orthog`, as per Table 3.4). These speedups for `orthog` are particularly promising because most approaches struggle to accurately outperform GEPP on this matrix, especially for large sizes [41, 91, 93, 105].

3.4.5 Parameter Selection

The success of BEAM depends heavily on both the tolerance and whether to apply the Woodbury formula. The block size also matters but, based on Tables 3.5 and 3.6, to a lesser extent; I suggest starting with the size of cache-blocking for non-pivoted LU or slightly larger. Below, I analyze in detail considerations for choosing the threshold and whether to

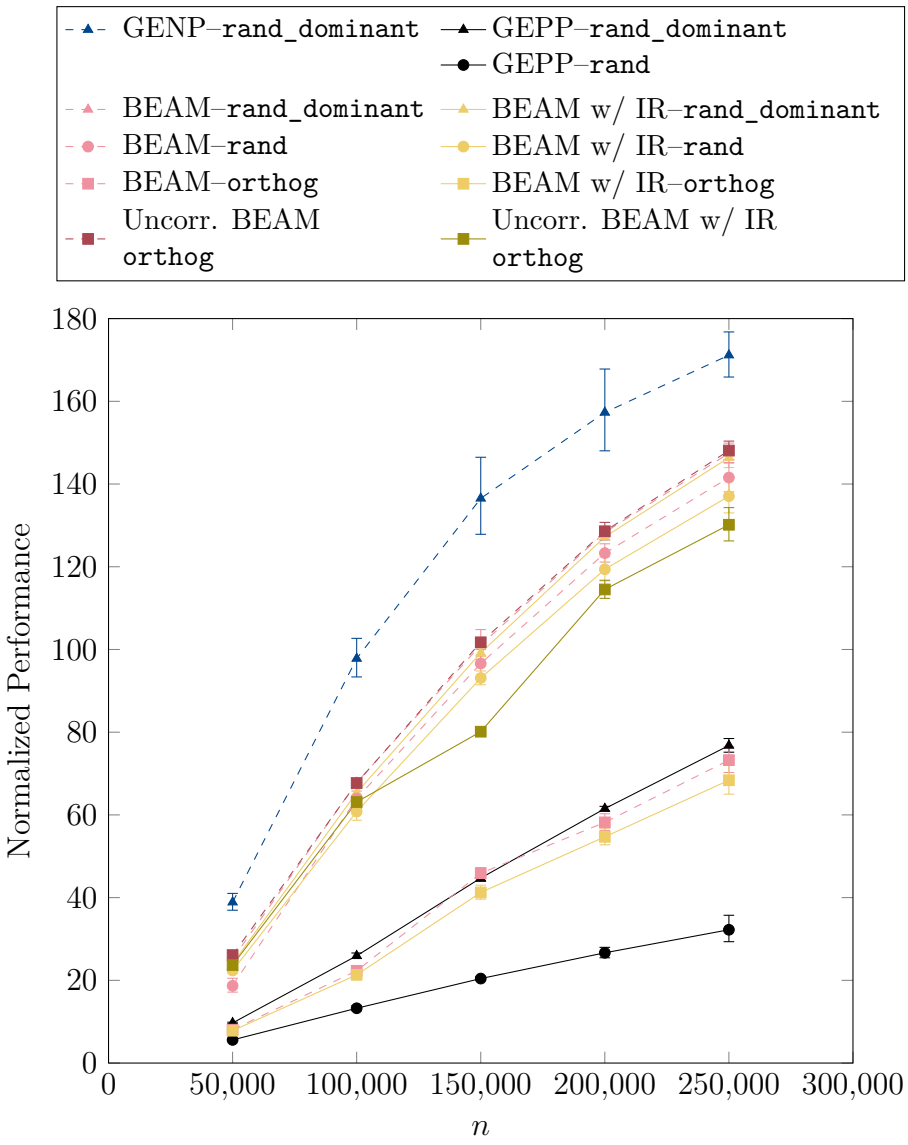


Figure 3.7: Performance of BEAM for three matrices compared with GEPP and GENP. The y-axis is equal to $\frac{2}{3}n^3 10^{-12}$ divided by the time in seconds; for GEPP and GENP, this is equivalent to TFLOP/s.

use the Woodbury formula. But as a starting point, I suggest $\hat{\tau} = \min(\frac{1}{2}\kappa_2(A), 10^{-8})$ and no Woodbury formula. For most applications, however, various configurations should be tested on the linear systems produced by representative domain problems.

For the Woodbury formula, recall that in Table 3.4 and Fig. 3.7, the corrected solver never outperformed the corresponding uncorrected one. However, a few ill-conditioned cases failed to converge without the Woodbury formula, while all cases converged when using the Woodbury formula (except for `zielkeNS`, which overflowed for both). Furthermore, as noted before, Table 3.4 indicates that using the Woodbury formula can enable convergence when $\hat{\tau}\kappa_2(A) \geq 1$. Thus, the Woodbury formula appears to be preferable for ill-conditioned matrices. And iterative refinement already measures the quality of the factorization. So, I suggest initially skipping the Woodbury formula. Then if iterative refinement fails to converge within, e.g., five iterations, use the Woodbury formula in subsequent iterations.

For selecting $\hat{\tau}$, I first wish to draw attention to the importance of the inequality $\hat{\tau}\kappa_2(A) \ll 1$. For BEAM without a Woodbury correction, (3.10) implies that this inequality is a prerequisite to proving that the solution has at least one digit of accuracy and that iterative refinement can converge to full backward accuracy. For BEAM with a Woodbury correction, this is necessary to show that $\widetilde{A}$ is well conditioned using Theorem 3.1. Finally, the experimental results demonstrated that violating this inequality can lead to a failure of BEAM without the Woodbury formula. Although, the experimental results also failed to show a similar result when the Woodbury formula was applied. Thus, there may be a subtle interaction between the perturbations and the resulting Woodbury correction that leads to better stability than the existing analysis suggests.

Beyond ensuring $\hat{\tau}\kappa_2(A) \ll 1$, there are a few relative concerns in the selection of $\hat{\tau}$. First, consider the omission of the Woodbury correction. Recall that in Table 3.4, the smallest tolerance (i.e., 10^{-10}) outperformed the largest tolerance (i.e., 10^{-6}) in all but one case. Furthermore, the added perturbations become overwhelmed by the roundoff perturbations when $10^{-10} \lesssim \hat{\tau} \lesssim 10^{-8}$ (depending on the matrix). Thus, a small tolerance, such as the square root of unit roundoff, is preferable. Next, consider the inclusion of the Woodbury correction. Here the number of modifications becomes relevant in addition to their magnitude. Unfortunately, I know of no way to determine a priori the number of modifications that will result from a given tolerance. However, Table 3.4 suggests that, like in the uncorrected case, smaller tolerances usually result in better performance. Thus, I recommend starting with a similar tolerance to the uncorrected case.

3.4.6 Alternative block factorizations

The high cost of the SVD limits the block size that can be used without adding significant overhead. Thus, it is of interest to consider cheaper factorizations. Besides the SVD, most rank-revealing factorizations are based on the QR factorization, and I have focused on that family of factorizations. The most common is QR with column pivoting (QRCP), which factors the matrix one column at a time, pivoting the column with the largest magnitude to the front at each step.

Unfortunately, there are certain types of matrices for which QRCP completely fails to detect the small singular values [82]. The QLP factorization is designed to address these deficient cases [118]. QLP first applies a QRCP to the matrix, then applies an LQ factorization to the resulting R factor. (The second factorization can include row pivoting to improve stability, although just for specific counterexamples.) Because it takes the form of the first few steps of the QR algorithm for computing the SVD [79], QLP provides significantly more accurate approximations of the singular values than QRCP.

On the other side of the coin, QRCP (and thus QLP) requires computing the norm of each column at each step of the factorization. This significantly decreases the arithmetic intensity of the factorization. Recently, the pivoting avoiding QR factorization (PAQR) has been proposed to avoid this overhead [115]. PAQR only computes the norm of a column immediately before that column is factored; then, if the column is too small it is pivoted to the end of the matrix.[4]

To understand the ability of these alternatives for the block factorization, I implemented BEAM using Julia [25]. Because QLP and PAQR address different aspects of the factorization, I tested a few combinations of those methods. Table 3.7 lists the tested block factorizations. Most of the tested matrices are discussed elsewhere. The main exception is `kahan+randn`, which deliberately exposes the weakness of QRCP by replacing the 64×64 leading principal submatrix of `randn` with Kahan's matrix [82]. Table 3.8 shows the accuracy for problems of size $n = 2000$ with a block size of $n_b = 64$, tolerance of $\tau = 10^{-6}\|A\|_F$, and the Woodbury formula. The tolerance for PAQR was set to that of BEAM.

The first five test matrices were all accurately factored with BEAM, regardless of the inner factorization. But, as Table 3.3 indicates, these problems can all be solved with moderate accuracy without any perturbation. Thus, there are no problematically small singular values to detect. The `kahan+randn` matrix is the first interesting result; because of the leading Kahan matrix, the one-sided QR-style factorizations could not detect the small singular value.

[4]The original PAQR algorithm completely discards the deficient columns. However, this would introduce an additional, unnecessary perturbation in the context of BEAM.

Table 3.7: Factorizations tested within BEAM.

Factorization	Description
SVD	The canonical measure of the singular values.
QRCP	The traditional rank-revealing QR.
QR	The traditional QR factorization without rank-revealing properties.
PAQR	The new PAQR algorithm.
QLP	QRCP followed by the transposed algorithm on the R factor.
QLQ	QLP without the column or row pivoting.
PAQLP	QLP with PAQR and its transpose replacing QRCP.

Table 3.8: Spectral norm backward error of BEAM for different block factorizations.

Matrix	GEPP	GENP	SVD	QLP	PAQLP	QLQ	QRCP	PAQR	QR
`randn`	1×10^{-14}	5×10^{-12}	9×10^{-14}	9×10^{-14}	9×10^{-14}	1×10^{-13}	8×10^{-14}	9×10^{-14}	9×10^{-14}
`randb`	6×10^{-16}	NaN	4×10^{-15}	5×10^{-15}	5×10^{-15}	7×10^{-15}	5×10^{-15}	5×10^{-15}	5×10^{-15}
`randn+nI`	7×10^{-16}	8×10^{-16}	1×10^{-15}	6×10^{-16}	5×10^{-16}	5×10^{-16}	5×10^{-16}	5×10^{-16}	4×10^{-16}
`chebspec`	3×10^{-18}	2×10^{-8}	6×10^{-18}	5×10^{-18}	1×10^{-17}	6×10^{-18}	4×10^{-18}	3×10^{-18}	5×10^{-18}
`fiedler`	1×10^{-17}	NaN	5×10^{-16}	2×10^{-16}	5×10^{-16}	4×10^{-16}	2×10^{-16}	4×10^{-16}	4×10^{-16}
`kahan+randn`	1×10^{-14}	5×10^{-1}	7×10^{-13}	7×10^{-13}	7×10^{-13}	7×10^{-13}	7×10^{-6}	3×10^{-5}	2×10^{-5}
`orthog`	8×10^{-15}	1×10^{0}	5×10^{-9}	7×10^{-9}	2×10^{-7}	7×10^{-7}	2×10^{-8}	2×10^{-3}	1×10^{0}
`ris`	9×10^{-16}	1×10^{0}	2×10^{-10}	1×10^{-10}	2×10^{-10}	7×10^{-11}	3×10^{-10}	7×10^{-1}	1×10^{0}
`zielkeNS`	5×10^{-19}	NaN	8×10^{-4}	8×10^{-4}	8×10^{-4}	8×10^{-4}	NaN	NaN	NaN

On the other hand, the two-sided QLP-style factorizations correctly detected the deficiency due to the second orthogonal transform. The `zielkeNS` matrix was similar. The `orthog` matrix benefited more from column pivoting. For that problem, QRCP outperformed the other block factorizations except for SVD and QLP. Although, for the other factorizations, there was still a noticeable benefit of using the QLP-style factorizations instead of the QR-style ones. The `ris` matrix was similar to `orthog`, except QRCP was slightly worse than the QLP-style factorizations.

3.5 Comparing RBT and BEAM

Because RBT and BEAM were both able to significantly outperform GEPP, the natural question is how they compare with each other. First, they try to limit growth in different ways. RBT works at a global level to even out the nonsingularity. BEAM, on the other hand, focuses on just the diagonal blocks and perturbing them to be manageable. Thus, the two approaches likely have different weaknesses, although Tables 3.1 and 3.3 show that both methods struggle on `orthog` and `ris`. Combining the two approaches would likely provide a more robust solver than either of them alone. Second, the have significantly different performance profiles. Most of BEAM's overhead occurs in correcting any modifications, while RBT's main overhead is transforming the system matrix. Thus, BEAM's performance will vary depending on the numerical properties, while RBT's performance is determined more by the matrix's distribution. This means that the performance of the RBT solver will be easier to predict *a priori*.

3.5.1 Experimental performance

To better understand how the two approaches compare, I tested them both on Summit. Both used iterative refinement and terminated the iterations with the ∞-norm backward error was less than unit round off. While threshold pivoting was not explicitly compared, the best case performance of GEPP provides an upper bound on its performance. BEAM used the Woodbury correction and a block size of 64. To show the case where BEAM requires a large correction, that solver was also tested with `orthog`. RBT was not tested with `orthog` since Table 3.1 shows that iterative refinement does not improve the solution.

The test configuration matched that of the RBT performance experiments from Section 3.3.2. The software stack included GCC 9.1.0, CUDA 11.0.3, Spectrum MPI 10.4.0.3, ESSL 6.3.0, and Netlib LAPACK 3.9.1. Summit was run in `smt1` mode to disabling simultaneous multithreading. Results were measured using a modified version of SLATE's

tester. The solvers' parameters were tuned for performance. Tests were run with the flags `--ref n --origin h --target d --seed 42 --seedB 64 --ib 64 --lookahead 1 --grid 4x4`. GENP also set `--nb 512`; the RBT solver was run with same the flags as GENP as well as the `--fallback n --depth 2` flags. GEPP also used the flags `--nb 896 --panel-threads 16`. BEAM also used the flag `--addtol -1e-8` to match Section 3.4.4. The flags `--dim`, `--matrix`, and `--matrixB` were also used to control the test matrices. A problem of size 10 000 was solved before running the actual performance tests to avoid measuring the first-time initialization costs of BLAS and MPI.

Figure 3.8 shows the result of this experiment. Overall, BEAM with only a few modifications is comparable to RBT aligned to the tiles. The unaligned RBT performs worse, but BEAM with a high-rank Woodbury correction performs the slowest—close to the best case of GEPP. However, it's worth recalling that the worst case of BEAM is on a matrix that the RBT solver cannot solve.

Because the good cases of BEAM and RBT are similar, I look at them in more detail. For small problem sizes, the performance of the two are very close which suggests the performance is dominated by iterative refinement and other startup costs. RBT is the most competitive for medium sizes, while BEAM matches RBT's no-communication case for large sizes. This indicates that BEAM's overheads scale better with problem size than RBT's overheads. BEAM's overheads (when the number of modifications is small) come from the extra work to process the diagonal tiles and should be $\mathcal{O}(n)$. RBT's overheads come from managing the large tile-wise communication and should be $\mathcal{O}(n^2)$.

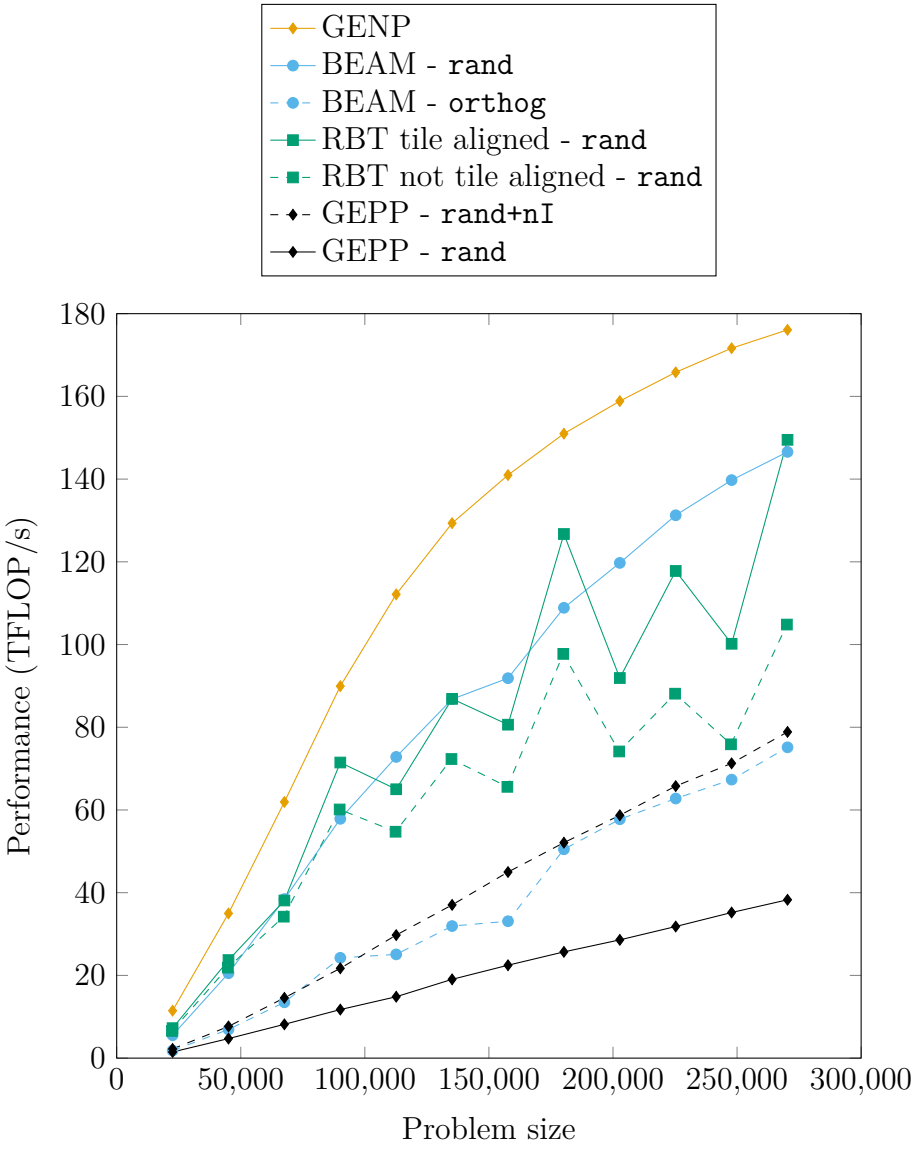


Figure 3.8: Performance of RBT compared to GENP and GEPP.

Chapter 4

Mixed Precision GMRES[1]

The generalized minimal residual method (GMRES) is a go-to Krylov method for solving sparse, non-symmetric systems of linear equations. Like most sparse, iterative solvers, GMRES is memory bound. Thus, developing techniques to reduce data movement will directly improve its performance. Solving the problem in single precision instead of double precision reduces the data movement by up to a factor of two, depending on the kernel. However, this limits the achievable accuracy, particularly for ill-conditioned problems. Thus, I have investigated selectively reducing parts of the method to single precision to improve performance while retaining double-precision accuracy. Algorithm 4.1 shows the formulation of GMRES used in this work.

While GMRES can be implemented with various orthogonalization schemes, I have focused on two: modified Gram-Schmidt (MGS) and classical Gram-Schmidt with reorthogonalization (CGS2). Often, MGS is recommended due to its stability and low operation count. However, CGS2 can be more efficient on distributed and GPU-accelerated systems, despite doubling the operation count, by consolidating the dot-productions into two synchronizations [51]. (See the results in Section 4.3.2.) Additionally, CGS2 provides better numerical stability [55, 102], which may compensate for the reduced precision.

4.1 Numerics of Mixed-Precision GMRES

To develop mixed-precision algorithms, we must understand how various parts of the solver affect the final accuracy. First, note that restarted GMRES is equivalent to iterative refinement where the error correction is computed by non-restarted GMRES. This equivalence can be

[1]This chapter reuses material from two of my published papers [92] (© 2020 Springer) and [94] (© 2021 IEEE). Coauthors include Piotr Luszczek and Jack Dongarra. Reused coauthor contributions are limited to high-level guidance and textual improvements.

$A \in \mathbb{R}^{n\times n};\ x_0, b \in \mathbb{R}^n;\ M^{-1} \approx A^{-1}$
for $k = 1, 2, \ldots$ **do**
  $z_k \leftarrow b - Ax_k$ ▷ FP64
  If $\|z_k\|_2$ is small enough, stop ▷ FP64
  $r_k \leftarrow M^{-1} z_k$ ▷ FP32
  $\beta \leftarrow \|r_k\|_2;\ s_0 \leftarrow \beta;\ v_1 \leftarrow r_k/\beta;\ V_1 \leftarrow [v_1]$ ▷ FP32
  $j \leftarrow 0$
  loop until the restart condition is met
    $j \leftarrow j + 1$
    $w \leftarrow M^{-1} A v_j$ ▷ FP32
    $w, h_{1,j}, \ldots, h_{j,j} \leftarrow \text{orth}(w, V_j)$ ▷ MGS or CGS2
    $h_{j+1,j} \leftarrow \|w\|_2;\ v_{j+1} \leftarrow w/h_{j+1,j};\ V_{j+1} \leftarrow [V_j, v_{j+1}]$ ▷ FP32
    for $i = 1, \ldots, j-1$ **do**
      $\begin{bmatrix} h_{i,j} \\ h_{i+1,j} \end{bmatrix} \leftarrow \begin{bmatrix} \alpha_i & \beta_i \\ -\beta_i & \alpha_i \end{bmatrix} \times \begin{bmatrix} h_{i,j} \\ h_{i+1,j} \end{bmatrix}$ ▷ FP32
    $\begin{bmatrix} \alpha_j & \beta_j \\ -\beta_j & \alpha_j \end{bmatrix} \leftarrow \text{rotation\_matrix}\left(\begin{bmatrix} h_{j,j} \\ h_{j+1,j} \end{bmatrix}\right)$ ▷ FP32
    $\begin{bmatrix} h_{j,j} & s_j \\ h_{j+1,j} & s_{j+1} \end{bmatrix} \leftarrow \begin{bmatrix} \alpha_j & \beta_j \\ -\beta_j & \alpha_j \end{bmatrix} \times \begin{bmatrix} h_{j,j} & s_j \\ h_{j+1,j} & 0 \end{bmatrix}$ ▷ FP32
  $H \leftarrow \{h_{i,\ell}\}_{1\le i,\ell\le j};\ s \leftarrow [s_1, \ldots s_j]^T$
  $u_k \leftarrow V_j H^{-1} s$ ▷ FP32
  $x_{k+1} \leftarrow x_k + u_k$ ▷ FP64

procedure MGS(w, V_j)
  $[v_1, \ldots, v_j] \leftarrow V_j$
  for i = 1, 2, . . . , j **do**
    $h_{i,j} \leftarrow {v_i}^T w$ ▷ FP32
    $w \leftarrow w - v_i h_{i,j}$ ▷ FP32
  return $w, h_{1,j}, \ldots, h_{j,j}$
procedure CGS2(w, V_j)
  $h \leftarrow V_j^T w$ ▷ FP32
  $w \leftarrow w - V_j h$ ▷ FP32
  $g \leftarrow V_j^T w$ ▷ FP32
  $w \leftarrow w - V_j g$ ▷ FP32
  $[h_{0,j}, \ldots, h_{j,j}]^T \leftarrow h + g$ ▷ FP32
  return $w, h_{1,j}, \ldots, h_{j,j}$

Algorithm 4.1: Restarted GMRES in mixed precision with left preconditioning [112] © 2021 IEEE

seen by noting that lines 6–18 of Algorithm 4.1 are equivalent to a non-restarted GMRES with a right-hand side of r and an initial guess of 0; the remaining lines are equivalent to iterative refinement. This structure suggests that A, b, x_k, and r_k should all be computed and stored in high precision [32]. Furthermore, it implies that the outer iteration can correct moderate floating-point errors in the solution update, u_k [33, 81]. This analysis suggests using reduced precision for lines 6–18 (including the orthogonalization procedure) and high precision for the rest of the algorithm.

Existing theoretical analyses further justify this combination of precisions. First, single-precision GMRES is backward stable to single precision under moderate assumptions [42, 101]. Furthermore, mixed-precision iterative refinement is backward stable to high precision as long as the inner-solver is backward stable to low precision and the matrix is reasonably conditioned [33]. Thus, Algorithm 4.1 should be backward-stable to full precision for most problems. However, this analysis ignores the possibility of restarting before achieving single-precision accuracy, a significant issue in practice due to the growing Krylov basis.

To understand the case in which GMRES is restarted before the inner-precision accuracy is reached, it is useful to investigate the effect of round-off error on the convergence. Recent work shows that MGS-GMRES initially converges at approximately the same rate regardless of whether the inner products of MGS are computed in finite or exact arithmetic [61]. However, that work assumes the rest of the process is computed exactly. Toward this end, I provide the following theorem, which says that finite-precision CGS2-GMRES converges at almost the same rate as its exact counterpart until a particular accuracy is reached. This implies that for restarted GMRES, if full-precision GMRES can converge, then reduced-precision GMRES should either converge similarly or reach the backward error threshold. In the latter case, iterative refinement will produce a backward stable solution [33]. In the former case, similar behavior is expected, but differences in vector directions could result in differences after restarting.

Theorem 4.1. *Let $x_j^{(f)}$ and $x_j^{(e)}$ be the solutions computed by j iterations of non-restarted GMRES in finite and exact precision, respectively, with* u *being the unit roundoff of finite precision. Let b be the right-hand side and p be the maximum number of nonzeros per row of A. Also let $c_1(n,j) \in \mathcal{O}(nj)$ and $c_4(n,j) \in \mathcal{O}(n^2j^3)$ be the same low-order polynomials as Giraud et al.'s error analysis of CGS2 [57]. Suppose* $\mathrm{u} < 10^{-3}$ *and* $c_4(n,j)\mathrm{u}\kappa_2(AV_j^{(f)}) < 1$

with $V_j^{(f)}$ being the computed Krylov basis. Let $\delta_+ = (1+\sqrt{\mathrm{u}})^{1/2}$ and $\delta_- = (1-\sqrt{\mathrm{u}})^{1/2}$. Then,

$$\begin{aligned}\|b - Ax_j^{(f)}\|_2 \le{}& \delta_+^2\delta_-^{-2}\|b - Ax_j^{(e)}\|_2 \\ &+ \left(\delta_+\gamma_p j^{1/2} + c_1(n,j)\mathrm{u}\right)\delta_+\delta_-^{-1}\|A\|_F\|x_j^{(W)}\|_2 \\ &+ \left(9\delta_+ j + \frac{c_1(n,j)}{\delta_- - \gamma_j j^{1/2}\delta_+}\right)\mathrm{u}\|A\|_2\|x_j^{(f)}\|_2 \\ &+ \frac{j^{1/2}\delta_+\|A\|_F\|x_j^{(f)}\|_2}{\delta_- - \gamma_j j^{1/2}\delta_+}\Big(\gamma_p + \gamma_j + (\gamma_j + 9uj\gamma_j + 9uj^{1/2})\delta_+\delta_-^{-1}(2+\gamma_p)\Big).\end{aligned}$$

where $x_j^{(W)}$ is the solution computed by j iterations of a particular weighted-GMRES, respectively.

Proof. Let $\cdot^{(f)}$ denote values computed by finite-precision GMRES and $\cdot^{(e)}$ denote values computed by exact GMRES. By line 3 in Algorithm 4.1, we can assume $x_0 = 0$ without loss of generality.

We start with Arnoldi's procedure. In finite precision, it produces $V_{j+1}^{(f)}$ and $H_j^{(f)}$ such that

$$AV_j^{(f)} + E_j = V_{j+1}^{(f)} H_j^{(f)}, \quad \text{and} \quad {V_{j+1}^{(f)}}^T V_{j+1}^{(f)} = I - F \tag{4.1}$$

where

$$\begin{aligned} E_j &= [\mathcal{E}_1 v_1, \mathcal{E}_2 v_2, \ldots, \mathcal{E}_j v_j] + \Delta H \\ |\mathcal{E}_i| &\le \gamma_p |A| && \text{for } i = 1, \ldots, j \text{ [71]}, \\ \|\Delta H\|_2 &\le c_1(n,j)\mathrm{u}\|A\|_2 && \text{[57, (25)]}, \\ c_1(n,j) &\in \mathcal{O}(nj^{3/2}). \end{aligned}$$

So, $\|E_j\|_2 \le \gamma_p \||A|\|_2 \||V_j^{(f)}|\|_2 + c_1(n,j)\mathrm{u}\|A\|_2$. By the assumption on the conditioning of $AV_j^{(f)}$, we have $\|F\|_2 \le \sqrt{\mathrm{u}}$ [57, Thm. 2, (32), and (38)]. Thus, there exists a symmetric positive definite matrix W such that ${V_{j+1}^{(f)}}^T W V_{j+1}^{(f)} = I$ and $\kappa(W) \le (1+\sqrt{\mathrm{u}})/(1-\sqrt{\mathrm{u}})$ [60, Lemma 2]. Hence,

$$\delta_- \le \sigma_1(V_{j+1}^{(f)}) \le \sigma_1(V_j^{(f)}) \le \sigma_j(V_j^{(f)}) \le \sigma_{j+1}(V_{j+1}^{(f)}) \le \delta_+$$

and

$$\|H_j^{(f)}\|_2 \le \left(\tfrac{\delta_+}{\delta_-}(1+\gamma_p) + c_1(n,j)\mathrm{u}\right)\|A\|_F.$$

The final computed solution is $x_j^{(f)} = V_{j+1}^{(f)} y_j^{(f)} + \delta x$ with $|\delta x| \leq \gamma_j |V_j^{(f)}||y_j^{(f)}|$ [71]. Then,

$$\|y_j^{(f)}\|_2 \leq \left(\delta_- - \gamma_j j^{1/2} \delta_+\right)^{-1} \|x_j^{(f)}\|_2$$

and

$$\|b - A x_j^{(f)}\|_2 \leq \|b - A V_j^{(f)} y_j^{(f)}\|_2 + \|A\delta x\|_2.$$

Combining this with the Arnoldi process error from (4.1) gives

$$\|b - A V_j^{(f)} y_j^{(f)}\|_2 = \|V_{j+1}^{(f)}(\beta e_1 - H_j^{(f)} y_j^{(f)})\|_2 + \|E_j y_j^{(f)}\|_2.$$

Thus, by the W-orthogonality of $V_{j+1}^{(f)}$,

$$\begin{aligned}\|V_{j+1}^{(f)}(\beta e_1 - H_j^{(f)} y_j^{(f)})\|_2 &\leq \|W^{-1/2}\|_2 \|W^{1/2} V_{j+1}^{(f)}(\beta e_1 - H_j^{(f)} y_j^{(f)})\|_2 \\ &\leq \delta_+ \|\beta e_1 - H_j^{(f)} y_j^{(f)}\|_2.\end{aligned}$$

Next, consider the least squares problem solved by lines 13–18 in Algorithm 4.1. Let $G_i^{(f)}$ and G_i be the computed and exact Givens rotation matrices to eliminate the subdiagonal elements of $H_j^{(f)}$, and let $R^{(f)}$ and R be the resulting computed and exact triangular matrices. Furthermore, let Q be the product of the $G_i^{(f)}$ matrices, and let $q^{(f)}$ and q be the computed and exact values of $Q^T \beta e_1$. Note that G_i, R and q may differ from the corresponding values computed by exact GMRES. Thus, $\|R^{(f)} - R\|_2 \leq 9uj\|H_j^{(f)}\|_2$ and $\|q^{(f)} - q\|_2 \leq 9uj\beta$ [71, Lemmas 19.8, 3.6, and 3.4]. So,

$$\|\beta e_1 - H_j^{(f)} y_j^{(f)}\|_2 \leq \|q^{(f)} - R^{(f)} y_j^{(f)}\|_2 + 9uj\beta + \|(R^{(f)} - R) y_j^{(f)}\|_2$$

Then, $y_j^{(f)}$ satisfies

$$(R^{(f)} + \Delta R) y_j^{(f)} = q^{(f)}[1{:}j]$$

where $|\Delta R| \leq \gamma_j |R^{(f)}|$. So,

$$\begin{aligned}\|q^{(f)} - R^{(f)} y_j^{(f)}\|_2 &= \left\| \begin{bmatrix} \Delta R y_j^{(f)} \\ q^{(f)}[j+1] \end{bmatrix} \right\|_2 \\ &\leq \|\Delta R y_j^{(f)}\|_2 + |q^{(f)}[j+1]| \\ &= \left(\min_y \|\beta e_1 - H_j^{(f)} y\|_2\right) + 9uj\beta + \gamma_j \||R^{(f)}|\|_2 \|y_j^{(f)}\|_2\end{aligned}$$

Additionally, $\||R^{(f)}|\|_2 \leq (j^{1/2} + 9j^{3/2}\mathrm{u})\|H_j^{(f)}\|_2$.

Next, we bound this minimization. Note that for any $y \in \mathbb{R}^j$

$$\begin{aligned}\|\beta e_1 - H_j^{(f)} y\|_2 &= \|W^{1/2} V_{j+1}^{(f)} (\beta e_1 - H_j^{(f)} y)\|_2 \\ &\leq \delta_-^{-1} \|b - V_{j+1}^{(f)} H_j^{(f)} y\|_2 \qquad ([60, (35)]) \\ &\leq \delta_-^{-1} \left(\|b - A V_j^{(f)} y\|_2 + \|E_j\|_2 \|y\|_2 \right).\end{aligned}$$

Additionally,

$$\begin{aligned}\left\| \arg\min_y (\|b - A V_j^{(f)} y\|_2 + \|E_j\|_2 \|y\|_2) \right\|_2 &\leq \left\| \arg\min_y (\|b - A V_j^{(f)} y\|_2) \right\|_2 \\ &\leq \delta_+ \|x_j^{(W)}\|_2\end{aligned}$$

where $x_j^{(W)}$ is the solution computed exactly by j iterations of W-weighted GMRES. So,

$$\min_y (\|b - A V_j^{(f)} y\|_2 + \|E_j\|_2 \|y\|_2) \leq \min_y (\|b - A V_j^{(f)} y\|_2) + \delta_+ \|E_j\|_2 \|x_j^{(W)}\|_2.$$

Because exact W-GMRES computes $\min_y(\|b - A V_j^{(f)} y\|_2)$,

$$\min_y \|b - A V_j^{(f)} y\|_2 \leq \tfrac{\delta_+}{\delta_-} \|b - A x_j^{(e)}\|_2.$$

Finally, combining the preceding inequalities gives

$$\begin{aligned}\|b - A x_j^{(f)}\|_2 \leq\ & \delta_+^2 \delta_-^{-2} \|b - A x_j^{(e)}\|_2 \\ &+ \left(\delta_+^2 \delta_-^{-1} \gamma_p j^{1/2} + \delta_+ \delta_-^{-1} c_1(n,j) \mathrm{u} \right) \|A\|_F \|x_j^{(W)}\|_2 \\ &+ \left(9\delta_+ j + \frac{c_1(n,j)}{\delta_- - \gamma_j j^{1/2} \delta_+} \right) \mathrm{u} \|A\|_2 \|x_j^{(f)}\|_2 \\ &+ \frac{j^{1/2} \delta_+ \|A\|_F \|x_j^{(f)}\|_2}{\delta_- - \gamma_j j^{1/2} \delta_+} \left(\gamma_p + \gamma_j + (\gamma_j + 9uj\gamma_j + 9uj^{1/2}) \delta_+ \delta_-^{-1} (2 + \gamma_p) \right). \square\end{aligned}$$

Note that $x_j^{(f)}$ and $x_J^{(e)}$ are equivalent to u_k from Algorithm 4.1 when GMRES is restarted after j iterations. When $\mathrm{u} = 2^{-24}$, $\|x_j^{(e)}\|_2 \approx \|x_j^{(f)}\|_2 \approx \|x_j^{(W)}\|_2$, $j^{3/2}\mathrm{u} \ll 1$, and $p^{3/2}\mathrm{u} \ll 1$, this can be simplified to

$$\frac{\|b - A x_j^{(f)}\|_2}{\|A\|_F \|x_j^{(f)}\|_2 + \|b\|_2} \lesssim 1.1 \frac{\|b - A x_j^{(e)}\|_2}{\|A\|_F \|x_j^{(e)}\|_2 + \|b\|_2} + (4j^{3/2} + 30j + 3pj^{1/2} + 3c_1(n,j))\mathrm{u}.$$

The low-order polynomials c_1 and c_4 can quickly become onerous for realistically sized matrices. Fortunately, this bound is worse than will occur in practice. First, they assume

round-off error will always accumulate without cancellation. However, recent probabilistic analysis of dot-products has shown that the relative error is almost always bounded by $u\sqrt{n}$, compared the worst-case bound of un [74]. Second, the error bounds assume that the dot product is summed sequentially. However, parallel reductions use tree structures, which incur lower round-off errors [71].

4.1.1 Numerical Experiments

To support the analytic conclusions, I ran a series of numerical experiments. The precisions of the various storage and arithmetic types within GMRES were templated to allow exploring the effect various terms have on the rate of convergence. The Kokkos performance portability library with the OpenMP backend was used for parallelism [45]. To maximize flexibility, custom mixed-precision kernels were used for the fundamental linear algebra building blocks. All of the tested matrices came from the SuiteSparse collection [38] and were stored in compressed sparse row (CSR) format. The right-hand sides were generated from random solution vectors with entries uniformly distributed between 0 and 1. An incomplete LU (ILU) was used as the preconditioner.

To verify the analysis of Section 4.1, I first tested the effect of each variable by storing it in single precision while the rest of the solver remained in double precision. Figure 4.1 shows the normwise backward error after each inner iteration as if the solver had terminated for the `airfoil_2d` matrix. This matrix has 14 214 rows, 259 688 nonzeros, and a condition of 1.8×10^6. In the figure, the "Refinement Variables" include the matrix when used for the residual, the right-hand side, the solution, and the vector used to compute the unpreconditioned residual; the "Correction Variables" include the matrix when used to compute the next Krylov vector, the unpreconditioned residual, the Krylov vector being orthogonalized, the orthogonal basis, the upper triangular matrix from the orthogonalization process, and the vectors to solve the least-squares problems with Givens rotations. The convergence when storing the preconditioner in single precision was visually indistinguishable from the double-precision baseline and omitted from the figure for the sake of clarity. Each solver was restarted after 300 iterations. All of the solvers behaved the same until reaching single precision, where all but the baseline stopped improving. After restarting, reduced-precision "correction variables" still allowed the accuracy to continue improving to double-precision accuracy (after a second restart). On the other hand, reduced-precision "refinement variables" limited the solver to single-precision accuracy.

The convergence test was repeated with two mixed-precision solvers that reduced the precision of multiple variables. The first used double precision only for computing the residual

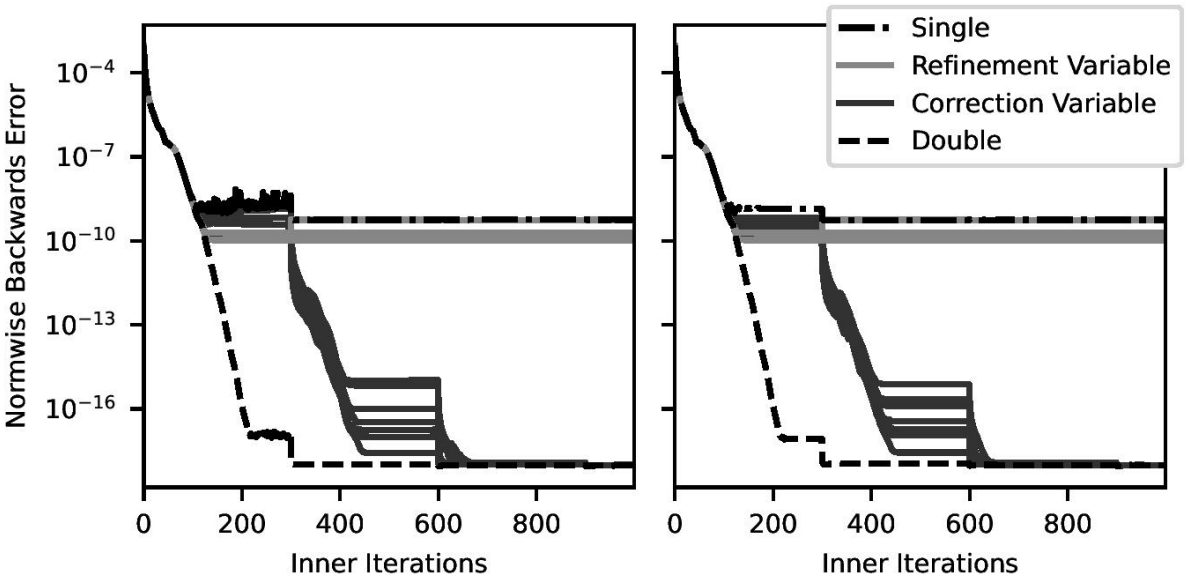


Figure 4.1: Rate of convergence when reducing the precision of individual variables for the `airfoil_2d` matrix when restarting every 300 iterations. Left uses MGS and right uses CGS2. © 2020 Springer

and error correction, i.e., it used single precision for lines 6–18 of Algorithm 4.1. The second was more limited, using single precision only to store A for computing the next Krylov vector, the preconditioner M^{-1}, and the Krylov basis V_j from Algorithm 4.1; these three variables contain most of the "correction variable" data. Figure 4.2 shows the normwise backward error after each inner iteration for using single, double, and mixed precisions with the `airfoil_2d` matrix. Both variants were able to achieve double-precision performance after two restarts. (Section 4.3 further shows the success of the first variant on almost every tested matrix.) Note that while limiting the use of mixed precision increases the improvement achieved before stalling, this improvement is small and does not reduce the importance of appropriately restarting. Additionally, reducing the precision of just the largest variables (i.e., the "limited" configuration) requires several mixed-precision kernels, while the aggressive mixed-precision implementation can be implemented using uniform-precision kernels and just two extra vector copies per restart.

One interesting observation was that the number of iterations before improvement stalled was approximately the same after each restart. However, the improvement in the relative residual was larger in the first outer iteration than the subsequent ones (see, for example, the MGS plot in Fig. 4.3). Table 4.1 displays the number of iterations before stalling after the first three restarts in the mixed-precision MGS-GMRES. Stalling was defined here to be the Arnoldi residual norm improving by less than a factor of 1.001 on the subsequent 5% of inner iterations per restart. This behavior appears to also hold for CGS2 but was not quantified because the Arnoldi residual continues to decrease even after the true residual reaches single-precision accuracy when using CGS2. I am unaware of any previous observations of this effect. I suspect that the errors accumulate similarly in each set of iterations, resulting in a similar number of iterations before convergence stalls. However, because the residual after restarting is (up to floating-point error) orthogonal to the previous Krylov subspace, it is likely that the first outer iteration is dominated by large eigenvectors while latter outer-iterations are dominated by smaller eigenvectors. However, because the convergence pattern of GMRES can be quite arbitrary [63], I doubt this trend holds for all linear systems.

4.2 Restart Strategies

Due to its connection with iterative refinement, the mixed-precision approach of Algorithm 4.1 depends on restarting often enough. For many problems, memory constraints or the increasing computations for orthogonalization will force a restart before the inner iterations have saturated low precision. However, for some systems, GMRES can produce a highly accurate solution in just a few iterations; in these cases, only restarting after a fixed number of

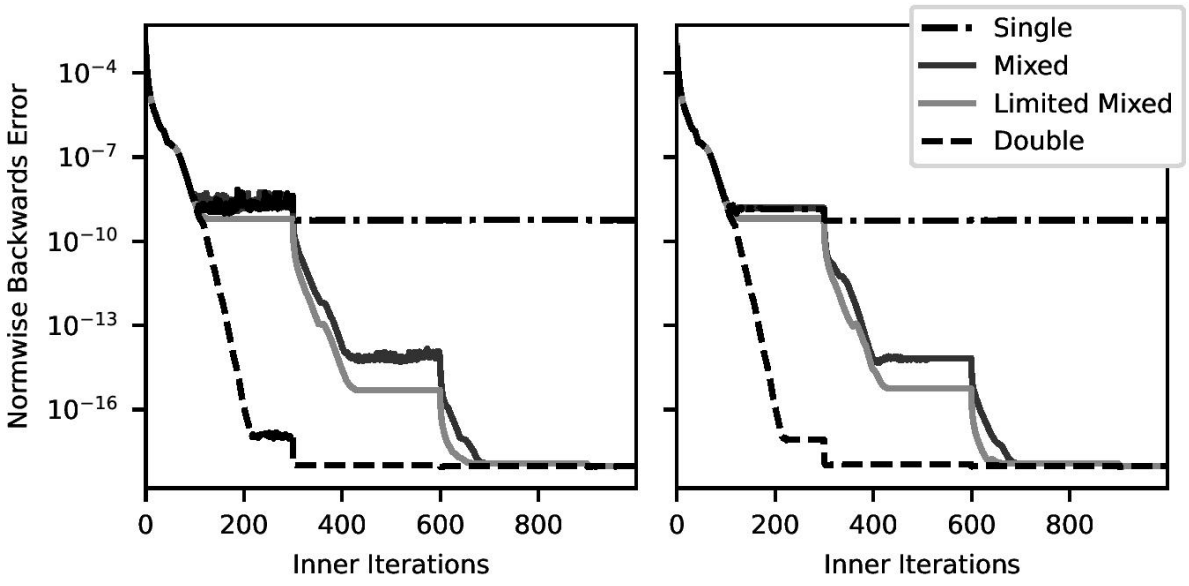


Figure 4.2: Rate of convergence when reducing the precision of several variables for the `airfoil_2d` matrix when restarting every 300 iterations. Left uses MGS and right uses CGS2. © 2020 Springer

Table 4.1: Number of iterations until the improvement stalls in mixed-precision MGS-GMRES © 2020 Springer

Matrix	Iterations per Restart	Iterations for 1st Stall	Iterations for 2nd Stall	Iterations for 3rd Stall
`airfoil_2d`	300	137	141	142
`big`	500	360	352	360
`cage11`	20	7	7	8
`Goodwin_040`	1250	929	951	924
`language`	75	23	21	21
`torso2`	50	28	27	25

inner iterations will cause the improvement to stall as the round-off error overwhelms any meaningful updates. On the other hand, the rate of convergence for GMRES relates to the approximation of the eigenvalues of A in the Arnoldi process [130]. Thus, restarting too often will increase the number of iterations to converge. Toward this end, I have investigated four restart strategies for mixed-precision GMRES. Loe et al. [97] have pointed out that this analysis focuses on maximizing the subspace dimension before restarting but that the increasing cost of orthogonalization causes GMRES to often benefit from restarting before memory limits are reached. Thus, these strategies should be used in addition to a similar iteration limit to double-precision GMRES.

Two important observations about the orthogonalization scheme affect the discussion of specific restart strategies. First, the Krylov basis vectors usually become linearly dependent when GMRES reaches the working-precision accuracy for, e.g., MGS [102], while they remain numerically orthogonal for, e.g., CGS2 [55]. Second, the norm of the Arnoldi residual (i.e., the residual for GMRES's least-squares problem) approximates the norm of the residual of the original preconditioned linear system of equations and is computed every iteration (s_{j+1} in Algorithm 4.1) [112, Proposition 6.9]. However, this approximation is only accurate until the working precision is reached [62]. The explanation is unknown, but the Arnoldi residual usually decreases past the true residual if and only if linearly independent vectors continue to be added to the Krylov basis [62]. Hence, the choice of orthogonalization scheme affects restart strategies based on the Arnoldi residual norm.

Our first restart strategy derives from the observation in Section 4.1.1 that convergence appears to stall after a similar number of inner iterations in each outer iteration. While this is not helpful for determining the first restart, it can be used for subsequent restarts either to trigger the restart directly or as a heuristic for when to start monitoring more expensive metrics. In my experiments, I use the following strategy to select the first restart when using this approach.

The second restart strategy is to monitor the Arnoldi residual norm until it improves by a particular amount. The simplest threshold is a fixed, scalar multiple, such as 10^{-6}. If this norm stops decreasing when floating-point accuracy is reached, such as with MGS, this criterion might not be met before improvement stalls. Thus, the threshold must be chosen carefully when using MGS. More advanced threshold selections may be effective but have not been explored.

Inspired by the problematic case of the second strategy, the third strategy is to detect when the Arnoldi residual norm stops improving. Obviously, this approach is only usable if the norm stops decreasing when GMRES has stalled (i.e., for MGS but not CGS2). However,

GMRES can have periods of no residual improvement during normal operation [63], which may cause premature restarts.

The final strategy is to detect when the orthogonalized basis becomes linearly dependent. It relates to the third strategy but uses an approach developed by Paige [100]. He and others have conjectured that MGS-GMRES converges to machine precision when the Krylov basis loses linear independence [101, 102]. For the basis matrix computed in the kth inner iteration, V_k, let $S_k = (I + U_k)^{-1}U_k$, where U_k is the strictly upper part of $V_k^H V_k$. Then, the basis is linearly dependent if and only if $\|S_k\|_2 = 1$ [100]. This matrix can be computed incrementally, appending one column per inner iteration, requiring $2nk + 2k^2$ FLOP per iteration. Estimating the 2-norm with i iterations of the power method requires an additional $i(2k^2 + 3k)$ FLOP by utilizing the strictly upper structure of the matrix.

4.2.1 Restart Experiments

To investigate the effectiveness of various restart strategies, I reused the experimental setup from Section 4.1.1.

First, consider the restart strategy based on a fixed improvement of the Arnoldi residual norm. Figure 4.3 shows the convergence of GMRES on the `airfoil_2d` matrix for various restart thresholds. For MGS, when the threshold is too ambitious, mixed-precision GMRES stalls before reaching it. Thus, this tolerance must be chosen conservatively, making it difficult to select a matrix-independent threshold. Compare Fig. 4.4, which shows the same test applied to the `big` matrix with 13 209 rows, 91 465 nonzeros, and an L2 norm of 4.4×10^7. The smallest successful threshold is two orders of magnitude less than that of `airfoil_2d`. Furthermore, the failed mixed-precision cases reiterate the observation from the end of Section 4.1.1 that the relative improvement decreases in the later outer iteration. Unlike MGS, CGS2 provided reliable convergence for all tolerances, with an aggressive tolerance usually providing faster overall convergence.

Next, consider the 2-staged approach, where the iteration count of the first restart is used to trigger the subsequent restarts. As mentioned before, the first restart was triggered by the relative improvement in the Arnoldi residual norm. Figure 4.5 show the convergence of this approach for the `airfoil_2d` matrix. Because only the choice of the first restart is important, more ambitious thresholds were tested than in Figs. 4.3 and 4.4. This two-staged approach performed a bit better than the simple threshold except when the first restart was not triggered. Figure 4.6 repeats this for the `big` matrix. Note how the same thresholds were used for the `big` test as the `airfoil_2d` test but were still able to converge and even

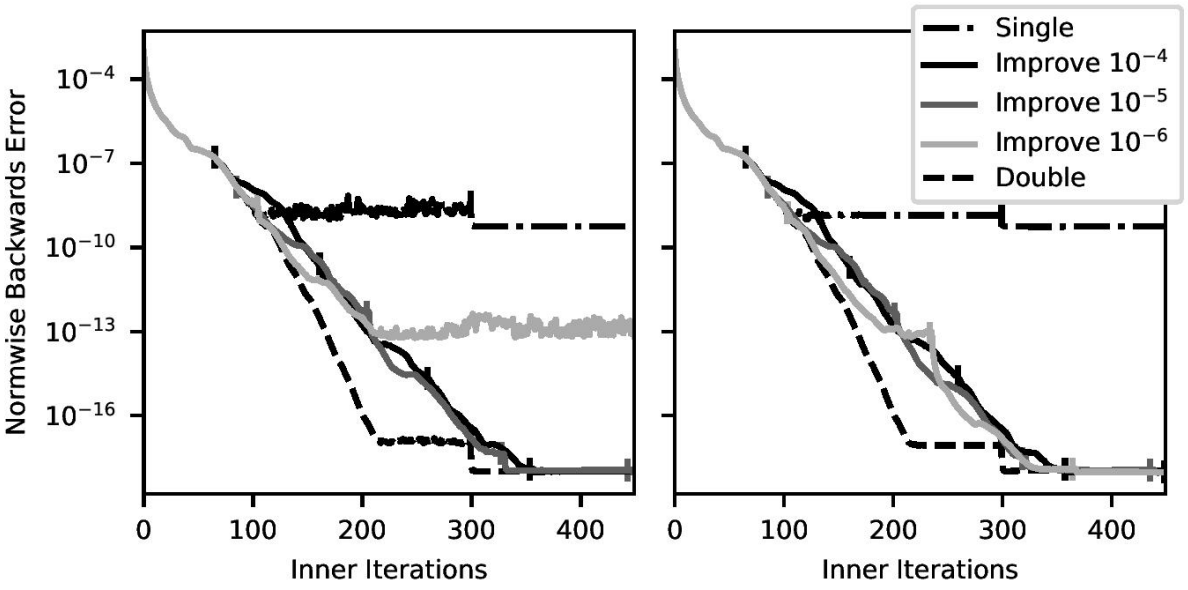


Figure 4.3: Rate of convergence for the `airfoil_2d` matrix when restarting mixed-precision GMRES after a fixed improvement in the Arnoldi residual norm. Left uses MGS and right uses CGS2. Vertical ticks indicate when restarts occurred. © 2020 Springer

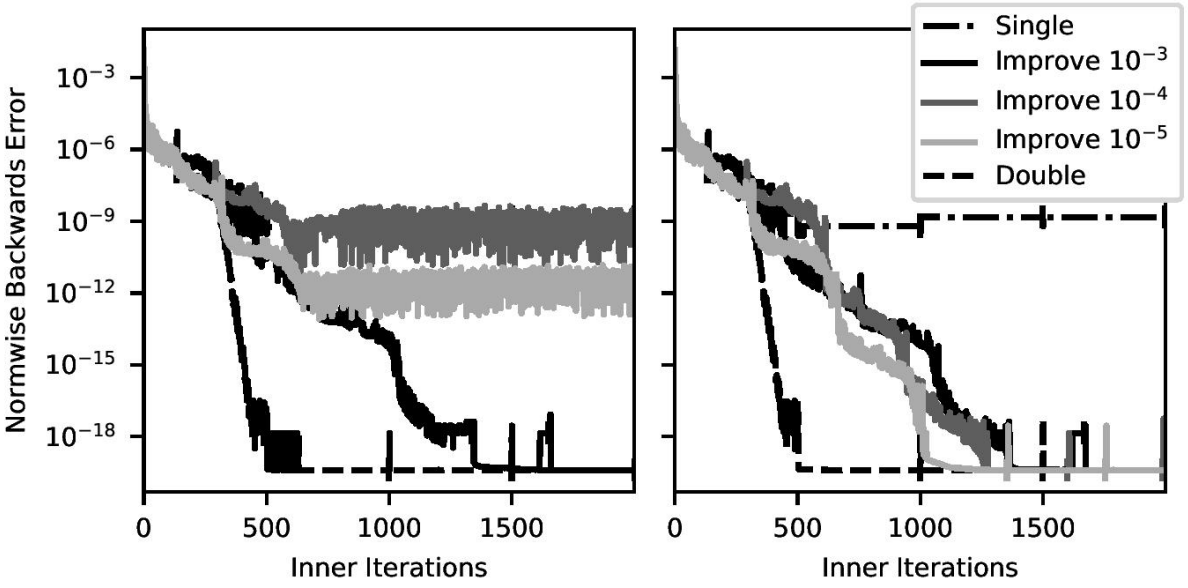


Figure 4.4: Rate of convergence for the `big` matrix when restarting mixed-precision GMRES after a fixed improvement in the Arnoldi residual norm. Left uses MGS and right uses CGS2. Vertical ticks to indicate when restarts occurred. © 2020 Springer

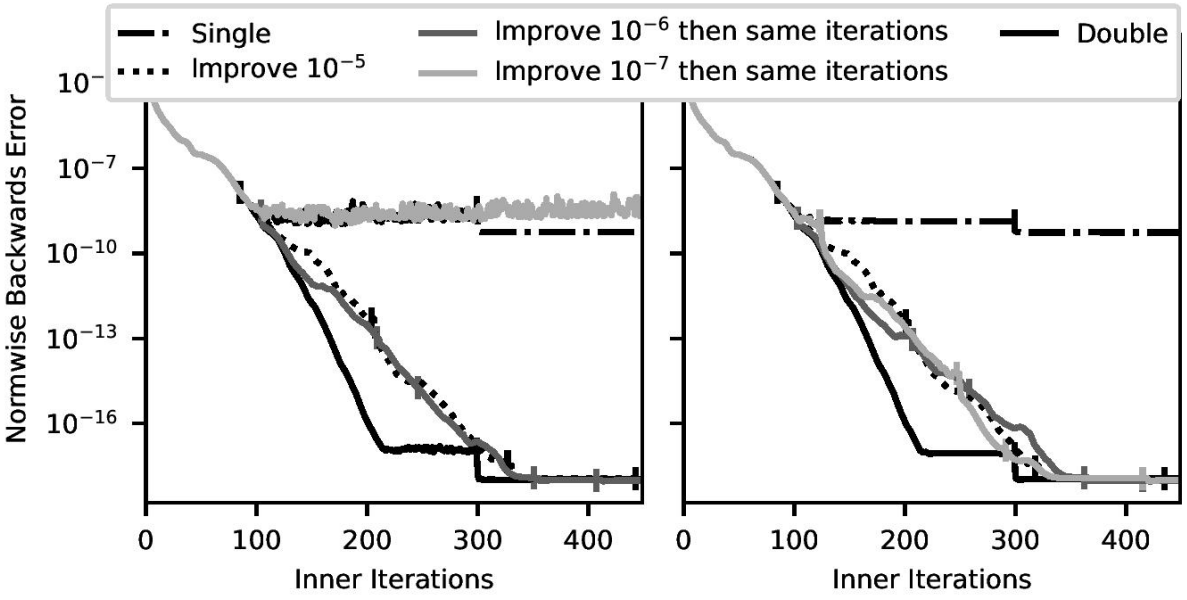


Figure 4.5: Rate of convergence for the `airfoil_2d` matrix when restarting mixed-precision GMRES after a fixed improvement in the Arnoldi residual norm for the first iteration and the same number of iterations thereafter. Left uses MGS and right uses CGS2. Vertical ticks to indicate when restarts occurred. The rate of convergence using just a fixed improvement threshold of 10^{-5} is added for comparison's sake. © 2020 Springer

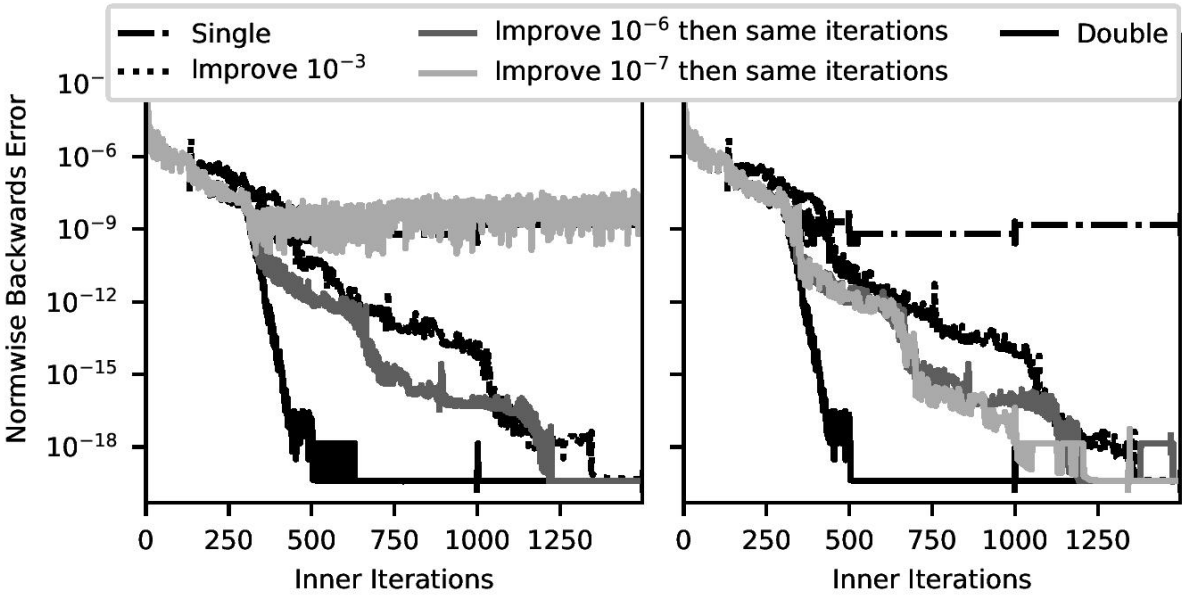


Figure 4.6: Rate of convergence for the `big` matrix when restarting mixed-precision GMRES after a fixed improvement in the Arnoldi residual norm for the first iteration and the same number of iterations thereafter. Left uses MGS and right uses CGS2. Vertical ticks to indicate when restarts occurred. The rate of convergence when restarting on a fixed improvement of 10^{-5} in the Arnoldi residual norm is added for comparison's sake. © 2020 Springer

outperform the previous tests with matrix-specific thresholds. This demonstrates reduced sensitivity to the threshold.

Finally, consider restarting based on the loss of orthogonality in the basis. Because CGS2 retains a high degree of orthogonality [55], this strategy was evaluated with only MGS-GMRES. Figure 4.7 shows the rate of convergence when restarting based on the norm of the S matrix. The spectral norm was computed using ten iterations of the power method. Additionally, the Frobenius norm was tested as a cheaper alternative to the spectral norm, although it does not provide the same theoretical guarantees. Interestingly, when using the spectral norm, a value of 0.5 was not detected until improvement had stalled for a noticeable period. Furthermore, even the larger Frobenius norm did not reach 1 until the improvement had visually stalled for a few dozen iterations. The cause of this deviation from Paige's theoretical results [100] is unknown but deserves further investigation.

4.3 Performance Experiments

Finally, I looked at the effect of reduced precision on performance. Additionally, by testing a larger variety of matrices, these tests further support the previous convergence results. The experiments are divided into CPU tests and GPU tests. Both sets of tests used various matrices from the SuiteSparse collection [38] in CSR format. Convergence was determined by the Frobenius-norm backward error:

$$\frac{\|b - Ax\|_2}{\|A\|_F\|x\|_2 + \|b\|_2} \leq 10^{-10}.$$

To measure variability, each test was run five times for CPU experiments or three times for GPU experiments. Speedups were computed with the median runtimes of each solver; error bars show the minimum and maximum speedup. The runtimes include any time spent constructing the preconditioner or copying the matrix to low precision.

These experiments were run on machines with two Haswell 10-core Intel® Xeon® CPU E5-2650 v3 @ 2.30GHz processors. The GPU-accelerated experiments used a node with a single NVIDIA V100 GPU. Each CPU core had a 32 KiB L1 instruction cache, a 32 KiB L1 data cache, and a 256 KiB L2 cache. Each socket had a shared 25 MiB L3 cache, and the entire node had 32 GiB of memory. The V100 card has 80 streaming multiprocessors, a 128 KiB L1 cache for each multiprocessor, a 6 MiB shared L2 cache, and a 16 GiB memory.

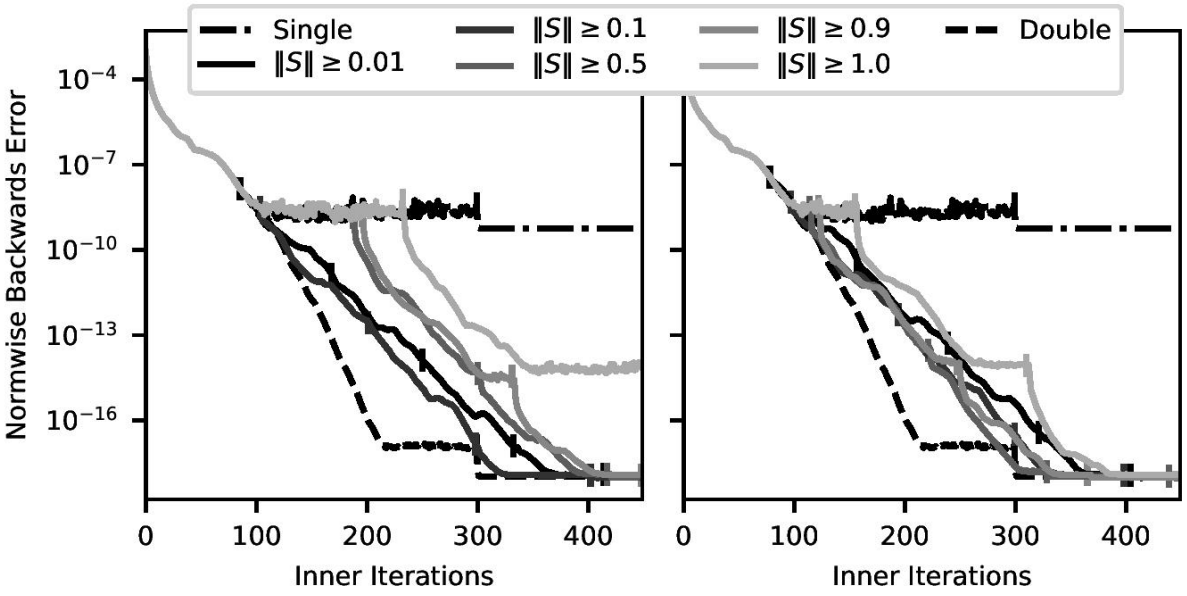


Figure 4.7: Rate of convergence for the `airfoil_2d` matrix when restarting based on Paige's S matrix. Left uses the spectral norm and right uses the Frobenius norm; both use MGS. Vertical ticks to indicate when restarts occurred. © 2020 Springer

4.3.1 CPU experiments

The first set of performance experiments considered CPU performance. Additionally, I tested the effectiveness of mixed precision both when the iteration limit forces a restart before single-precision accuracy is achieved and when GMRES converges without restarting. These experiments used an ILU preconditioner.

These experiments used Kokkos version 2.9.00 [45], KokkosKernels version 2.9.00, Intel C++ Compiler version 2018.1, Intel MKL version 2019.3.199, and Intel Parallel Studio Cluster Edition version 2019.3. Kokkos used the OpenMP backend with `OMP_NUM_THREADS=20 OMP_PROC_BIND=spread OMP_PROC_BIND=places`. The right-hand sides were generated from solution vectors with elements uniformly distributed on $[0, 1)$.

I first tested the performance improvement when the iteration limit forces GMRES to restart before reaching single-precision accuracy. For each of the tested systems, I computed the number of iterations for the double-precision solver to satisfy the convergence criterion without restarting. Then, the performance tests restarted after half that many iterations. For MGS, mixed precision took the same number of iterations as double precision on all but three systems; two took fewer iterations for mixed precision (`ecl32` and `mc2depi`) while the last took more (`dc1`). For CGS2, one additional system took more iterations for mixed precision (`big`). Figure 4.8 shows the speedup of the mixed-precision implementation and the single-precision ILU implementation relative to the double-precision implementation for each of the tested matrices. For the mixed-precision implementation, the geometric mean of the speedup was 19% and 24% for MGS and CGS2, respectively. For the single-precision ILU implementation, those means were both 2%.

The second set of performance tests shows what happens when GMRES is not forced to restart often enough for mixed precision. All of the matrices with an inner-iteration limit of less than 50 in the first experiment were tested again with an iteration limit of 50 iterations. For mixed-precision GMRES, the first restart could also be triggered by the Arnoldi residual improving by a factor of 10^{-6}, and subsequent restarts were triggered by the same number of inner iterations. To ensure the mixed-precision solver was not given any undue advantage, the other two solvers' performance was taken as the best time from three restart strategies: (1) the same strategy as mixed-precision GMRES; (2) after the Arnoldi residual improved by a factor of 10^{-8}; or (3) just the iteration limit. Figure 4.9 shows the results. For the mixed-precision implementation, the geometric mean of the speedup was -4% and 0% for MGS and CGS2, respectively. When just the preconditioner was reduced in precision, those means were 2% and 1% respectively. The matrices for which mixed-precision reduced performance were exactly those for which the double-precision implementation did not restart.

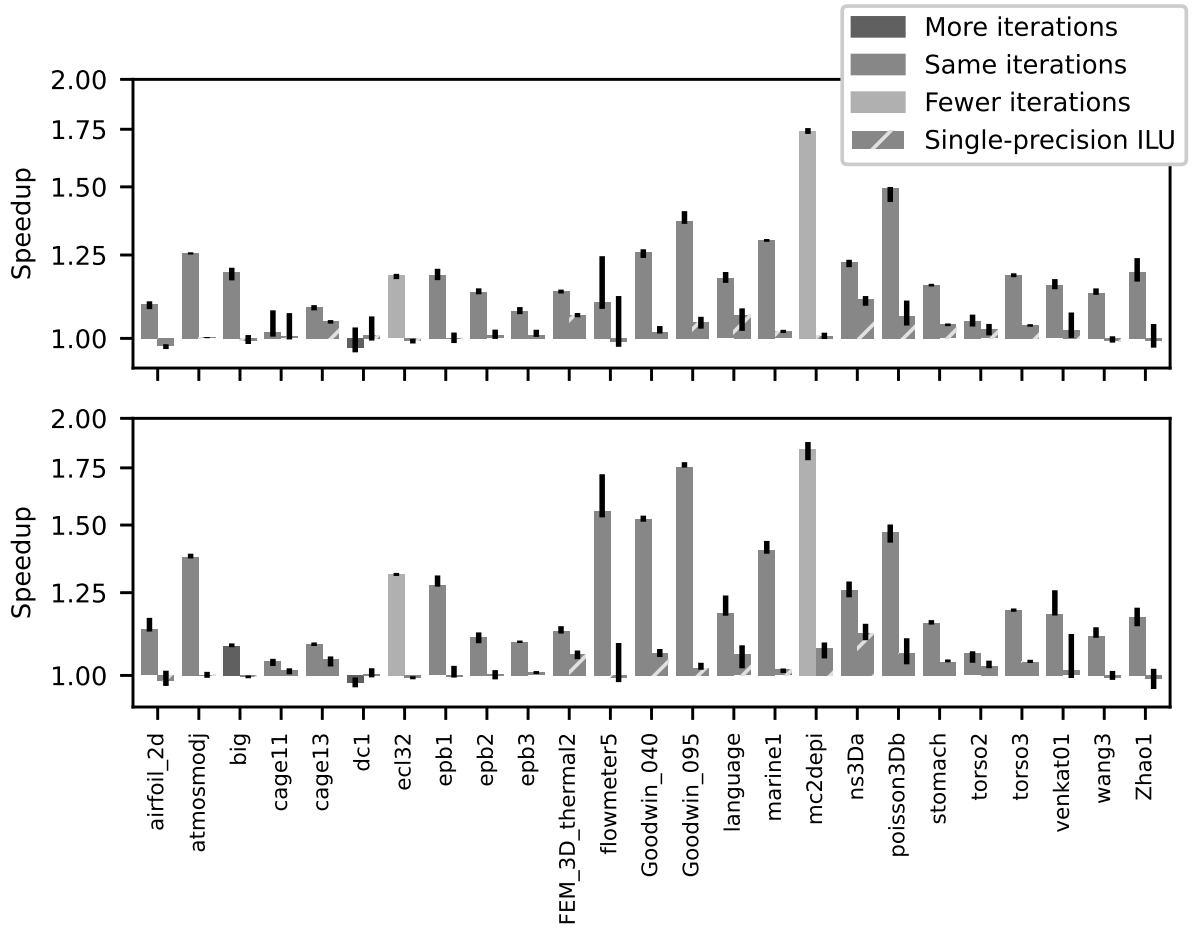


Figure 4.8: CPU performance when GMRES is restarted in half the number of iterations needed in double precision. Top plot uses MGS and bottom uses CGS2. © 2020 Springer

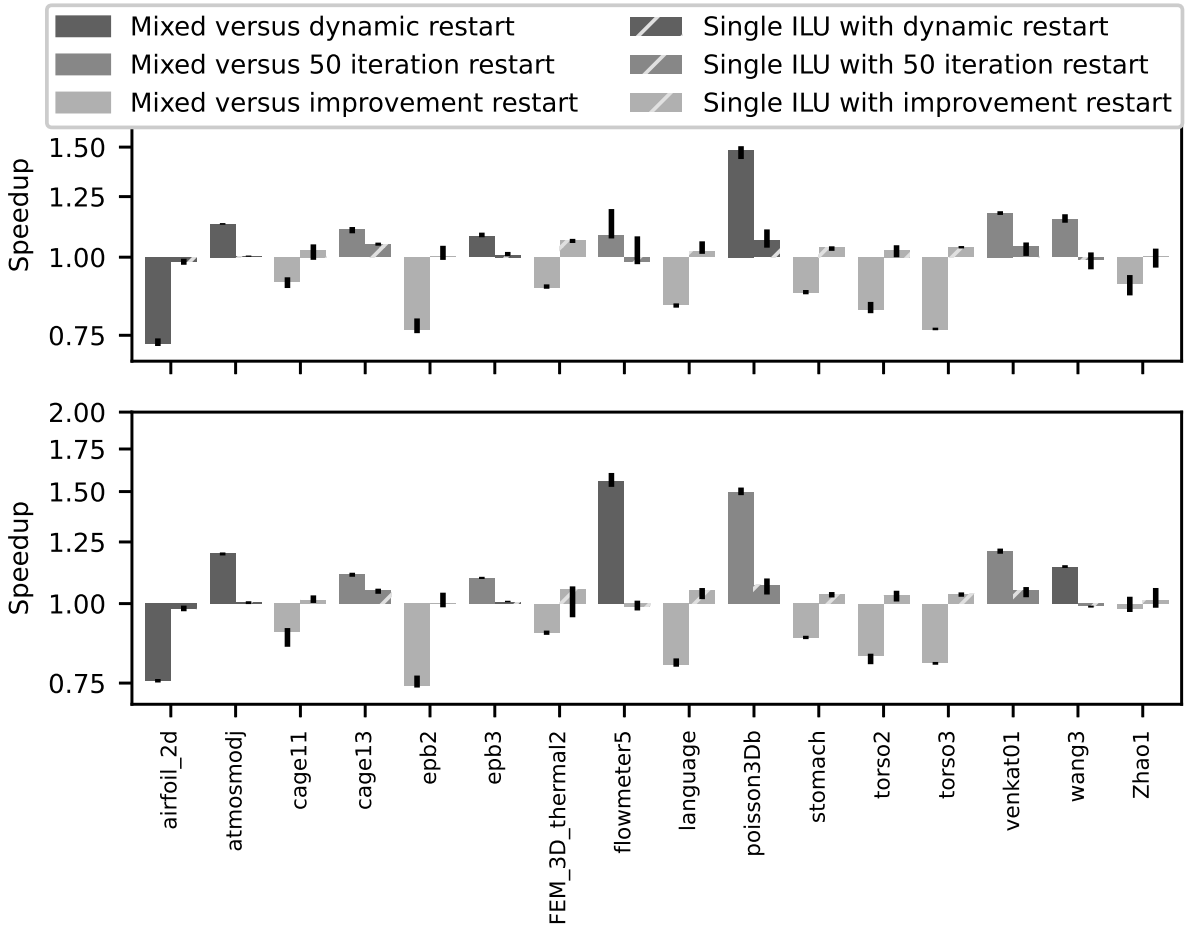


Figure 4.9: CPU performance when GMRES is restarted after 50 iterations or by the specified restart strategy. Top plot uses MGS and bottom uses CGS2. The bar colors indicate the optimal restart strategy used by the *baseline*. © 2020 Springer

4.3.2 GPU experiments

To test the performance of mixed-precision GMRES on GPUs, I implemented a restarted GMRES using the Kokkos [45], cuBLAS, and cuSPARSE libraries. To limit expensive memory transfers between CPU and GPU, all computation is done on the GPU and only the high-level control flow is done on the CPU. To understand the general effects of mixed precision, various matrices and preconditioners were tested.

Because of the high performance of GPUs, these experiments use matrices with at least one million nonzero elements. For a given preconditioner, I used only matrices where the double-precision implementation converged in fewer than 300 restarts. When SuiteSparse provided a file ending with `_b`, the first column was used as the right-hand side. Otherwise, the right-hand side was computed from a solution with elements randomly selected from the uniform range $[0, 1)$. Table 4.2 shows the tested matrices. In addition to structural properties, the table contains lower bounds of the condition numbers of these matrices, computed by testing forward error vectors of LSQR [13]. Many of the moderately ill-conditioned matrices reached the iteration limit before satisfying the convergence criterion, so they may have worse conditioning than these lower bounds imply.

I tested the mixed-precision approach with a variety of preconditioners that range from simple and cheap to highly effective but expensive. They are:

1. no preconditioner,
2. a scalar Jacobi preconditioner,
3. an ILU preconditioner, and
4. an ILU preconditioner using Jacobi iterations for the triangular solves.

The first three are standard techniques, but the fourth deserves a comment. Sparse triangular solves have little parallelism, resulting in poor GPU utilization. In contrast, Jacobi iterations process each row in parallel, allowing for better GPU performance [9]. Because the focus is on how well the mixed-precision approach works for different types of preconditioners, the preconditioners are not compared with each other. Furthermore, all preconditioners were constructed in double precision to ensure the same preconditioner is used, particularly for ILU. In addition to testing double-precision GMRES and the proposed mixed-precision GMRES, I also tested single-precision GMRES and double-precision GMRES with a single-precision preconditioner.

Average speedups were computed as the inverse of the geometric mean of the normalized mixed-precision times. Combined with a limit of 100 inner iterations before restarting, three restart strategies were tested:

1. just the inner-iteration limit;

Table 4.2: Properties of tested matrices. †Condition estimator reached 200 000 iterations without satisfying the convergence criterion. © 2021 IEEE

Matrix	Rows	Nonzeros	Condition Lower Bound	RHS Provided
af_0_k101	5.0×10^5	1.8×10^7	5.5×10^5 †	yes
af_shell9	5.0×10^5	1.8×10^7	1.2×10^6 †	yes
apache2	7.2×10^5	4.8×10^6	3.0×10^6 †	no
atmosmodj	1.3×10^6	8.8×10^6	6.4×10^3	yes
BenElechi1	2.5×10^5	1.3×10^7	1.3×10^6 †	yes
bone010	9.9×10^5	4.8×10^7	1.6×10^6 †	no
Bump_2911	2.9×10^6	1.3×10^8	7.5×10^6 †	no
cage13	4.5×10^5	7.5×10^6	1.1×10^1	no
cage14	1.5×10^6	2.7×10^7	9.6×10^0	no
crankseg_1	5.3×10^4	1.1×10^7	1.4×10^7 †	no
CurlCurl_2	8.1×10^5	8.9×10^6	4.1×10^5 †	no
CurlCurl_4	2.4×10^6	2.7×10^7	3.4×10^5 †	no
ecology2	1.0×10^6	5.0×10^6	3.2×10^7 †	no
F1	3.4×10^5	2.7×10^7	7.1×10^5 †	yes
FEM_3D_thermal2	1.5×10^5	3.5×10^6	2.5×10^3	no
G3_circuit	1.6×10^6	7.7×10^6	6.0×10^6 †	no
hood	2.2×10^5	9.9×10^6	3.8×10^5 †	no
language	4.0×10^5	1.2×10^6	5.9×10^2	no
marine1	4.0×10^5	6.2×10^6	3.8×10^5 †	yes
mc2depi	5.3×10^5	2.1×10^6	1.3×10^{14}	no
ns3Da	2.0×10^4	1.7×10^6	5.6×10^2	yes
parabolic_fem	5.3×10^5	3.7×10^6	2.1×10^5 †	yes
poisson3Db	8.6×10^4	2.4×10^6	2.6×10^3	yes
pwtk	2.2×10^5	1.2×10^7	6.9×10^5 †	no
rajat31	4.7×10^6	2.0×10^7	4.0×10^6	no
stomach	2.1×10^5	3.0×10^6	2.9×10^1	no
t2em	9.2×10^5	4.6×10^6	2.2×10^5 †	no
thermal2	1.2×10^6	8.6×10^6	1.5×10^6 †	yes
tmt_unsym	9.2×10^5	4.6×10^6	2.3×10^8 †	no
torso2	1.2×10^5	1.0×10^6	2.0×10^1	no
torso3	2.6×10^5	4.4×10^6	9.5×10^1	no
venkat01	6.2×10^4	1.7×10^6	1.3×10^5 †	yes

2. the residual approximation improving by a factor of 10^{-10}; and
3. the residual approximation improving by a factor of 10^{-6} for the first restart and the same number of inner iterations is used after that.

Mixed-precision GMRES was only tested with the third strategy while the other implementations were tested with each. In the latter cases, the strategy with the smallest median was plotted; this ensured that the baseline was not penalized by the choice of restart strategy.

First, the results for unpreconditioned GMRES are shown in Fig. 4.10. The average speedups for the mixed-precision approach were 18 % for MGS and 61 % for CGS2. Furthermore, it provided a speedup for most of the tested matrices and, with CGS2, almost doubled the performance for many of the matrices; only a few matrices saw a slowdown. The single-precision implementation satisfied the target accuracy on only 17 or 16, respectively, of the 23 problems and had average speedups of 0 % and 35 % on the remaining problems. Table 4.3 shows the total number of inner iterations for each test; note that mixed precision required noticeably more iterations in only a few cases. An interesting exception is `rajat31`, which needed significantly fewer iterations for reduced precision; I speculate that the floating-point error happened to perturb the Krylov subspace to better contain the solution. For the matrices with comparable iteration counts, mixed precision with CGS2 can almost achieve a 2× speedup on many of them; however, some obtained only modest speedups. These reduced speedups correlate with the matrices having many nonzeros per row relative to the size of the basis; thus, accessing matrix indices is still a substantial portion of the data moment. As per Table 4.2, even matrices with a condition number larger than 10^7 could be solved more efficiently with the mixed-precision approach, although this may depend on the right-hand side. Finally, the matrices with SuiteSparse-provided right-hand sides behaved similarly to those with generated right-hand sides.

Second, Fig. 4.11 shows the results for a scalar Jacobi preconditioner. The average speedups for mixed precision were 12 % and 50 % for MGS and CGS2, respectively. The single-precision implementation failed to satisfy the target accuracy in 11 of the 30 problems; the remaining problems had average speedups of −8 % (i.e., a slowdown) and 23 %, respectively. Using single precision for just the preconditioner led to one failure and provided an average slowdown of 8 % for both orthogonalization schemes on the successful matrices. For MGS, there were five matrices where the mixed-precision implementation failed to outperform double precision; for CGS2, mixed precision improved performance for all matrices except `language`. The total number of inner iterations on a given matrix was mostly consistent for different configurations, as is shown in Table 4.4. Like the unpreconditioned case, better speedups were achieved on matrices with few nonzeros per row relative to the size of the basis.

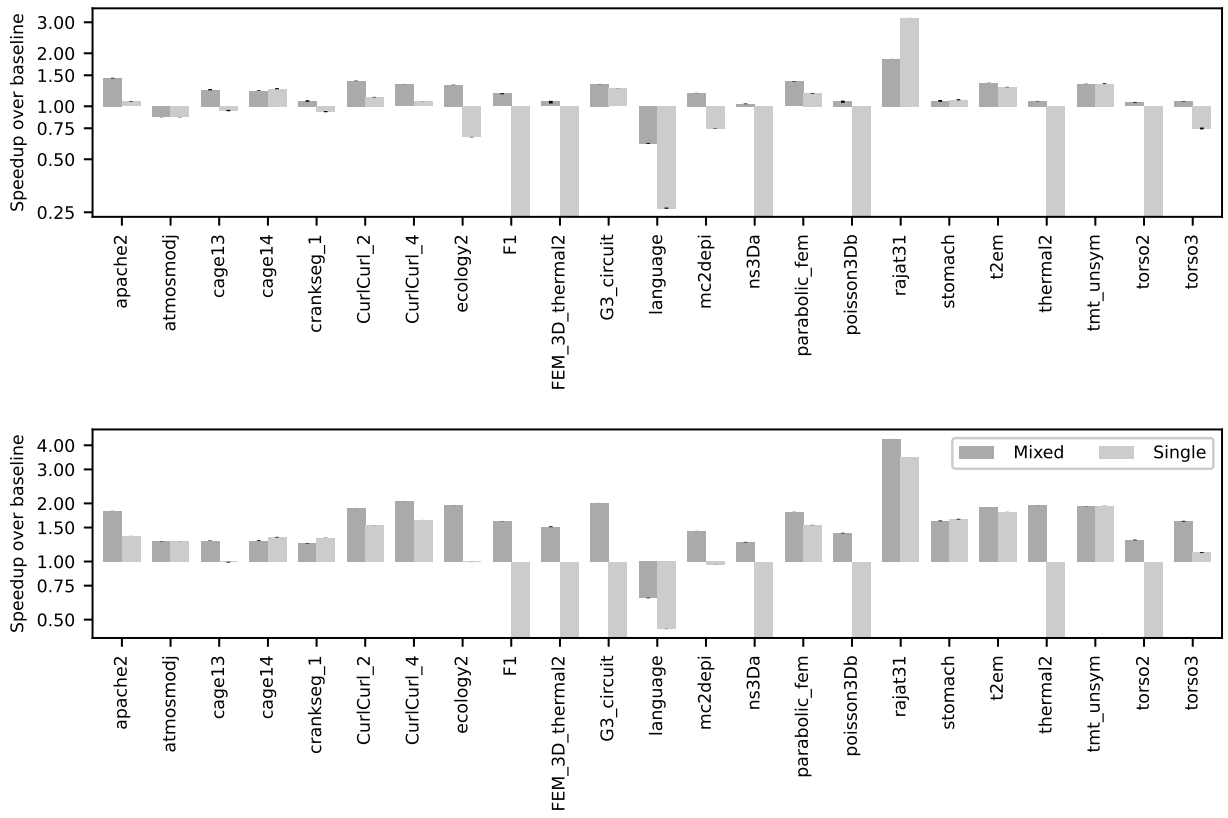


Figure 4.10: GPU performance of unpreconditioned GMRES. Top plot uses MGS and bottom uses CGS2. © 2021 IEEE

Table 4.3: Inner iteration counts for unpreconditioned GMRES. © 2021 IEEE

	Double		Mixed		Single	
Matrix	MGS	CGS2	MGS	CGS2	MGS	CGS2
apache2	21 400	21 400	21 500	21 500	29 200	29 300
atmosmodj	200	200	300	300	300	300
cage13	30	30	30	30	45	45
cage14	30	30	30	30	30	30
crankseg_1	6 300	6 200	6 300	6 300	7 300	5 900
CurlCurl_2	9 900	9 900	9 900	9 900	12 300	12 300
CurlCurl_4	21 100	21 100	21 100	21 100	26 400	26 400
ecology2	900	900	900	900	1 800	1 800
F1	29 200	29 200	29 200	29 200	-	-
FEM_3D_thermal2	300	300	300	300	-	-
G3_circuit	28 200	28 200	27 500	28 200	29 200	-
language	29	29	58	58	145	87
mc2depi	10 400	10 200	12 100	12 900	19 500	19 500
ns3Da	1 400	1 400	1 400	1 400	-	-
parabolic_fem	3 500	3 500	3 500	3 500	4 100	4 100
poisson3Db	300	300	300	300	-	-
rajat31	4 000	4 000	2 900	2 000	1 700	2 500
stomach	300	300	300	300	300	300
t2em	4 800	4 800	4 800	4 800	5 100	5 100
thermal2	21 100	28 500	25 600	28 500	-	-
tmt_unsym	500	500	500	500	500	500
torso2	80	80	80	80	-	-
torso3	200	200	200	200	300	300

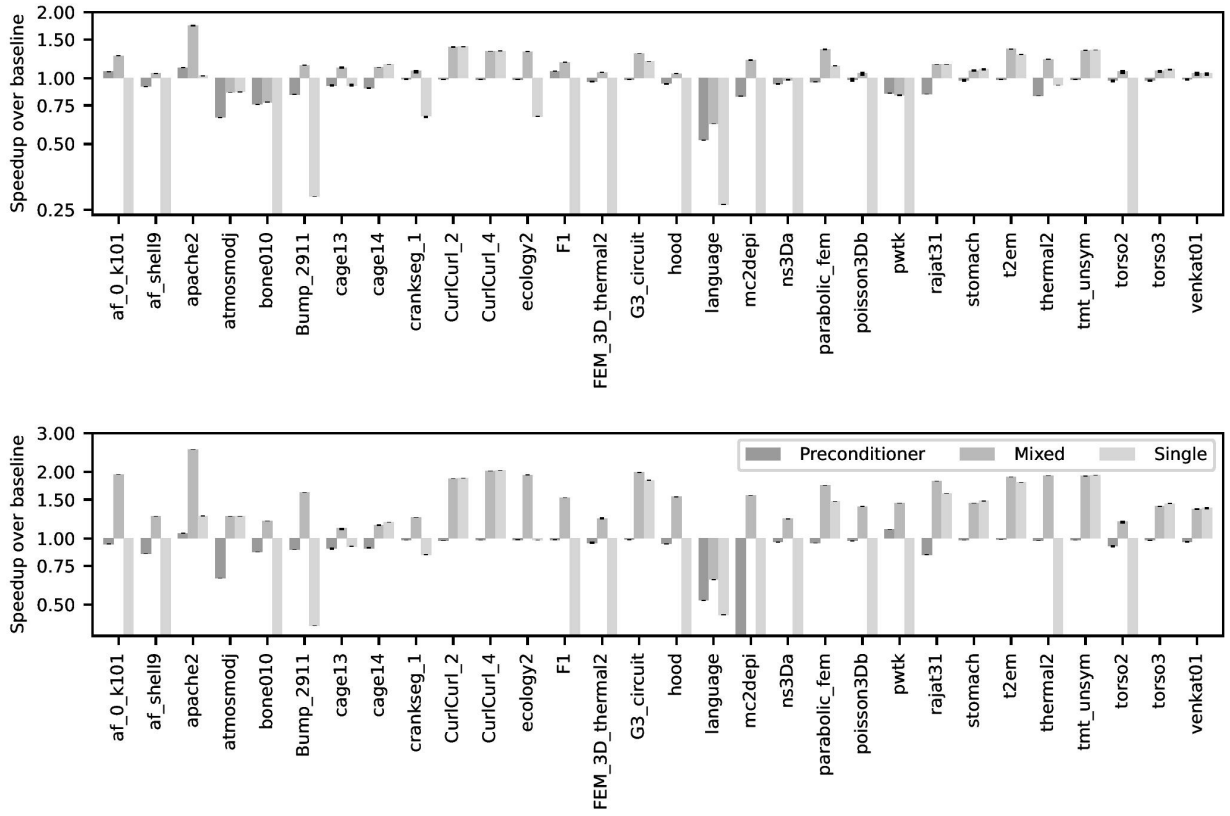


Figure 4.11: GPU performance of GMRES with a scalar Jacobi preconditioner. Top plot uses MGS and bottom uses CGS2. © 2021 IEEE

Table 4.4: Inner iteration counts for GMRES with a scalar Jacobi preconditioner. © 2021 IEEE

	Double		Preconditioner		Mixed		Single	
Matrix	MGS	CGSR	MGS	CGSR	MGS	CGSR	MGS	CGSR
af_0_k101	18 200	14 300	16 800	14 900	19 800	12 700	-	-
af_shell9	21 000	20 500	22 700	23 600	27 600	28 400	-	-
apache2	11 700	11 700	10 400	10 900	9 800	8 500	16 700	17 200
atmosmodj	200	200	300	300	300	300	300	300
bone010	14 600	19 000	19 100	21 500	25 600	28 200	-	-
Bump_2911	3 500	3 500	4 100	3 900	4 100	4 100	16 400	16 600
cage13	22	19	22	22	22	22	33	33
cage14	22	22	22	19	22	22	22	22
crankseg_1	800	800	800	800	800	800	1 300	1 200
CurlCurl_2	1 500	1 500	1 500	1 500	1 500	1 500	1 500	1 500
CurlCurl_4	1 900	1 900	1 900	1 900	1 900	1 900	1 900	1 900
ecology2	800	800	800	800	800	800	1 600	1 600
F1	3 600	3 600	3 300	3 600	3 600	3 800	-	-
FEM_3D_thermal2	60	60	60	60	60	60	-	-
G3_circuit	1 100	1 100	1 100	1 100	1 100	1 100	1 200	1 200
hood	3 900	3 900	4 100	4 100	4 100	4 000	-	-
language	29	29	58	58	58	58	145	87
mc2depi	11 000	10 600	13 200	-	12 600	12 200	-	-
ns3Da	14 300	14 400	14 800	14 600	15 000	14 900	-	-
parabolic_fem	3 600	3 600	3 700	3 700	3 700	3 700	4 400	4 400
poisson3Db	400	400	400	400	400	400	-	-
pwtk	13 500	18 400	15 700	16 600	17 700	20 200	-	-
rajat31	600	600	700	700	700	700	700	800
stomach	130	100	130	100	130	130	130	130
t2em	4 800	4 800	4 800	4 800	4 800	4 800	5 100	5 100
thermal2	21 400	25 300	25 400	25 500	22 700	25 500	29 800	-
tmt_unsym	500	500	500	500	500	500	500	500
torso2	56	47	56	56	56	56	-	-
torso3	134	100	100	100	134	134	134	134
venkat01	96	96	96	96	96	96	96	96

Next, Fig. 4.12 shows the results for an ILU preconditioner. The average speedup for mixed precision was −9 % and −7 % for MGS and CGS2, respectively (i.e., a slowdown). For the single-precision preconditioner, the speedups were instead −8 % and −9 %. The single-precision implementation failed to produce an accurate enough solution to 7 of the problems; the remaining problems had average speedups of −11 % and −10 %, respectively. There are a few factors that contribute to the slowdowns. First, the sparse triangular solves have limited parallelism for the GPU to exploit; this results in poor utilization of the GPU bandwidth, which limits the benefit of reducing the size of the data. Furthermore, the poor performance of the triangular solves causes them to make up a large part of the performance. Second, because the factorization is always done in double precision, it is a fixed cost in the performance. Third, because of the effectiveness of the preconditioner, double precision can solve most problems without restarting; however, the mixed-precision implementation must always restart at least once. These restarts incur overhead from extra computation and from reducing the rate of convergence for some matrices. Table 4.5 shows the relevant iteration counts and that the baseline can converge without restarting for 11 out of the 29 matrices.

Finally, Fig. 4.13 shows the results for an ILU preconditioner with five Jacobi iterations for triangular solves. The speedup was 8 % and 13 % for MGS and CGS2, respectively. Additionally, the single-precision preconditioner was able to achieve some improvement overall, with speedups of 3 % and 4 %, respectively. The single-precision implementation failed to produce an accurate enough solution to three of the problems; the remaining problems had average speedups of 4 % and 4 %, respectively. Note that while the triangular solves have been improved, the other factors limiting the improvement of the regular ILU preconditioner remain. Table 4.6 shows the iteration counts and that the baseline needed to restart on only 5 of the 13 matrices.

There does not appear to be any previous experimental performance comparisons between MGS and CGS2 on modern GPUs. As this work tested both schemes, it was natural to compare them; Table 4.7 does so in the context of the overall GMRES performance. In spite of requiring twice as much work, CGS2-GMRES provides better performance than MGS-GMRES. Recall that MGS requires j dot-products alternated with j vector additions for the jth inner iteration while CGS2 requires only four matrix-vector products. Thus, CGS2 launches fewer kernels, which reduces overhead; furthermore, high GPU utilization is easier to obtain with large kernels than with small ones. The better speedup for mixed- and single-precision implementations likely comes from reductions in the cost of the kernel's execution making the kernel launches more costly relative to the total time. Similarly, when GMRES uses a cheaper preconditioner, it spends a higher percentage of its runtime doing the orthogonalization, which results in a larger performance difference. Thus, the traditional

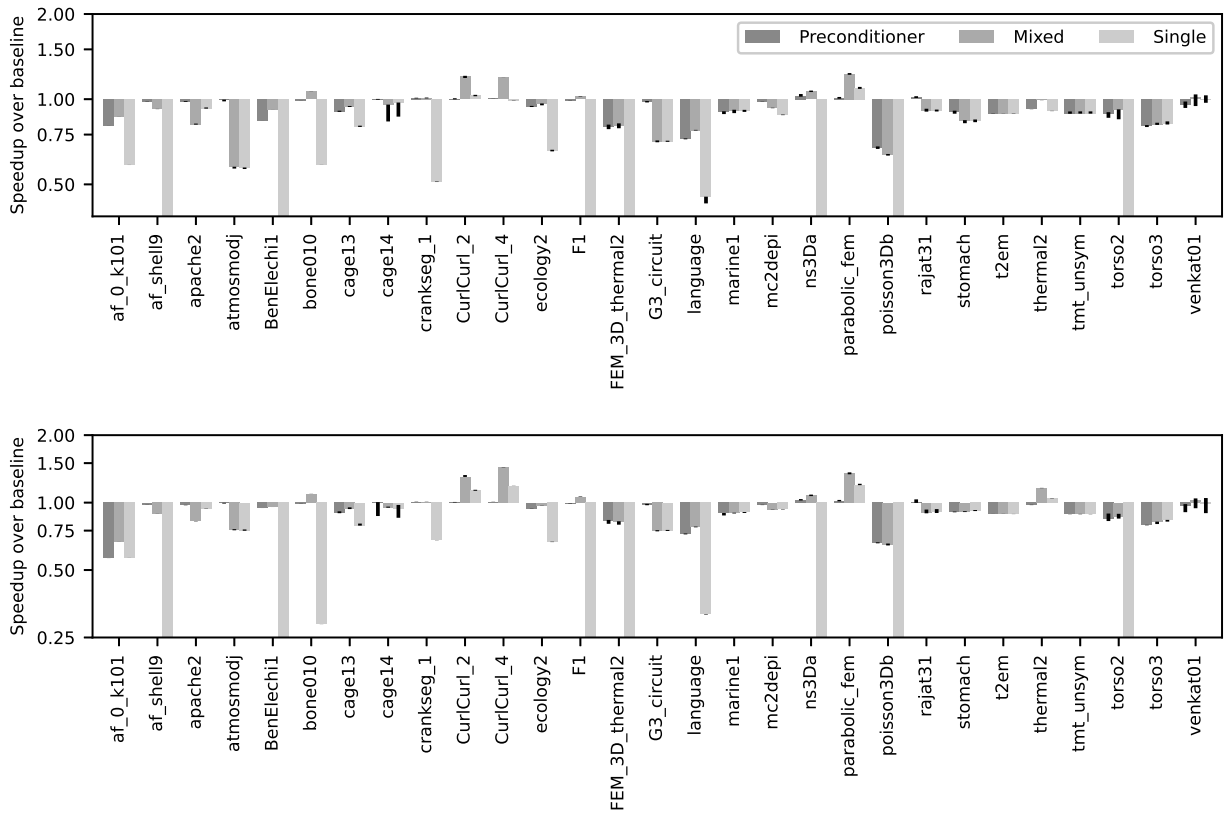


Figure 4.12: GPU performance of GMRES with an ILU preconditioner. Top plot uses MGS and bottom uses CGS2. © 2021 IEEE

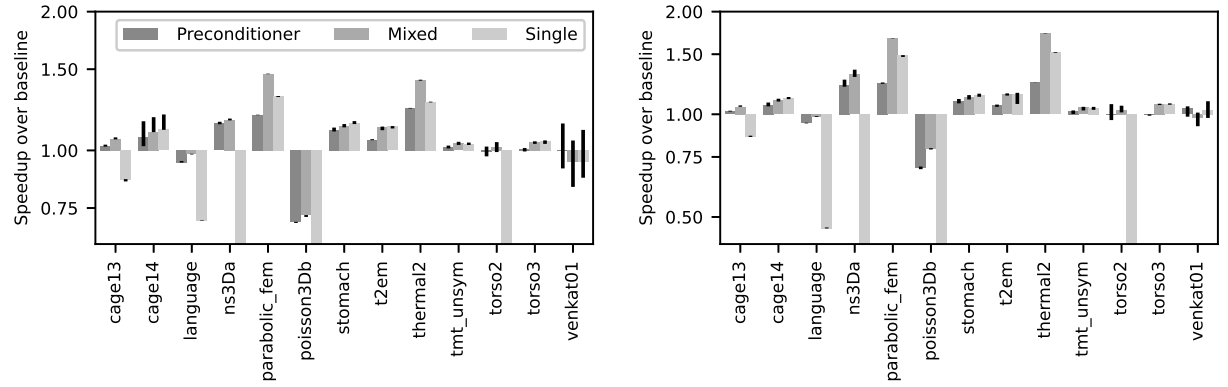


Figure 4.13: GPU performance of GMRES with an ILU preconditioner using five Jacobi iterations for triangular solves. Top plot uses MGS and bottom uses CGS2. © 2021 IEEE

Table 4.5: Inner iteration counts for GMRES with an ILU preconditioner. © 2021 IEEE

	Double		Preconditioner		Mixed		Single	
Matrix	MGS	CGSR	MGS	CGSR	MGS	CGSR	MGS	CGSR
af_0_k101	4 500	4 400	5 500	7 700	5 200	6 600	7 700	7 800
af_shell9	2 200	2 200	2 200	2 200	2 400	2 500	-	-
apache2	600	600	600	600	800	800	700	700
atmosmodj	82	82	82	82	200	164	200	164
BenElechi1	3 900	4 300	4 600	4 500	4 200	4 500	-	-
bone010	1 200	1 200	1 200	1 200	1 200	1 200	2 200	4 700
cage13	7	7	8	8	8	8	12	12
cage14	7	7	7	7	8	8	8	8
crankseg_1	200	200	200	200	200	200	400	300
CurlCurl_2	600	600	600	600	600	600	700	700
CurlCurl_4	1 400	1 400	1 400	1 400	1 400	1 400	1 700	1 700
ecology2	200	200	200	200	200	200	300	300
F1	1 000	1 000	1 000	1 000	1 000	1 000	-	-
FEM_3D_thermal2	10	10	12	12	12	12	-	-
G3_circuit	200	200	200	200	300	300	300	300
language	9	9	14	14	14	14	28	42
marine1	300	300	300	300	300	300	300	300
mc2depi	1 700	1 700	1 600	1 600	1 700	1 700	1 800	1 700
ns3Da	200	200	200	200	200	200	-	-
parabolic_fem	800	800	800	800	800	800	900	900
poisson3Db	100	100	174	174	174	174	-	-
rajat31	10	10	9	9	18	18	18	18
stomach	17	17	18	18	20	18	20	18
t2em	600	600	600	600	600	600	600	600
thermal2	4 800	5 100	5 100	5 100	5 200	5 100	5 700	5 700
tmt_unsym	200	200	200	200	200	200	200	200
torso2	11	11	12	12	12	12	-	-
torso3	35	35	48	48	48	48	48	48
venkat01	12	12	12	12	12	12	12	12

Table 4.6: Inner iteration counts for GMRES preconditioned with ILU using five Jacobi iterations for triangular solves. © 2021 IEEE

	Double		Preconditioner		Mixed		Single	
Matrix	MGS	CGSR	MGS	CGSR	MGS	CGSR	MGS	CGSR
cage13	7	7	8	8	8	8	12	12
cage14	7	7	8	8	8	8	8	8
language	9	9	14	14	14	14	21	35
ns3Da	200	200	200	200	200	200	-	-
parabolic_fem	800	800	800	800	800	800	900	900
poisson3Db	100	100	174	174	174	174	-	-
stomach	21	21	22	22	22	22	22	22
t2em	800	800	800	800	800	800	800	800
thermal2	5 100	5 100	5 000	5 100	5 100	5 100	5 700	5 800
tmt_unsym	200	200	200	200	200	200	200	200
torso2	11	11	12	12	12	12	-	-
torso3	48	48	64	64	64	64	64	64
venkat01	16	16	16	16	16	16	16	16

Table 4.7: Average speedups of CGS2-GMRES over MGS-GMRES on GPU. © 2021 IEEE

Preconditioner	Double Speedup	Mixed Speedup	Single Speedup
Identity	36%	87%	181%
Jacobi	34%	79%	116%
ILU	4%	7%	4%
ILU with Jacobi	8%	13%	4%

guidance to prefer MGS over CGS2 does not hold for GPUs without support for fusing the dot product's reduction into other kernels.

4.4 Conclusions

This work demonstrates that selectively reducing precision is an effective way to improve the performance of GMRES. A common concern when discussion mixed-precision GMRES is whether the reductions in accuracy will result in significantly more iterations. However, both theoretical and experimental results indicate that this is not at issue for most problems. On the other hand, a carefully selected restart criterion is important to ensure GMRES restarts before improvement stalls; otherwise, the iteration count will be inflated by unusable work. A natural question is whether the better stability of CGS2 can partially compensate for reduced precision compared to MGS; Section 4.3.2 shows that for most problems, there is little difference in iteration count. However on GPU systems, CGS2 is preferable anyways, due to its higher performance.

There have been several publications on mixed-precision GMRES since content from this chapter was published [4, 97, 136, 139]. Three of them directly build on my work [97, 136, 139], while the last one proposes an alternative combination of precisions called CB-GMRES [4]. The latter reduces the precision less aggressively, changing just the storage of the Krylov basis. I expect that there are fewer cases of poor convergence compared to my approach. Furthermore, because CB-GMRES modifies only the storage of one matrix, it can easily support alternative data representations like posits [69] or SZ compression [40]. However, because of the larger reduction in data movement, I expect Algorithm 4.1 to outperform CB-GMRES using single-precision as the reduced precision in most cases. Unfortunately, there have been no serious comparisons of the two approaches. All of the other work on mixed-precision GMRES has used the trivial restart strategy based on the number of inner iterations [4, 12, 97, 136, 139], even those written after the publication of Section 4.2's content. When GMRES converges quickly, this can result in many wasted iterations. All three of the subsequent works using Algorithm 4.1 have explicitly noted a significant increase in the number of iterations for quickly converging problems but did not consider using an alternative restart strategy [97, 136, 139].

Bibliography

[1] A. Abdelfattah, H. Anzt, E. G. Boman *et al.*, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, Mar. 2021. https://doi.org/10.1177/10943420211003313 (cited on p. 8)

[2] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–10. https://doi.org/10.1145/3079079.3079103 (cited on p. 66)

[3] E. Agullo, F. Cappello, S. Di *et al.*, "Exploring variable accuracy storage through lossy compression techniques in numerical linear algebra: A first application to flexible GMRES," Inria Bordeaux Sud-Ouest, Research Report RR-9342, 2020. (cited on pp. 9, 10)

[4] J. I. Aliaga, H. Anzt, T. Grützmacher *et al.*, "Compressed basis GMRES on high-performance graphics processing units," *The International Journal of High Performance Computing Applications*, Aug. 2022. https://doi.org/10.1177/10943420221115140 (cited on pp. 9, 10, 115)

[5] ——, "Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 14, p. e6515, 2022. https://doi.org/10.1002/cpe.6515 (cited on p. 10)

[6] P. Amestoy, A. Buttari, N. J. Higham *et al.*, "Five-precision GMRES-based iterative refinement," The University of Manchester, Manchester, UK, MIMS EPrint 2021.5, 2021. (cited on p. 9)

[7] P. Amodio and F. Mazzia, "A new approach to backward error analysis of LU factorization," *BIT Numerical Mathematics*, vol. 39, no. 3, pp. 385–402, Sep. 1999. https://doi.org/10.1023/A:1022358300517 (cited on pp. 33, 60, 62)

[8] K. D. Andersen, "A modified Schur-complement method for handling dense columns in interior-point methods for linear programming," *ACM Transactions on Mathematical Software*, vol. 22, no. 3, pp. 348–356, Sep. 1996. https://doi.org/10.1145/232826.232937 (cited on p. 8)

[9] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 650–661. https://doi.org/10.1007/978-3-662-48096-0_50 (cited on p. 104)

[10] H. Anzt, G. Flegar, T. Grützmacher, and E. S. Quintana-Ortí, "Toward a modular precision ecosystem for high-performance computing," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1069–1078, Nov. 2019. https://doi.org/10.1177/1094342019846547 (cited on p. 10)

[11] H. Anzt, V. Heuveline, and B. Rocker, "Mixed precision iterative refinement methods for linear systems: Convergence analysis based on Krylov subspace methods," in *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing - Volume 2*, ser. PARA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 237–247. https://doi.org/10.1007/978-3-642-28145-7_24 (cited on pp. 9, 10)

[12] ——, "An error correction solver for linear systems: Evaluation of mixed precision implementations," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 58–70. https://doi.org/10.1007/978-3-642-19328-6_8 (cited on pp. 9, 115)

[13] H. Avron, A. Druinsky, and S. Toledo, "Spectral condition-number estimation of large sparse matrices," *Numerical Linear Algebra with Applications*, vol. 26, no. 3, May 2019. https://doi.org/10.1002/nla.2235 (cited on p. 104)

[14] M. Baboulin, D. Becker, G. Bosilca *et al.*, "An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems," *Parallel Computing*, vol. 40, no. 7, pp. 213–223, Jul. 2014. https://doi.org/10.1016/j.parco.2013.12.003 (cited on pp. 7, 42)

[15] M. Baboulin, A. Buttari, J. Dongarra *et al.*, "Accelerating scientific computations with mixed precision algorithms," *CoRR*, vol. abs/0808.2794, Dec. 2009. https://doi.org/10.1016/j.cpc.2008.11.005 (cited on p. 9)

[16] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Transactions on Mathematical Software*, vol. 39, no. 2, pp. 1–13, Feb. 2013. https://doi.org/10.1145/2427023.2427025 (cited on pp. 7, 36, 42, 44, 46)

[17] M. Baboulin, R. A. Dongarra Jack, T. Stanimire, and Y. Ichitaro, "Dense symmetric indefinite factorization on GPU accelerated architectures," in *Parallel Processing and Applied Mathematics. PPAM 2015*, ser. Lecture Notes in Computer Science, Volume 9573, R. Wyrzykowski, E. Deelman, J. Dongarra *et al.*, Eds. Springer, Apr. 2016. https://doi.org/10.1007/978-3-319-32149-3_9 (cited on p. 7)

[18] M. Baboulin, A. Jamal, and M. Sosonkina, "Using random butterfly transformations in parallel Schur complement-based preconditioning," in *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2015, pp. 649–654. https://doi.org/10.15439/2015F177 (cited on p. 7)

[19] M. Baboulin, A. Khabou, and A. Rémy, "A randomized LU-based solver using GPU and Intel Xeon Phi accelerators," in *Euro-Par 2015: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, S. Hunold, A. Costan, D. Giménez *et al.*, Eds. Springer International Publishing, Dec. 2015, pp. 175–184. https://doi.org/10.1007/978-3-319-27308-2_15 (cited on p. 7)

[20] M. Baboulin, X. S. Li, and F.-H. Rouet, "Using random butterfly transformations to avoid pivoting in sparse direct methods," in *High Performance Computing for Computational Science – VECPAR 2014*, ser. Lecture Notes in Computer Science, M. Daydé, O. Marques, and K. Nakajima, Eds. Springer International Publishing, Apr. 2015, pp. 135–144. https://doi.org/10.1007/978-3-319-17353-5_12 (cited on p. 7)

[21] M. Baboulin, S. Tomov, and J. Dongarra, "Some issues in dense linear algebra for multicore and special purpose architectures," in *PARA 2008, 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, 2008. (cited on p. 7)

[22] J. L. Barlow and H. Zha, "Growth in Gaussian elimination, orthogonal matrices, and the 2-norm," *SIAM Journal on Matrix Analysis and Applications*, vol. 19, no. 3, pp. 807–815, Jan. 1998. https://doi.org/10.1137/S0895479896309912 (cited on pp. 33, 60)

[23] D. W. Barron and H. P. F. Swinnerton-Dyer, "Solution of simultaneous linear equations using a magnetic-tape store," *The Computer Journal*, vol. 3, no. 1, pp. 28–33, Jan. 1960. https://doi.org/10.1093/comjnl/3.1.28 (cited on p. 6)

[24] D. Becker, M. Baboulin, and J. Dongarra, "Reducing the amount of pivoting in symmetric indefinite systems," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 133–142. https://doi.org/10.1007/978-3-642-31464-3_14 (cited on p. 7)

[25] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017. https://doi.org/10.1137/141000671 (cited on p. 78)

[26] D. Bielich, J. Langou, S. Thomas *et al.*, "Low-synch Gram–Schmidt with delayed reorthogonalization for Krylov solvers," *Parallel Computing*, vol. 112, p. 102940, Sep. 2022. https://doi.org/10.1016/j.parco.2022.102940 (cited on p. 4)

[27] A. Bienz, L. Olson, and W. Gropp, "Node-aware improvements to Allreduce," in *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. Denver, CO, USA: IEEE Press, Nov. 2019, pp. 19–28. https://doi.org/10.1109/ExaMPI49596.2019.00008 (cited on p. 18)

[28] A. Bouras and V. Frayssé, "Inexact matrix-vector products in Krylov methods for solving linear systems: A relaxation strategy," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 3, pp. 660–678, Jan. 2005. https://doi.org/10.1137/S0895479801384743 (cited on p. 10)

[29] A. Buttari, J. Dongarra, J. Kurzak *et al.*, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, no. 4, pp. 17:1–17:22, Jul. 2008. https://doi.org/10.1145/1377596.1377597 (cited on p. 10)

[30] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, Jan. 2009. https://doi.org/10.1016/j.parco.2008.10.002 (cited on p. 6)

[31] B. Carpentieri, "Krylov subspace methods for big data analysis of large computational electromagnetics applications," in *Modern Management Based on Big Data II and*

Machine Learning and Intelligent Systems III, A. J. Tallón-Ballesteros, Ed. IOS Press, Dec. 2021, pp. 57–65. https://doi.org/10.3233/FAIA210232 (cited on p. 1)

[32] E. Carson and N. J. Higham, "A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, Jan. 2017. https://doi.org/10.1137/17M1122918 (cited on pp. 9, 86)

[33] ——, "Accelerating the solution of linear systems by iterative refinement in three precisions," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, Jan. 2018. https://doi.org/10.1137/17M1140819 (cited on pp. 54, 67, 71, 86)

[34] Chang Zhai, Yingyu Liu, Shugang Jiang *et al.*, "Integrated simulation and analysis of super large slotted waveguide array," *Applied Computational Electromagnetics Society Journal*, vol. 35, no. 7, pp. 813–820, Jul. 2020. (cited on p. 1)

[35] A. Charara, D. Keyes, and H. Ltaief, "Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs," *ACM Transactions on Mathematical Software*, vol. 45, no. 2, pp. 15:1–15:28, May 2019. https://doi.org/10.1145/3267101 (cited on p. 66)

[36] A. T. Chronopoulos and C. W. Gear, "S-step iterative methods for symmetric linear systems," *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, Feb. 1989. https://doi.org/10.1016/0377-0427(89)90045-9 (cited on p. 4)

[37] M. Clark, R. Babich, K. Barros *et al.*, "Solving lattice QCD systems of equations using mixed precision solvers on GPUs," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, 2010. https://doi.org/10.1016/j.cpc.2010.05.002 (cited on p. 10)

[38] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. https://doi.org/10.1145/2049662.2049663 (cited on pp. 90, 99)

[39] J. W. Demmel, N. J. Higham, and R. S. Schreiber, "Stability of block LU factorization," *Numerical Linear Algebra with Applications*, vol. 2, no. 2, pp. 173–190, Mar. 1995. https://doi.org/10.1002/nla.1680020208 (cited on pp. 57, 58, 64, 65)

[40] S. Di and F. Cappello, "Fast Error-Bounded Lossy HPC Data Compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730–739. https://doi.org/10.1109/IPDPS.2016.11 (cited on p. 115)

[41] S. Donfack, J. Dongarra, M. Faverge *et al.*, "A survey of recent developments in parallel implementations of Gaussian elimination," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1292–1309, Jun. 2015. https://doi.org/10.1002/cpe.3306 (cited on pp. 6, 46, 52, 75)

[42] J. Drkošová, A. Greenbaum, M. Rozložník, and Z. Strakoš, "Numerical stability of GMRES," *BIT Numerical Mathematics*, vol. 35, no. 3, pp. 309–330, Sep. 1995. https://doi.org/10.1007/BF01732607 (cited on p. 86)

[43] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, 2nd ed. Oxford, UK: Oxford University Press, Mar. 2017. (cited on pp. 6, 16)

[44] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 889–901, Jan. 1999. https://doi.org/10.1137/S0895479897317661 (cited on p. 8)

[45] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, Dec. 2014. https://doi.org/10.1016/j.jpdc.2014.07.003 (cited on pp. 90, 101, 104)

[46] M. Emans and A. van der Meer, "Mixed-precision AMG as linear equation solver for definite systems," *Procedia Computer Science*, vol. 1, no. 1, pp. 175–183, May 2010. https://doi.org/10.1016/j.procs.2010.04.020 (cited on p. 10)

[47] T. Endo and K. Taura, "Highly latency tolerant Gaussian elimination," in *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.* Seattle, WA, USA: IEEE Press, Nov. 2005, pp. 91–98. https://doi.org/10.1109/GRID.2005.1542729 (cited on p. 5)

[48] M. Faverge, J. Herrmann, J. Langou *et al.*, "Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 32–46, Nov. 2015. https://doi.org/10.1016/j.jpdc.2015.06.007 (cited on p. 6)

[49] W.-c. Feng and K. Cameron, "The Green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, Dec. 2007. https://doi.org/10.1109/MC.2007.445 (cited on p. 30)

[50] P. Fischer, E. Merzari, M. Min *et al.*, "Highly optimized full-core reactor simulations on Summit," arXiv:2110.01716, Oct. 2021. https://doi.org/10.48550/arXiv.2110.01716 (cited on pp. 1, 3)

[51] V. Frayssé, L. Giraud, and H. Kharraz-Aroussi, "On the influence of the orthogonalization scheme on the parallel performance of GMRES," in *Euro-Par'98 Parallel Processing*, ser. Lecture Notes in Computer Science, D. Pritchard and J. Reeve, Eds. Berlin, Heidelberg: Springer, 1998, pp. 751–762. https://doi.org/10.1007/BFb0057927 (cited on p. 84)

[52] F. R. Gantmacher, *The Theory of Matrices*. Providence, RI, USA: American Mathematical Soc., 1959, vol. 1. (cited on p. 22)

[53] M. Gates, J. Kurzak, A. Charara *et al.*, "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*, ser. SC '19. Denver, CO, USA: Association for Computing Machinery, Nov. 2019, pp. 1–18. https://doi.org/10.1145/3295500.3356223 (cited on p. 3)

[54] G. A. Geist and C. H. Romine, "LU factorization algorithms on distributed-memory multiprocessor architectures," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 4, pp. 639–649, Jul. 1988. https://doi.org/10.1137/0909042 (cited on pp. 5, 6)

[55] L. Giraud, J. Langou, and M. Rozložník, "The loss of orthogonality in the Gram-Schmidt orthogonalization process," *Comput. Math. Appl.*, vol. 50, no. 7, pp. 1069–1075, Oct. 2005. https://doi.org/10.1016/j.camwa.2005.08.009 (cited on pp. 84, 94, 99)

[56] L. Giraud, A. Haidar, and L. T. Watson, "Mixed-precision preconditioners in parallel domain decomposition solvers," in *Domain Decomposition Methods in Science and Engineering XVII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 357–364. https://doi.org/10.1007/978-3-540-75199-1_44 (cited on p. 9)

[57] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numerische Mathematik*, vol. 101, no. 1, pp. 87–100, Jul. 2005. https://doi.org/10.1007/s00211-005-0615-4 (cited on pp. 86, 87)

[58] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: The John Hopkins University Press, 2013. (cited on p. 50)

[59] A. González, "Trends in processor architecture," in *Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms: A Cross-layer Approach*, W. Fornaciari and D. Soudris, Eds. Cham: Springer International Publishing, Oct. 2018, pp. 23–42. https://doi.org/10.1007/978-3-319-91962-1_2 (cited on p. 1)

[60] S. Gratton, E. Simon, D. Titley-Peloquin, and P. Toint, "Exploiting variable precision in GMRES," arXiv:1907.10550, Feb. 2020. https://doi.org/10.48550/arXiv.1907.10550 (cited on pp. 87, 89)

[61] S. Gratton, E. Simon, D. Titley-Peloquin, and P. L. Toint, "A Note on Inexact Inner Products in GMRES," *SIAM Journal on Matrix Analysis and Applications*, vol. 43, no. 3, pp. 1406–1422, Sep. 2022. https://doi.org/10.1137/20M1320018 (cited on pp. 10, 86)

[62] A. Greenbaum, M. Rozložník, and Z. Strakoš, "Numerical behaviour of the modified Gram-Schmidt GMRES implementation," *BIT Numerical Mathematics*, vol. 37, no. 3, pp. 706–719, Sep. 1997. https://doi.org/10.1007/BF02510248 (cited on p. 94)

[63] A. Greenbaum, V. Pták, and Z. Strakoš, "Any Nonincreasing Convergence Curve is Possible for GMRES," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 3, pp. 465–469, Jul. 1996. https://doi.org/10.1137/S0895479894275030 (cited on pp. 92, 95)

[64] T. N. E. Greville, "Note on the generalized inverse of a matrix product," *SIAM Review*, vol. 8, no. 4, pp. 518–521, Oct. 1966. https://doi.org/10.1137/1008107 (cited on p. 56)

[65] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Austin, Texas: IEEE Press, Nov. 2008, pp. 1–12. https://doi.org/10.1109/SC.2008.5214287 (cited on p. 5)

[66] ——, "CALU: A communication optimal LU factorization algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1317–1350, Oct. 2011. https://doi.org/10.1137/100788926 (cited on pp. 5, 6)

[67] D. P. Grossman, "On the problem of the numerical solution of systems of simultaneous linear algebraic equations," *Uspekhi Mat. Nauk*, vol. 5, no. 3(37), pp. 87–103, 1950. (cited on p. 22)

[68] T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí, "Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS," *Software: Practice and Experience*, Oct. 2021. https://doi.org/10.1002/spe.3041 (cited on p. 9)

[69] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, Apr. 2017. https://doi.org/10.14529/jsfi170206 (cited on p. 115)

[70] W. W. Hager, "Updating the inverse of a matrix," *SIAM Review*, vol. 31, no. 2, pp. 221–239, Jun. 1989. https://doi.org/10.1137/1031049 (cited on pp. 8, 49, 50)

[71] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002. https://doi.org/10.1137/1.9780898718027 (cited on pp. 4, 20, 33, 49, 65, 87, 88, 90)

[72] ——, "The matrix computation toolbox," 2002. http://www.ma.man.ac.uk/~higham/mctoolbox/ (cited on p. 44)

[73] N. J. Higham and D. J. Higham, "Large growth factors in Gaussian elimination with pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 10, no. 2, pp. 155–164, Apr. 1989. https://doi.org/10.1137/0610012 (cited on pp. 4, 20)

[74] N. J. Higham and T. Mary, "A new approach to probabilistic rounding error analysis," *SIAM Journal on Scientific Computing*, vol. 41, no. 5, pp. A2815–A2835, Jan. 2019. https://doi.org/10.1137/18M1226312 (cited on pp. 20, 33, 62, 90)

[75] N. J. Higham, S. Pranesh, and M. Zounon, "Squeezing a matrix into half precision, with an application to solving linear systems," *SIAM Journal on Scientific Computing*, vol. 41, no. 4, pp. A2536–A2551, Jan. 2019. https://doi.org/10.1137/18M1229511 (cited on p. 3)

[76] W. Hoffmann and K. Potma, "Threshold-pivoting in parallel Gaussian elimination for improved efficiency," in *Proceedings of the First Annual Conference of the Advanced School for Computing and Imaging*. Delft: Technical University Delft, 1995, pp. 63–68. (cited on p. 5)

[77] J. D. Hogg and J. A. Scott, "Pivoting strategies for tough sparse indefinite systems," *ACM Transactions on Mathematical Software*, vol. 40, no. 1, pp. 4:1–4:19, Oct. 2013. https://doi.org/10.1145/2513109.2513113 (cited on pp. 6, 30)

[78] C.-H. Hsu, W.-C. Feng, and J. Archuleta, "Towards efficient supercomputing: A quest for the right metric," in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005, pp. 8 pp.–. https://doi.org/10.1109/IPDPS.2005.440 (cited on p. 30)

[79] D. A. Huckaby and T. F. Chan, "On the convergence of Stewart's QLP algorithm for approximating the SVD," *Numerical Algorithms*, vol. 32, no. 2, pp. 287–316, Apr. 2003. https://doi.org/10.1023/A:1024082314087 (cited on p. 78)

[80] T. Iwashita, K. Suzuki, and T. Fukaya, "An integer arithmetic-based sparse linear solver using a GMRES method and iterative refinement," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Atlanta, GA: IEEE Computer Society Press, Sep. 2020. https://doi.org/10.1109/ScalA51936.2020.0000 (cited on p. 10)

[81] M. Jankowski and H. Woźniakowski, "Iterative refinement implies numerical stability," *BIT Numerical Mathematics*, vol. 17, no. 3, pp. 303–311, Sep. 1977. https://doi.org/10.1007/BF01932150 (cited on p. 86)

[82] W. Kahan, "Numerical linear algebra," *Canadian Mathematical Bulletin*, vol. 9, no. 5, pp. 757–801, Dec. 1966. https://doi.org/10.4153/CMB-1966-083-2 (cited on p. 78)

[83] A. Khan, H. Sim, S. S. Vazhkudai *et al.*, "An analysis of system balance and architectural trends based on Top500 supercomputers," in *The International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2021. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 11–22. https://doi.org/10.1145/3432261.3432263 (cited on pp. viii, 2)

[84] P. Kocher, J. Horn, A. Fogh *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, Jun. 2020. https://doi.org/10.1145/3399742 (cited on p. 1)

[85] P. M. Kogge and W. J. Dally, "Frontier vs the exascale report: Why so long? and are we really there yet?" in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Nov. 2022, pp. 26–35. https://doi.org/10.1109/PMBS56514.2022.00008 (cited on pp. viii, 1, 2)

[86] J. Kurzak, M. Gates, A. Charara *et al.*, "Linear systems solvers for distributed-memory machines with GPU accelerators," in *Euro-Par 2019: Parallel Processing*, ser. Lecture

Notes in Computer Science, R. Yahyapour, Ed. Göttingen, Germany: Springer, Cham, 2019, pp. 495–506. https://doi.org/10.1007/978-3-030-29400-7_35 (cited on p. 26)

[87] G. Kwasniewski, M. Kabic, T. Ben-Nun *et al.*, "On the parallel I/O optimality of linear algebra kernels: Near-optimal matrix factorizations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–15. https://doi.org/10.1145/3458817.3476167 (cited on pp. 4, 5)

[88] C. Lau, E. F. Jaeger, N. Bertelli *et al.*, "AORSA full wave calculations of helicon waves in DIII-D and ITER," *Nuclear Fusion*, vol. 58, no. 6, Apr. 2018. https://doi.org/10.1088/1741-4326/aab96d (cited on p. 1)

[89] O. S. Lawlor, "In-memory data compression for sparse matrices," in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA^3 '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–6. https://doi.org/10.1145/2535753.2535758 (cited on p. 10)

[90] X. S. Li and J. Demmel, "Making sparse Gaussian elimination scalable by static pivoting," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. San Jose, CA, USA: IEEE Computer Society, Nov. 1998, pp. 34–34. https://doi.org/10.1109/SC.1998.10030 (cited on pp. 8, 71)

[91] N. Lindquist, M. Gates, P. Luszczek, and J. Dongarra, "Threshold pivoting for dense LU factorization," in *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*. Dallas, Texas, USA: IEEE Computer Society, Nov. 2022, pp. 34–42. https://doi.org/10.1109/ScalAH56622.2022.00010 (cited on pp. iii, 1, 11, 75)

[92] N. Lindquist, P. Luszczek, and J. Dongarra, "Improving the performance of the GMRES method using mixed-precision techniques," in *Driving Scientific and Engineering Discoveries through the Convergence of HPC, Big Data and AI*. Oak Ridge, TN, USA: Springer, Aug. 2020. https://doi.org/10.1007/978-3-030-63393-6_4 (cited on pp. iii, 1, 84)

[93] ——, "Replacing pivoting in distributed Gaussian elimination with randomized techniques," in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Atlanta, GA, USA: IEEE Press, Nov. 2020, pp. 35–43. https://doi.org/10.1109/ScalA51936.2020.00010 (cited on pp. iii, 32, 42, 52, 75)

[94] ——, "Accelerating restarted GMRES with mixed precision arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 1027–1037, Apr. 2022. https://doi.org/10.1109/TPDS.2021.3090757 (cited on pp. iii, 1, 84)

[95] ——, "Using additive modifications in LU factorization instead of pivoting," in *Proceedings of the 37th ACM International Conference on Supercomputing*. Orlando, FL, USA: ACM, Jun. 2023. https://doi.org/10.1145/3577193.3593731 (cited on pp. iii, 1, 32)

[96] Y. Liu and H. Zhu, "A survey of the research on power management techniques for high-performance systems," *Software: Practice and Experience*, vol. 40, no. 11, pp. 943–964, 2010. https://doi.org/10.1002/spe.952 (cited on p. 30)

[97] J. A. Loe, C. A. Glusa, I. Yamazaki *et al.*, "Experimental evaluation of multiprecision strategies for GMRES on GPUs," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Jun. 2021, pp. 469–478. https://doi.org/10.1109/IPDPSW52791.2021.00078 (cited on pp. 9, 94, 115)

[98] J. Malard, "Threshold pivoting for dense LU factorization on distributed memory multiprocessors," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Albuquerque, NM, USA: Association for Computing Machinery, Nov. 1991, pp. 600–607. https://doi.org/10.1145/125826.126136 (cited on p. 5)

[99] National Research Council, *Getting up to Speed: The Future of Supercomputing*. Washington, DC: The National Academies Press, Nov. 2004. https://doi.org/10.17226/11148 (cited on p. 1)

[100] C. C. Paige, "A useful form of unitary matrix obtained from any sequence of unit 2-norm n-vectors," *SIAM J. Matrix Anal. Appl.*, vol. 31, no. 2, pp. 565–583, May 2009. https://doi.org/10.1137/080725167 (cited on pp. 95, 99)

[101] C. C. Paige, M. Rozložník, and Z. Strakoš, "Modified Gram-Schmidt (MGS), least squares, and backward stability of MGS-GMRES," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 1, pp. 264–284, 2006. https://doi.org/10.1137/050630416 (cited on pp. 86, 95)

[102] C. C. Paige and Z. Strakoš, "Residual and backward error bounds in minimum residual Krylov subspace methods," *SIAM J. Sci. Comput.*, vol. 23, no. 6, pp. 1898–1923, Jun. 2001. https://doi.org/10.1137/S1064827500381239 (cited on pp. 84, 94, 95)

[103] V. Y. Pan, G. Qian, and X. Yan, "Random multipliers numerically stabilize Gaussian and block Gaussian elimination: Proofs and an extension to low-rank approximation," *Linear Algebra and its Applications*, vol. 481, pp. 202–234, Sep. 2015. https://doi.org/10.1016/j.laa.2015.04.021 (cited on p. 7)

[104] V. Y. Pan, G. Qian, and A.-L. Zheng, "Randomized preprocessing versus pivoting," *Linear Algebra and its Applications*, vol. 438, no. 4, pp. 1883–1899, Feb. 2013. https://doi.org/10.1016/j.laa.2011.02.052 (cited on p. 8)

[105] V. Y. Pan and L. Zhao, "Numerically safe Gaussian elimination with no pivoting," *Linear Algebra and its Applications*, vol. 527, pp. 349–383, Aug. 2017. https://doi.org/10.1016/j.laa.2017.04.007 (cited on pp. 7, 36, 75)

[106] D. S. Parker, "Explicit formulas for the results of Gaussian elimination," Computer Science Department, UCLA, Tech. Rep. CSD-950025, 1995. (cited on p. 33)

[107] ——, "Random butterfly transformations with applications in computational linear algebra," Computer Science Department, UCLA, Los Angeles, CA, USA, Tech. Rep. CSD-950023, Jul. 1995. (cited on pp. 7, 34, 36)

[108] ——, "A randomizing butterfly transformation useful in block matrix computations," Computer Science Department, UCLA, Los Angeles, CA, USA, Tech. Rep. CSD-950024, Jul. 1995. (cited on p. 7)

[109] D. S. Parker and D. Lê, "How to eliminate pivoting from Gaussian elimination — by randomizing instead," Computer Science Department, UCLA, Los Angeles, CA, USA, Tech. Rep. CSD-950022, Jul. 1995. (cited on p. 7)

[110] J. Peca-Medlin, "Numerical, spectral, and group properties of random butterfly matrices," Ph.D. dissertation, UC Irvine, 2021. (cited on p. 7)

[111] J. Peca-Medlin and T. Trogdon, "Growth factors of random butterfly matrices and the stability of avoiding pivoting," arXiv:2203.15921, Jan. 2023. https://doi.org/10.48550/arXiv.2203.15921 (cited on p. 7)

[112] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM Press, 2003. https://doi.org/10.1137/1.9780898718003 (cited on pp. x, 85, 94)

[113] J. Schur, "Über Potenzreihen, die im Innern des Einheitskreises Beschränkt sind," *Journal für die reine und angewandte Mathematik*, vol. 1917, no. 147, pp. 205–232, Jan. 1917. https://doi.org/10.1515/crll.1917.147.205 (cited on pp. 22, 23, 60)

[114] Z. Shen, J. Zhang, and T. Suzuki, "Task-parallel tiled direct solver for dense symmetric indefinite systems," *Parallel Computing*, p. 102900, Feb. 2022. https://doi.org/10.1016/j.parco.2022.102900 (cited on p. 7)

[115] W. M. Sid-Lakhdar, S. Cayrols, D. Bielich *et al.*, "PAQR: Pivoting avoiding QR factorization," in *37th IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '23. St. Petersburg, FL, USA: IEEE, May 2023. (cited on p. 78)

[116] V. Simoncini and D. B. Szyld, "Theory of inexact Krylov subspace methods and applications to scientific computing," *SIAM Journal on Scientific Computing*, vol. 25, no. 2, pp. 454–477, Jan. 2003. https://doi.org/10.1137/S1064827502406415 (cited on p. 10)

[117] D. C. Sorensen, "Analysis of pairwise pivoting in Gaussian elimination," *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 274–278, Mar. 1985. https://doi.org/10.1109/TC.1985.1676570 (cited on p. 6)

[118] G. W. Stewart, "The QLP approximation to the singular value decomposition," *SIAM Journal on Scientific Computing*, vol. 20, no. 4, pp. 1336–1348, Jan. 1999. https://doi.org/10.1137/S1064827597319519 (cited on p. 78)

[119] ——, "Modifying pivot elements in Gaussian elimination," *Mathematics of Computation*, vol. 28, no. 126, pp. 537–542, 1974. https://doi.org/10.1090/S0025-5718-1974-0343559-8 (cited on p. 8)

[120] R. Strzodka and D. Göddeke, "Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, USA, 2006, pp. 259–270. https://doi.org/10.1109/FCCM.2006.57 (cited on p. 10)

[121] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer, 2010, pp. 157–173. https://doi.org/10.1007/978-3-642-11261-4_11 (cited on p. 26)

[122] J. Todd, *Basic Numerical Mathematics*. Basel: Birkhäuser, 1977. https://doi.org/10.1007/978-3-0348-7286-7 (cited on p. 71)

[123] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5, pp. 232–240, Jun. 2010. https://doi.org/10.1016/j.parco.2009.12.005 (cited on p. 7)

[124] L. N. Trefethen, "Three mysteries of Gaussian elimination," *ACM SIGNUM Newsletter*, vol. 20, no. 4, pp. 2–5, Oct. 1985. https://doi.org/10.1145/1057954.1057955 (cited on p. 21)

[125] L. N. Trefethen and R. S. Schreiber, "Average-case stability of Gaussian elimination," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 335–360, Jul. 1990. https://doi.org/10.1137/0611023 (cited on pp. 4, 5, 6)

[126] T. Trogdon, "On spectral and numerical properties of random butterfly matrices," *Applied Mathematics Letters*, vol. 95, pp. 48–58, 2019. https://doi.org/10.1016/j.aml.2019.03.024 (cited on p. 7)

[127] A. M. Turing, "Rounding-off errors in matrix processes," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 1, no. 1, pp. 287–308, Jan. 1948. https://doi.org/10.1093/qjmam/1.1.287 (cited on p. 33)

[128] K. Turner and H. F. Walker, "Efficient high accuracy solutions with GMRES(m)," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 3, pp. 815–825, May 1992. https://doi.org/10.1137/0913048 (cited on p. 9)

[129] J. van den Eshof and G. L. G. Sleijpen, "Inexact Krylov subspace methods for linear systems," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 1, pp. 125–153, Jan. 2004. https://doi.org/10.1137/S0895479802403459 (cited on p. 10)

[130] H. A. Van der Vorst and C. Vuik, "The superlinear convergence behaviour of GMRES," *Journal of Computational and Applied Mathematics*, vol. 48, no. 3, pp. 327–341, 1993. https://doi.org/10.1016/0377-0427(93)90028-A (cited on p. 94)

[131] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992. https://doi.org/10.1137/1.9781611970999 (cited on p. 36)

[132] H. Wang, S. Potluri, D. Bureddy *et al.*, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct. 2014. https://doi.org/10.1109/TPDS.2013.222 (cited on p. 13)

[133] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Princeton, NJ, USA: Prentice-Hall, 1963. (cited on pp. 4, 8, 58, 60, 62)

[134] ——, *The Algebraic Eigenvalue Problem*. London, UK: Oxford University Press, 1965. (cited on p. 20)

[135] M. A. Woodbury, *Inverting Modified Matrices*, ser. Memorandum Report. Princeton, NJ: Statistical Research Group, 1950, vol. 42. (cited on p. 49)

[136] I. Yamazaki, C. Glusa, J. Loe *et al.*, "High-performance GMRES multi-precision benchmark: Design, performance, and challenges," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Nov. 2022, pp. 112–122. https://doi.org/10.1109/PMBS56514.2022.00015 (cited on pp. 9, 115)

[137] E. L. Yip, "A note on the stability of solving a rank-p modification of a linear system by the Sherman-Morrison-Woodbury formula," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 2, pp. 507–513, Apr. 1986. https://doi.org/10.1137/0907034 (cited on p. 55)

[138] F. Zhang, T. Ando, C. Brezinski *et al.*, *The Schur Complement and Its Applications*, ser. Numerical Methods and Algorithms. New York, NY: Springer Science & Business Media, 2005. https://doi.org/10.1007/b105056 (cited on p. 60)

[139] Y. Zhao, T. Fukaya, L. Zhang, and T. Iwashita, "Numerical investigation into the mixed precision GMRES(m) method using FP64 and FP32," *Journal of Information Processing*, vol. 30, pp. 525–537, 2022. https://doi.org/10.2197/ipsjjip.30.525 (cited on pp. 9, 115)

[140] H. Zheng and J. Li, "A practical solution for KKT systems," *Numerical Algorithms*, vol. 46, no. 2, pp. 105–119, Oct. 2007. https://doi.org/10.1007/s11075-007-9129-8 (cited on p. 8)

[141] G. Zielke, "Testmatrizen mit maximaler Konditionszahl," *Computing*, vol. 13, no. 1, pp. 33–54, Mar. 1974. https://doi.org/10.1007/BF02268390 (cited on p. 68)

Index

Vita

Neil Lindquist was born in southern Minnesota on September 3rd, 1996. In 2019, he graduated from Saint John's University with a B.A. in Computer Science and Mathematics. Neil then enrolled as a doctoral student at the University of Tennessee in Knoxville. There, he worked in the Innovative Computing Laboratory as a graduate research assistant under the guidance of Jack Dongarra and Piotr Luszczek. In December 2022, Neil earned his M.S. degree in Computer Science. He expects to earn his Ph.D. degree in August 2023. Neil's research interests include numerical linear algebra, approximate computing algorithms, and high-performance computing.