Responsible Recklessness: Why High Performance Computing Must Embrace Randomized Numerical Linear Algebra

A Dissertation Presented for

The Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Maksim Melnichenko

December 2025

© by Maksim Melnichenko, 2025
All Rights Reserved.

Acknowledgements

I owe this dissertation, and any academic success I've had, to many colleagues, mentors, friends, and loved ones.

I would like to begin by expressing my deepest gratitude to my academic advisor, Dr. Jack Dongarra. Despite his immense academic height and extraordinary reputation in our field, both of which come at a cost of being constantly overbooked with responsibilities, Dr. Dongarra devoted much attention to the works of his students, including me. I am humbled by his combination of professionalism, as well as managerial and interpersonal skills - the very qualities that made the Innovative Computing Laboratory possible.

Next, I wish to acknowledge my informal advisor, Dr. Piotr Luszczek. Through the years, Piotr patiently answered my countless questions, provided close-up guidance when I needed it, and steered me towards the promising research directions while still leaving me freedom to explore. Furthermore, I am grateful for the fact that Piotr and I had many informal conversations about all things in life, and I can only dream of achieving his level of both omniscience and conversational charisma.

I am deeply thankful to my dissertation committee: Dr. James Demmel, Dr. Michael W Mahoney, Dr. Harwig Anzt, and Dr. Michael Berry.

Dr. Demmel and Dr. Mahoney both played a great role in my dive into the field of Randomized Numerical Linear Algebra. I still remember how excited I felt when Dr. Dongarra first offered to introduce me to collaborators at UC Berkeley during the very first semester of my PhD. At that moment, I thought that if I'm worthy of a formal introduction to the well-known scientists, then I must somewhat deserve to be here, after all! In the years that followed, Dr. Demmel and Dr. Mahoney gave me some of the best summer research opportunities of my life. It is thanks to them that many of the people in the field of RandNLA associate my name with UC Berkeley.

I am equally grateful to Dr. Anzt and Dr. Berry for their constructive feedback on my doctoral proposal and early drafts of this dissertation. Their comments aided in strengthening the overall quality of my work and also helped me shape the main message behind it. I owe a special thanks to Dr. Anzt for sacrificing a day of his vacation to attend my dissertation defense.

A very special acknowledgment is owed to Dr. Riley Murray, who, in all fairness, deserves to be the seventh member of my doctoral committee. As I've been saying throughout graduate school, Riley to me is like Batman to Robin. From the second semester of my PhD onward, Riley and I went through countless "20-minute debugging adventures," and jumped through so many academic and professional hoops together that I could have written an entire dissertation on that subject alone. All in all, I can firmly admit that Riley's contribution to making me a scientist is immeasurable, and I will forever be grateful to him for being both my mentor and a great friend.

I must also thank Dr. William Killian, my undergraduate advisor. Killian is responsible for the fact that I began pursuing computer science instead of just applied mathematics in the first place, and, furthermore, he was the driving force behind my immersion into the world of HPC. It is due to his efforts that I decided to pursue a PhD back in the day. I often think of the fact that if I had never met Killian during the second year of my undergraduate studies, my life would have certainly gone in a completely different way.

To the entire Innovative Computing Laboratory: thank you for being such inspiring and supportive colleagues.

Many friends and colleagues shaped my academic and personal life throughout the past nine and a half years of my presence in the United States. Among my undergraduate friends, special thanks go to Steveo Forsythe and Joe Dunton. From my graduate years, I would like to acknowledge Heiko Weichelt, Parth Nobel, Oleg Balabanov, and Rob Webber.

Finally, I must thank those closest to me: my father, Iurii, my mother, Tatiana, my sister, Anna, my brother-in-law, Alex, and, of course, my wife, Tanya. Thank you so much for all the love and support you've given to me over the years; this accomplishment is as much yours as it is mine.

This work was partially funded by an NSF Collaborative Research Framework: Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC), a project of the International Computer Science Institute, the University of Tennessee's ICL, the University of California at Berkeley, and the University of Colorado at Denver (NSF Grant Nos. 2004235, 2004541, 2004763, 2004850, respectively).

This dissertation reuses material from my existing work, published in the *SIAM Journal on Matrix Analysis and Applications (SIMAX)* [93], as well as from a preprint of my paper, posted on *Cornell University arXiv* [94], with limited use of coauthor contributions.

Abstract

Randomized Numerical Linear Algebra (RandNLA) is a relatively young branch of the field of Numerical Linear Algebra (NLA). It leverages randomization as the main approach to develop algorithms for computing solutions to classical linear algebra problems with performance superior to the deterministic NLA schemes. Despite that, RandNLA methods have yet to gain broad adoption within mainstream NLA software libraries, and, by extension, within the High-Performance Computing (HPC) toolkit aimed at combating the challenges posed by the end of Moore's law. This dissertation argues that the HPC community is missing a powerful opportunity by not embracing RandNLA methods.

As part of this work, I propose and analyze *three* RandNLA methods for the *two* NLA problem classes: full matrix factorizations through QR decomposition with column pivoting (QRCP), and low-rank factorization via Singular Value Decomposition (SVD). The core contribution of the projects I present lies in my development of high-performance implementations for each method, integrated into the open-source RandLAPACK library.

First, I present a novel QRCP algorithm for tall matrices. I provide a comprehensive analysis of the method in exact arithmetic and give a broad characterization of its finite-precision arithmetic analysis. Experiments on the algorithm's implementation demonstrate order-of-magnitude speedups over LAPACK's standard function for QRCP.

Next, I present an algorithmic framework that enables the design of practical QRCP algorithms for general matrices through user-controlled choices of the core subroutines. I provide a comprehensive overview of how to navigate these choices on modern hardware platforms, offering detailed descriptions of alternative methods for both CPUs and GPUs. CPU experiments demonstrate that the proposed method achieves performance improvements of up to two orders of magnitude over LAPACK's

standard QRCP routine. GPU experiments suggest that my method attains approximately 65% of the performance of cuSOLVER's unpivoted QR factorization.

Finally, I discuss a high-performance implementation of a partial SVD algorithm based on the Krylov subspace method. The algorithm is tested against randomized SVD and a deterministic SVD based on the implicitly restarted Lanczos method. Results on a dense synthetic dataset and a real-world sparse model reduction problem show the superiority of the algorithm I implemented.

Keywords: Randomized Numerical Linear Algebra, RandNLA, High-Performance Computing, Partial SVD, Krylov Subspace Methods, QRCP, RandLAPACK, GPU.

Table of Contents

1 Introduction **1**

1.1 NLA problems under consideration . . . 2

1.1.1 QR factorization with column pivoting. . . . 2

1.1.2 Partial Singular Value Decomposition. . . . 6

1.2 Common definitions and notation . . . 8

1.3 RandNLA software . . . 9

1.3.1 Shared experimental methodology . . . 10

2 Cholesky QR with Randomization and Pivoting **12**

2.0.1 Where is the randomness? . . . 12

2.0.2 The CholeskyQR method and its variants . . . 13

2.0.3 Randomized preconditioning for CholeskyQR . . . 14

2.0.4 Chapter outline . . . 15

2.0.5 Chapter-specific definitions and notation . . . 16

2.0.6 Chapter-specific experiments setup . . . 18

2.1 Getting to know CQRRPT . . . 19

2.1.1 CQRRPT's deterministic core . . . 19

2.1.2 Probabilistic aspects of CQRRPT . . . 24

2.2 Numerical stability - condensed remarks . . . 26

2.2.1 Summary . . . 27

2.2.2 Determining numerical rank in practice . . . 28

2.3 Pivot quality experiments . . . 29

2.3.1 Example low-coherence matrices . . . 31

2.3.2 An example high-coherence matrix . . . 31

2.4 CQRRPT output quality . . . 33

2.5 Algorithm benchmarking . . . 36

2.5.1 Profiling and optimizing CQRRPT . . . 37
2.5.2 CQRRPT on extremely tall matrices . . . 41
2.6 Chapter conclusion . . . 41

3 Anatomy of High-Performance Column-Pivoted QR Decomposition 45
3.0.1 Chapter outline . . . 47
3.0.2 Chapter-specific definitions and notation . . . 48
3.0.3 Chapter-specific experiments setup . . . 49
3.1 The framework . . . 52
3.1.1 A practical wide QRCP selection . . . 54
3.1.2 Numerical rank selection . . . 55
3.1.3 Tall QR selection . . . 57
3.1.4 Selection of methods for applying the Q-factor . . . 61
3.1.5 Selection of implementation for column permutation . . . 62
3.2 Practical implementation of BQRRP and storage management . . . 65
3.2.1 Input and output specification . . . 65
3.2.2 Initial sketching details . . . 67
3.2.3 Block partitions in BQRRP . . . 67
3.2.4 Processing the sketch and column permutation . . . 68
3.2.5 Panel QR factorizations details . . . 69
3.2.6 Applying transposed Q and updating factors . . . 71
3.2.7 Algorithm termination and sample update . . . 71
3.3 Considerations specific to implementations targeting GPU accelerators 72
3.3.1 Limited LAPACK functionality . . . 72
3.3.2 Column permutation . . . 73
3.3.3 The view of a practical BQRRP_GPU . . . 74
3.4 Performance profiling of major computational kernels . . . 75
3.5 Pivot quality on the Kahan matrix . . . 77
3.6 BQRRP output quality . . . 79
3.7 Performance results . . . 82
3.7.1 CPU algorithms performance . . . 82
3.7.2 Performance of GPU Implementations . . . 86
3.8 Investigating HQRRP performance . . . 90
3.8.1 Alternative view of Fig. 1.1 . . . 90
3.8.2 Varying HQRRP block size . . . 92

3.8.3 HQRRP subroutines profiling . . . 94
3.9 CQRRPT revisited . . . 94
3.10 Chapter conclusion . . . 99

4 Accelerating the partial singular value decomposition 102
4.0.1 Chapter-specific definitions and notation . . . 102
4.0.2 Chapter-specific experiments setup . . . 103
4.1 Description of algorithms . . . 104
4.1.1 Alternative methods under evaluation . . . 107
4.1.2 Common tuning parameters and metrics . . . 109
4.2 Performance results . . . 110
4.2.1 Experiments on dense data . . . 110
4.2.2 Experiments on sparse data . . . 115
4.3 Chapter conclusion . . . 118

5 Epilogue 120
5.1 The persuasion . . . 121
5.2 The future . . . 122

Bibliography 123

A ABRIK additional content 136
A.1 Experiments on the remaining dense matrices . . . 136

Vita 141

List of Tables

2.1 Key details of the hardware and software configuration of the platform where CQRRPT testing was performed. 20
2.2 Reconstruction error and orthogonality loss in CQRRPT. 35
2.3 Canonical GFLOP/s scaling with the number of threads used for various tall QR and QRCP methods. 38

3.1 Key details of the hardware and software configuration of the platforms where BQRRP CPU testing was performed. 50
3.2 Key details of the hardware and software configuration of the platform where BQRRP GPU testing was performed. 50
3.3 Reconstruction error and orthogonality loss in GEQP3 and BQRRP. . 81

4.1 Key details of the hardware and software configuration of the platform where ABRIK testing was performed. 105

List of Figures

1.1 Speedup of HQRRP and standard LAPACK unpivoted QR, GEQRF, over the standard LAPACK QRCP, GEQP3. 5

2.1 Pivot quality of CQRRPT and GEQP3 for low-coherence matrices with two types of spectral decay. 32
2.2 Pivot quality of CQRRPT and GEQP3 for a high-coherence matrix under two choices of sketching distribution parameters. 34
2.3 Tall QR and QRCP performance comparisons. 38
2.4 Percentages of CQRRPT's runtime, occupied by its respective subroutines. 40
2.5 Effect of varying the embedding factor size on performance of CQRRPT. 40
2.6 Performance for HQRRP vs GEQP3 for varying numbers of OpenMP threads. 42
2.7 Performance of tall QR and QRCP methods on extremely tall matrices. 42
2.8 Runtime breakdown of CQRRPT on extremely tall matrices. 44

3.1 Performance of the candidate methods for qrcp_wide function. 56
3.2 Performance of the candidate methods for qr_tall function. 60
3.3 Performance of the candidate methods for apply_trans_q function. . . 63
3.4 A visualization of how all the major components of BQRRP are stored at an arbitrary iteration of the algorithm. 66
3.5 Percentages of BQRRP_CPU runtime, occupied by its respective subroutines. 76
3.6 Percentages of BQRRP_GPU runtime, occupied by its respective subroutines. 78
3.7 Spectrum of Kahan matrix. 80
3.8 Pivot quality results for BQRRP. 80

3.9 Performance of various general QR and QRCP methods. 83
3.10 Effects of varying the numbers of threads on the FLOP rates in general QR and QRCP methods. 85
3.11 Performance of various general QR and QRCP methods on small input matrices. 87
3.12 Performance of various general QR and QRCP methods on rectangular matrices. 88
3.13 Performance of standard QR and two versions of BQRRP_GPU. . . . 89
3.14 Flop rates of HQRRP, GEQRF, and GEQP3 on 10,000 × 10,000 inputs. 91
3.15 Performance comparison of HQRRP, GEQRF, and GEQP3 for varying HQRRP block sizes. 93
3.16 Percentages of HQRRP runtime, occupied by its respective subroutines. 95
3.17 Additional performance of various general QR and QRCP methods on small input matrices. 97
3.18 Performance of various tall QR and QRCP, captured on newest Intel hardware. 98
3.19 Runtime breakdown of CQRRPT+BQRRP, captured on the new Intel hardware. 101

4.1 Spectra of the six matrices from a dense synthetic set. 105
4.2 Speed and accuracy of partial SVD methods on a dense input with fast-decaying spectrum. 111
4.3 Performance and accuracy of partial SVD methods on a dense input with fast-decaying spectrum. 112
4.4 Breakdown of ABRIK runtime by subroutine for dense inputs. 114
4.5 Speed and accuracy of partial SVD methods on a smaller sparse input. 116
4.6 Speed and accuracy of partial SVD methods on a larger sparse input. 117
4.7 Breakdown of ABRIK runtime by subroutine on sparse inputs. 119

A.1 Speed and accuracy of partial SVD methods on matrix two from the dense synthetic set. 137
A.2 Speed and accuracy of partial SVD methods on matrix three from the dense synthetic set. 138
A.3 Speed and accuracy of partial SVD methods on matrix four from the dense synthetic set. 139

A.4 Speed and accuracy of partial SVD methods on matrix five from the dense synthetic set. 140

List of Algorithms

1 CholeskyQR with randomization and pivoting for tall matrices 20
2 Core computations in CQRRPT 32
3 Blocked QR with Randomization and Pivoting 53
4 A practical wide QRCP 59
5 Cholesky QR + dependencies in the context of implicit QR 59
6 Sequential approach to column permutation 66
7 Column-parallel approach to column permutation 76
8 Adaptive Block Randomized Incremental Krylov method 106
9 Add to QR decomposition 114

Chapter 1

Introduction

Numerical Linear Algebra (NLA) is concerned with operating on matrices and vectors with numerical entries. Its importance for science and engineering is reflected in the observation from [47], specifically, that three out of the ten most influential algorithms of the 20th century are rooted in NLA. Notably, NLA truly "comes to life" through its computational realizations and applications, particularly viewed through the lens of High-Performance Computing (HPC). Innovation in HPC often comes from unexpected places. While the mainstream trends have focused on ever-lower precision [46], specialized hardware [125] [2], and architectural tuning [16] [37] [38] to delay the impact of Moore's law breakdown [115], another path forward has emerged from within theoretical computer science: Randomized Numerical Linear Algebra (RandNLA).

RandNLA leverages randomization to develop high-performance algorithms for classical NLA problems, achieving significant speedups over traditional deterministic methods, simultaneously matching their robustness. Notably, the efficiency of RandNLA methods carries across various architectures without the requirement of careful tuning to specific machines, but solely due to clever refactoring of the existing mathematical primitives. Initially viewed as a theoretical curiosity, RandNLA has matured significantly over the past two decades, though much of the progress has come from small-scale, focused efforts. The remarkable practical potential of RandNLA begs the following question: why are these algorithms not yet widely incorporated into the standard HPC-related NLA software? Are software vendors and scientific libraries developers overlooking innovation that practically lies in their hands? This

dissertation aims to reinforce the idea that RandNLA should be taken seriously by the HPC community.

As part of my dissertation work, I propose *three* novel* randomized methods applicable to the *two* NLA problem classes. The first two methods solve the problem of QR decomposition with column pivoting (QRCP), delivering a full matrix factorization. The third method addresses the problem of partial Singular Value Decomposition (partial SVD), focusing on providing a low-rank factorization. A key aspect of each of the three projects is the practical, high-performance implementation of the proposed methods, which are made available through the open-source RandLAPACK library [95]. Chapters 2, 3, and 4 discuss the specifics of each of the proposed methods, and Chapter 5 provides broad concluding remarks on this entire work.

In the remainder of this section, I discuss the challenges associated with QR with column pivoting and the partial SVD, conduct a literature review, and provide the motivation for the three projects presented in the subsequent chapters. After that, I outline the notation used throughout the dissertation and provide a brief on the supporting software, as well as the conventions used across the conducted experiments.

1.1 NLA problems under consideration

1.1.1 QR factorization with column pivoting.

The QR factorization, considered one of "The Big Six Matrix Factorizations" [73], is fundamental to the field of numerical linear algebra. The roots of this method go back to the works of Gram [64] and Schmidt [111] in the 1800s and early 1900s. In the late 1950s, Householder introduced the *Householder transformation*† [74], which preceded the actual formulation of the QR method in the two independent works, [57] and [78] in the early 1960s. The popularization of this method in NLA can be attributed to Wilkinson [106], particularly due to the genesis of rigorous and foundational analysis in [121].

*The implementations, algorithm engineering, and communication work here are novel; two out of three of the algorithms (Chapters 3, 4), as mathematical concepts, are not novel.

†The term "Householder transformation" was gradually adopted in the field; the author of the original work did not practice self-naming.

The applications of QR factorization range from linear least squares problems [28] and block orthogonalization [113] to contemporary randomized low-rank approximation algorithms [120]. Beyond the basic QR decomposition, QR with column pivoting (QRCP), originally introduced in [31], has additional benefits for numerically challenging problems, as it can help with rank-revealing tasks. To describe this problem concretely, let $\mathbf{M}$ be a matrix of size $m \times n$ with $m \geq n$. QRCP is concerned with finding a column permutation matrix $\mathbf{\Pi}$ and a QR decomposition of the product $\mathbf{M\Pi}$, i.e.,

$$\mathbf{M\Pi} = \mathbf{QR}$$

such that the information on the leading and trailing singular values of $\mathbf{M}$ can be inferred from the spectra of leading and trailing blocks in 2×2 partitions of $\mathbf{R}$.

Remark 1. I use the term *spectrum* in reference to singular values, not eigenvalues. In this context, eigenvalues are of interest to me only insofar as they coincide with singular values for positive semidefinite matrices.

QRCP is considerably more expensive than unpivoted QR from a communication standpoint, even with straightforward pivoting strategies. For example, Householder QR with Businger and Golub's *max-norm pivoting* requires updating column norms of every partial decomposition of $\mathbf{M}$ as the algorithm progresses from left to right across the columns [31]. These column-norm updates entail matrix-vector (BLAS 2) operations, which are far less suitable for modern hardware than the matrix-matrix (BLAS 3) operations abundant in classic algorithms for unpivoted QR. The significance of this problem has been known for decades and has compounded over time. Consequently, the standard LAPACK function for QRCP, called `GEQP3` [6], remains very slow due to memory bandwidth constraints.

Existing work. Naturally, efforts to revise the standard approach to QRCP have emerged in the past, among which are the works of Bischof and Quintana-Ortí [26, 25], as well as the independent works of Martinsson [89] and Duersch and Gu [54], with subsequent extensions by Martinsson *et al.* [90] and also by Xiao, Gu, and Langou [122]. Of particular interest are randomized algorithms described in [54] and [90]. Both these papers, in addition to the derivation of the pseudocode schemes, present software with practical QRCP implementations.

The early randomized Householder methods. In [54], the authors show several benchmarks of prospective QRCP methods (written in Fortran 90), compared against both the standard pivoted and unpivoted QR algorithms, provided by the contemporary (though unspecified) version of Intel Math Kernel Library (MKL). The `RQRCP` method, described in the pseudocode [54, Algorithm 4] exhibits an order-of-magnitude speedup over the standard pivoted QR and achieves asymptotically up to 60% of the performance of the standard unpivoted QR, when applied to full-rank square matrices. Their work also presents a `TRQRCP` scheme [54, Algorithm 6], which proves faster than `RQRCP` in low-rank cases.

In the other work [90], the authors present a randomized algorithm called `HQRRP` that offers an order-of-magnitude speedup over `GEQP3` and that achieves up to 80% of the performance of the standard unpivoted QR, `GEQRF`. In the benchmarks shown therein, the standard pivoted and unpivoted QR functions were taken from LAPACK version 3.4.0, and the implementations were linked to BLAS from MKL version 11.2.3. The performance results were reported for HQRRP that was implemented with libflame version 11104 [127]. In addition to the libflame-based implementation, the authors presented an LAPACK-compatible version of HQRRP [91]. The emphasis on the practicality and portability of HQRRP can be seen throughout the [90] paper, suggesting that, at its time, this algorithm qualified for an ideal replacement of `GEQP3` in LAPACK.

Projects motivation. The development of these algorithms, however, was not the final chapter in the quest for a high-performance QRCP method. As with its predecessors, neither `HQRRP` nor any of the variants of `RQRCP` made it into LAPACK. Many years of hardware development have passed, and benchmarking on modern machines shows that the gap in performance between the standard pivoted and unpivoted QR factorization algorithms has grown from roughly $10\times$ to near $100\times$. Simultaneously, while the LAPACK-compatible version of HQRRP proved to be easily portable to modern libraries, its performance did not increase commensurately. See Fig. 1.1 for details.

With that, following the steps of my predecessors, I accepted the challenge of formulating a modern approach for QR with column pivoting that would match the contemporary performance of the algorithms using Level 3 BLAS functionality. Chapters 2 and 3 detail the two projects I have completed in pursuit of this goal.

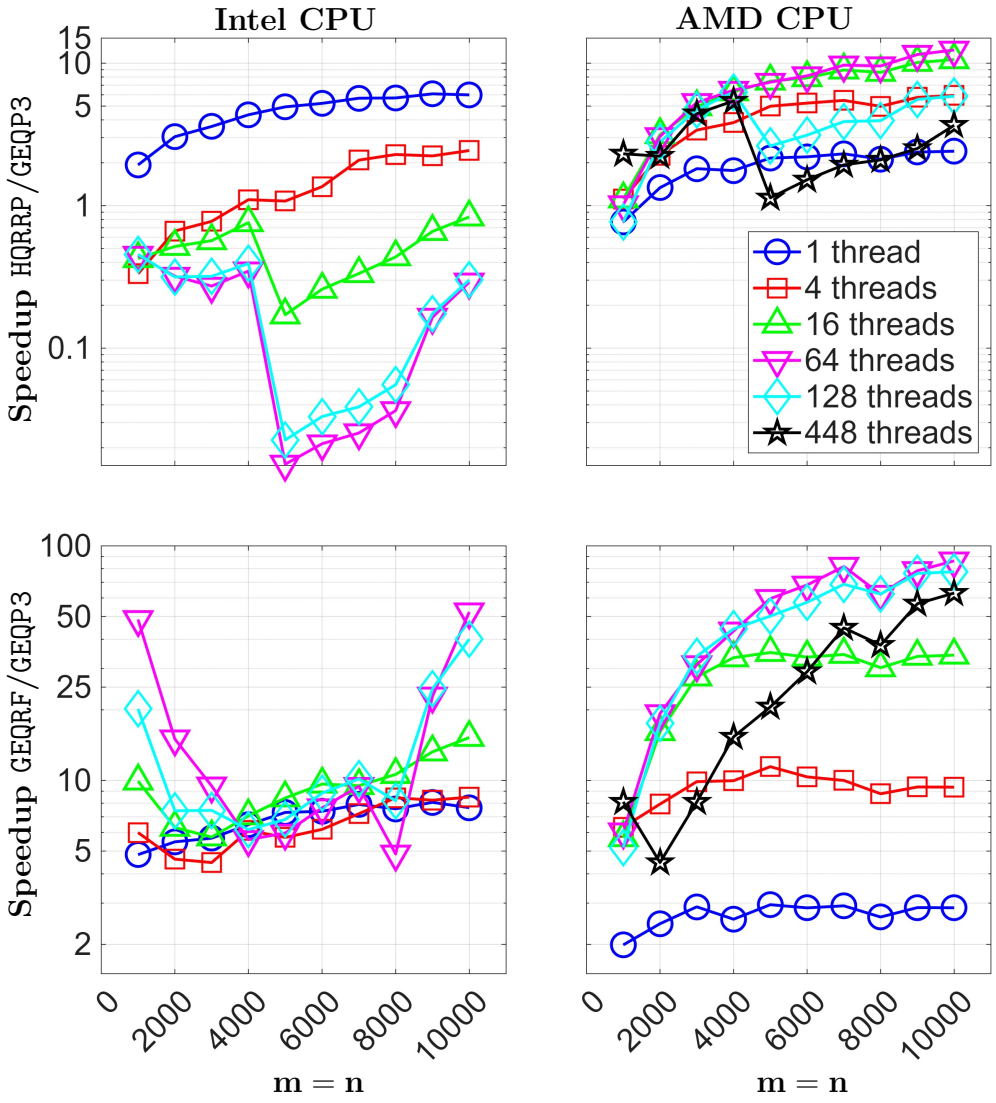


Figure 1.1: Speedup of `HQRRP` [90] (first row) and standard LAPACK unpivoted QR, `GEQRF` (second row), over the standard LAPACK QRCP, `GEQP3`, attained on matrices of sizes between 1,000-by-1,000 to 10,000-by-10,000, when varying the number of OpenMP threads used. In all experiments, `HQRRP` block size was set to 128, as suggested in [90, Sec. 4.1]. Results are captured on machines, described in Table 3.1. On an Intel system, the performance of `HQRRP` relative to that of `GEQP3` stagnates with the increase of the number of OpenMP threads used; on an AMD system, `HQRRP` is able to achieve performance above what was reported in [90, Fig. 1]. On both systems, the speedup of `GEQRF` over `GEQP3` greatly exceeds the speedup observed in [90, Fig. 5]. I further discuss the observed results in Section 3.8.

1.1.2 Partial Singular Value Decomposition.

The Singular Value decomposition (SVD) can rightfully be referred to as the most iconic tool in NLA [62] due to its generality, power, and breadth of applications. While the development of SVD traces back through a long history in mathematics, its first formal formulation was introduced by Eckart and Young in the 1930s in [55].

Over time, with the increase in NLA problem sizes, performing the full SVD became a computationally infeasible task in many practical cases. This has led to the development of partial SVD algorithms, used for extracting the dominant singular values and vectors of a given matrix.

Given a matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$, the partial SVD is a decomposition identifying the first k singular triplets $(\boldsymbol{u}_i, \boldsymbol{v}_i, \sigma_i)_{0\le i\le k-1}$, where the singular values are ordered $\sigma_0 \ge \sigma_1 \ge \cdots \ge \sigma_{k-1}$. Here, $\boldsymbol{u}_i \in \mathbb{R}^m$ is a left singular vector of $\mathbf{M}$, the associated right singular vector is $\boldsymbol{v}_i \in \mathbb{R}^n$, and the associated singular value is $\sigma_i \in \mathbb{R}_+$. By definition, the singular triplets satisfy

$$\sigma_i = \boldsymbol{u}_i^{\mathrm{T}}\mathbf{M}\boldsymbol{v}_i = \max \boldsymbol{u}_*^{\mathrm{T}}\mathbf{M}\boldsymbol{v}_*,$$

where the maximum is taken over unit-length vectors $\|\boldsymbol{u}_*\| = \|\boldsymbol{v}_*\| = 1$, which satisfy the orthogonality relationships

$$\boldsymbol{u}_* \perp \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{i-1} \quad \text{and} \quad \boldsymbol{v}_* \perp \boldsymbol{v}_0, \ldots, \boldsymbol{v}_{i-1}.$$

This means that the singular vectors are orthonormal, and the singular values are ordered from largest to smallest: $\sigma_0 \ge \sigma_1 \ge \cdots$.

The partial SVD leads to an optimal low-rank approximation of the matrix, namely

$$\hat{\mathbf{M}} = \sum\nolimits_{i=0}^{k-1} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^{\mathrm{T}}.$$

This low-rank approximation leads to accelerated linear algebra routines for matrix-vector products, norm calculations, and shifted linear solves. Moreover, $\hat{\mathbf{M}}$ serves as a reliable surrogate for $\mathbf{M}$ when the leading singular values $\sigma_0, \ldots, \sigma_{k-1}$ are much larger than all the rest.

The partial SVD identifies dominant structure in the matrix, in terms of the leading singular values and vectors. The left singular vectors reveal the dominant

patterns in the columns of the matrix, while the right singular vectors reveal the dominant patterns in the rows.

Existing work. LAPACK offers four functions that compute the full SVD: `GESVD` [63], `GESVJ` [53], `GESDD` [66], and `GESVDQ` [79]. These methods differ in how they compute an SVD, and as such, their performance and accuracy guarantees vary accordingly. As mentioned previously, the cost of computing a full SVD ramps up rapidly with the increase of the input matrix size, as it requires $\Theta(mn^2)$ operations.

Alternatively, various methods exist for computing a subset of the singular values and vectors of a matrix. These include both classical techniques [81] [82] [80] and more recent approaches. Among the modern methods, randomized singular value decomposition (RSVD) and randomized subspace iteration (RSI) stand out as the most prominent techniques for partial SVD ([48], [49], [70], [92]).

Despite their popularity, both RSVD and RSI are known to struggle with input matrices that have little variation in singular values (i.e., a slowly decaying spectrum). This particular structure can be seen in many practical datasets, such as SNP data containing individuals from genetically distinct subpopulations, leading to highly structured genetic variation. In such cases, an alternative, less common RandNLA method, Randomized Block Krylov Iteration (RBKI), can be employed. In [1], the authors compare RBKI and RSI, demonstrating that the former can be substantially more accurate in certain problems. A more recent manuscript, [120], provides theoretical analysis and numerical comparisons of RBKI, RSVD, and RSI, further advocating for the use of RBKI in suitable contexts. Additionally, the authors evaluate these methods in a practical context of genetic clustering, utilizing a real (though modest in size) dataset, sourced from [4].

Project motivation. While [120] presents a solid theoretical foundation and compelling practical illustrations, justifying the use of RBKI, a version of this method has yet to be implemented as part of high-performance linear algebraic software. In Chapter 4, I describe the process of development of such an implementation.

1.2 Common definitions and notation

As the three projects discussed in this dissertation differ significantly in nature, each employs specialized notation that is specific to its context. Such project-specific notation is introduced within the relevant subsections of the corresponding chapters: 2.0.5, 3.0.2, and 4.0.1. This section provides a brief overview of the common notation shared across all three projects.

Matrices appear in boldface sans-serif capital letters. The transpose of a matrix $\mathbf{X}$ is given by $\mathbf{X}^{\mathrm{T}}$, its ℓ_2 condition number is $\kappa(\mathbf{X})$, and its elementwise absolute value is $|\mathbf{X}|$. Numerical vectors appear as boldface lowercase letters, while index vectors appear as uppercase letters. The identity matrix is denoted by $\mathbf{I}$. I enumerate components of matrices and vectors with indices starting from zero, rather than starting from one.

The matrix that a given algorithm ultimately aims to decompose is denoted by $\mathbf{M}$ and has dimensions m-by-n, with m and n not related in any particular way unless otherwise specified.

I use the letters i, j, k, and ℓ as zero-based integer indices between 0 and $\min\{m, n\} - 1$. Their precise meaning is determined by context; note that I may redefine these symbols depending on the precise topic of discussion. In some contexts, i denotes the iteration of an algorithm loop.

I extract the leading k columns of $\mathbf{X}$ by writing $\mathbf{X}(:, 0{:}k)$; this range excludes column k. The trailing $n - k$ columns are extracted by writing $\mathbf{X}(:, k{:}n)$, this range includes column k and excludes column n. The $(i, j)^{\text{th}}$ entry of $\mathbf{X}$ is $\mathbf{X}(i, j)$. Similar conventions apply to extracting the rows of a matrix or components of a vector.

Remark 2. The choice of range tail-exclusive notation is primarily motivated by the desire to ensure consistency between the pseudocode descriptions of any given algorithm presented as part of my dissertation and its corresponding C++ implementation as part of the RandLAPACK library. As stated on the C++ algorithms library webpage, the standard C++ convention defines a given range as a half-open interval, that is, including the first element and excluding the last [33]. Furthermore, the range tail-exclusive notation is used in order to avoid having to deal with excessive presence of "-1" when specifying any given range that would add an excessive amount of visual clutter to the writing. The readers are to keep this fact in mind throughout my dissertation.

Remark 3. Chapters 2 and 3 use the special type of vector called the *permutation vector*, denoted $\boldsymbol{J}$. A permutation vector $\boldsymbol{J}$ of size n contains integer entries such that if $\boldsymbol{J}(j) = i + 1$, then the j^{th} column of $\mathbf{X}(:, \boldsymbol{J})$ was the i^{th} column of $\mathbf{X}$. Note that in LAPACK convention, the permutation vector $\boldsymbol{J}$ stores entries in a *one-based* index format used by Fortran and MATLAB (hence the presence of "+1" term in the $\boldsymbol{J}(j) = i + 1$ expression). In this dissertation, I use one-based indexing only when operating on the permutation vectors.

Throughout this dissertation, I aim to clearly identify the purpose of each function name, which are presented in `typewriter-style` font.

I often refer to linear algebraic functions using their conventional BLAS and LAPACK names in uppercase. The explanation of the standard naming conventions can be found in [29]. As explained further in Section 1.3.1, all benchmarks presented as part of my dissertation use real, double-precision arithmetic. As such, whenever I mention a given LAPACK function name, I avoid using the letter that denotes the precision, assuming that all computations are performed in double precision (for example, I use "`GEQP3`" instead of "`DGEQP3`").

1.3 RandNLA software

Every method conceived as part of my dissertation work has been implemented in the open-source RandLAPACK software [95]. RandLAPACK is a modern C++ library that offers a variety of mature RandNLA algorithms along with essential auxiliary methods. It is designed to serve as a platform for developing, testing, and benchmarking high-performance RandNLA algorithms for both CPU and GPU architectures. RandLAPACK software was produced as part of the BALLISTIC project [36] ; I serve as the primary contributor and maintainer of the RandLAPACK software.

RandLAPACK relies on RandBLAS [103] for obtaining the methods for generating and applying sketching operators, the tools used to reduce the dimensionality of matrices while approximately preserving key properties like distances and norms. These operations are central to all RandNLA methods. Furthermore, both RandBLAS and RandLAPACK rely on BLAS++ and LAPACK++ [61] for the purpose of obtaining basic linear algebraic functions; BLAS++ and LAPACK++ serve as wrappers around the low-level vendor-optimized BLAS and LAPACK packages. As such, many

vendor-optimized libraries (Intel's MKL [76], AMD's AOCL [5], Apple Accelerate [8], etc.) can be used to supply RandLAPACK and RandBLAS with basic linear algebra capabilities. RandBLAS relies on the `Random123` [109] library for exporting random number generators.

Remark 4. Note that, by contrast to BLAS++ and LAPACK++, RandBLAS and RandLAPACK currently only support real arithmetic types.

For a detailed description of RandBLAS, visit [102]. In contrast to RandBLAS, RandLAPACK's scope still continues to change and expand. Because of this, I have yet to define explicit web-based project documentation, though the library source is well-documented.

1.3.1 Shared experimental methodology

This section outlines the aspects of the experimental setup that are common to all three projects. For project-specific experimental details, refer to Sections 2.0.6, Section 3.0.3, 4.0.2.

Algorithm performance metric. Throughout this work, I measure the algorithm performance via the *canonical* FLOP rate, unless specified otherwise. This metric is obtained by dividing the FLOP count of a *standard* algorithm for a given matrix size by the wall clock time required to run a comparable algorithm under consideration. For example, for comparing canonical FLOP rates of various versions of QR and QRCP factorizations, I use the flop count of the standard LAPACK unpivoted QR called `GEQRF` [7, Page 121] and divide it by the wall clock time of any given QR or QRCP algorithm being compared. The canonical FLOP rate gives a clear way to compare algorithm performance by using a standard FLOP count divided by the algorithm's actual runtime. Unlike the raw FLOP rate, which gives credit to unnecessary work, the canonical FLOP rate sets a fair comparison by holding all algorithms to the same amount of work. While runtime alone could be used for comparisons, the FLOP rates are useful because they connect to familiar performance standards, such as the peak FLOP rates advertised for the hardware or standard algorithms like `GEMM` for matrix-matrix multiply. FLOP rates also help reveal scalability issues across problem sizes: if an algorithm's FLOP rate drops a lot as the problem size grows, it may reveal an inefficient method or implementation. Canonical

FLOP rates, therefore, give a balanced measure that considers both runtime and standard performance benchmarks.

Hardware and software configuration. All of my benchmarks use double-precision arithmetic, and all code was compiled with the `-O3` flag. I specify the exact compilers used in each project in the project-specific sections (Sections 2.0.6, Section 3.0.3, 4.0.2). The performance of each algorithm is determined by selecting its best execution time from five consecutive runs. Instead of running each algorithm individually five times, I execute the entire set of compared algorithms in sequence, repeating this set five times to capture comparatively consistent performance.

When running benchmarks on CPUs, I set the number of OpenMP threads (with `OMP_NUM_THREADS` environment variable) to the maximum value of threads available, unless specified otherwise. While using the maximum number of available threads may not always be the optimal approach to achieve peak performance, I advocate doing so for harnessing all available computational resources. All CPU systems used in this work featured dual-socket configurations. I launched tests with the `numactl --interleave=all` command when running the experiments to balance the memory usage across the two memory controllers.

To ensure reproducibility, I provide links to the specific algorithms, their benchmarks, and benchmark output files in the relevant project-specific sections (Sections 2.0.6, Section 3.0.3, 4.0.2).

Chapter 2

Cholesky QR with Randomization and Pivoting

In the first project as part of my dissertation work, I use methods common in Randomized Numerical Linear Algebra (RandNLA) to develop a fast and reliable QRCP algorithm, called `CQRRPT`*, which stands for "CholeskyQR with Randomization and Pivoting for Tall matrices." As shown below, `CQRRPT` can outperform not only other QRCP algorithms but also specialized communication-avoiding algorithms for unpivoted QR, such as shifted CholeskyQR3 [58] and TSQR [37]. The dominant cost of this algorithm is attributed exclusively to BLAS 3 operations and a random projection, rendering it highly efficient from a communication standpoint. On distributed memory systems, it can be implemented with just two sum-reduce exchanges. Compared to shifted CholeskyQR3, `CQRRPT` requires $\frac{4}{3}\times$ fewer data passes and $\frac{3}{2}\times$ fewer inter-processor messages. Furthermore, compared to TSQR, `CQRRPT` lends itself more easily to performance optimization on massively parallel architectures, thanks to its simple reduction operator.

2.0.1 Where is the randomness?

`CQRRPT` uses randomness only once: as its very first step, it samples a linear dimension reduction map from a prescribed distribution. The dimension reduction map is called a *sketching operator* and is said to be sampled from a *sketching distribution*. Its next step is to apply the sketching operator to the input matrix to produce

*pronounced "see-crypt"

a *sketch*. Computing a sketch can be as simple as selecting rows from a matrix and as complicated as fast Fourier transforms. My implementation of `CQRRPT` uses structured sparse sketching operators, since these can provide exceptional speed without sacrificing the reliability of the algorithm.

Analysis of sketching-based algorithms often comes in two parts. In the first part, the algorithm is analyzed as a purely deterministic method, conditional on an event that the sketch retains suitable geometric information from the input matrix. The second part uses tools from random matrix theory to bound the probability with which the desirable event happens, where the bound is *unconditional on numerical properties of the input matrix*. My analysis of `CQRRPT` follows this tradition. The presentation strongly emphasizes the first phase of analysis, since the probabilistic event that the first phase conditions on is extremely well understood.

I refer the reader to the recent monograph [101] (and the many references therein) for a broad overview of how randomness can be leveraged in numerical linear algebra.

2.0.2 The CholeskyQR method and its variants

The idea behind CholeskyQR is simple. Given the $m \times n$ matrix $\mathbf{M}$ with $m \geq n$, compute the Gram matrix $\mathbf{G} = \mathbf{M}^{\mathrm{T}}\mathbf{M}$. If one can factor $\mathbf{G} = \mathbf{R}^{\mathrm{T}}\mathbf{R}$ where $\mathbf{R}$ is an invertible upper-triangular matrix, then they can obtain the QR decomposition's orthogonal factor, hereafter, the "$\mathbf{Q}$-factor," as $\mathbf{Q} = \mathbf{M}\mathbf{R}^{-1}$. Note that this procedure only works if $\mathbf{M}$ has rank n.

Computational profile. Computing $\mathbf{G}$ and $\mathbf{R}$ in CholeskyQR can be carried out by the BLAS function `SYRK` and the LAPACK function `POTRF`, respectively. The $\mathbf{Q}$-factor can be formed explicitly with the Level 3 BLAS function `TRSM`. The *flop count* for CholeskyQR using these three functions is roughly $2mn^2 + n^3/3$ [7, Page 120], which is close to the $2mn^2 - (2/3)n^3$ flop count of Householder QR computed by LAPACK's `GEQRF` function [7, Page 122] when $m \gg n$.

Notably, CholeskyQR returns an explicit $\mathbf{Q}$-factor, while `GEQRF` outputs an implicit representation of its $\mathbf{Q}$-factor. The implicit representation is preferable for manipulating the full $m \times m$ orthogonal operator in a "non-economical" QR decomposition. Nevertheless, many common numerical methods, such as subspace projection methods or singular value decomposition of tall matrices, require computing the $\mathbf{Q}$-factor explicitly. Translating `GEQRF`'s implicit factor into an explicit

form (using LAPACK's `ORGQR`) takes an additional $2mn^2 - (2/3)n^3$ flops [7, Page 122]. Consequently, when $r := m/n > 1$, CholeskyQR requires a fraction $(6r+1)/(12r+4)$ of the flops of the traditional method of obtaining a fully explicit QR factorization.

Limitations. Despite its simplicity and speed, CholeskyQR is rarely used in practice. One clear shortcoming is that it fails if the computed Gram matrix is numerically rank-deficient, which can happen even if the input matrix is numerically full-rank. More generally, if $\kappa(\mathbf{M})$ denotes the condition number of $\mathbf{M}$ in the spectral norm and u denotes the unit roundoff in working precision, then rounding errors can lead to significant *orthogonality loss*, in the sense that $\|\mathbf{Q}^{\mathrm{T}}\mathbf{Q} - \mathbf{I}\|_2$ can be as large as $\mathcal{O}(\mathsf{u}\,\kappa(\mathbf{M})^2)$; by contrast, Householder QR remains robust even in singular cases, including inputs with zero rows and zero columns [37] [60] [100].

The orthogonality loss of CholeskyQR can be mitigated by a variety of methods.[†] For example, *Jacobi preconditioning* normalizes all columns of $\mathbf{M}$ to have unit norm and brings the condition number of $\mathbf{M}$ to within a factor $\sqrt{n}$ of the best-possible diagonal preconditioner [112, Theorem 3.5]; similar results are available for normalizing column blocks via a block-diagonal preconditioner [35]. Recently, it has been suggested to obtain a preconditioner from an LU decomposition of $\mathbf{M}$ [116] or from CholeskyQR on a regularized version of $\mathbf{M}$ [58]. Another approach to mitigating orthogonality loss involves reorthogonalization (essentially running CholeskyQR twice) resulting in "CholeskyQR2" [60, 123]. However, this method can still fail when $\kappa(\mathbf{M}) > \mathsf{u}^{-1/2}$. Finally, a mixed precision approach computes the Gram matrix in higher precision whereby the problematic conditioning can be better controlled [124].

2.0.3 Randomized preconditioning for CholeskyQR

The following algorithm serves as the starting point of this work. Given the matrix $\mathbf{M}$, it sketches $\mathbf{M}$ and then uses the triangular factor of the sketch's QR decomposition as a preconditioner in CholeskyQR. Like standard (unpreconditioned) CholeskyQR, it has a hard requirement that $\mathbf{M}$ is full rank in exact arithmetic.

1. Compute $\mathbf{M}^{\mathrm{sk}} = \mathbf{S}\mathbf{M}$ using a sketching operator $\mathbf{S} \in \mathbb{R}^{d \times m}$ with $n \leq d \leq m$.

[†]Indeed, this project adds `CQRRPT` to the list of orthogonality-loss mitigation methods, even though `CQRRPT` is concerned with pivoted rather than unpivoted QR.

2. Compute $[\mathbf{Q}^{\text{sk}}, \mathbf{R}^{\text{sk}}] = \texttt{qr}(\mathbf{M}^{\text{sk}})$ and discard $\mathbf{Q}^{\text{sk}}$.

3. Form $\mathbf{M}^{\text{pre}} = \mathbf{M}(\mathbf{R}^{\text{sk}})^{-1}$ explicitly, using back substitution.

4. Compute $\mathbf{G} = (\mathbf{M}^{\text{sk}})^{\text{T}}(\mathbf{M}^{\text{sk}})$ and $\mathbf{R}^{\text{pre}} = \texttt{chol}(\mathbf{G}, \text{“upper”})$.

5. Set $\mathbf{Q} = \mathbf{M}^{\text{pre}}(\mathbf{R}^{\text{pre}})^{-1}$ and $\mathbf{R} = \mathbf{R}^{\text{pre}}\mathbf{R}^{\text{sk}}$, then return $\mathbf{Q}$ and $\mathbf{R}$.

This method was first introduced [56] without the context of the broader RandNLA literature; it was first studied in detail with the RandNLA literature in mind in [11] and subsequently in [71]. Notably, the possibility of first computing an orthonormal $\mathbf{Q}$-factor $\mathbf{M}^{\text{pre}}$, and subsequently retrieving the ℓ_2-orthonormal $\mathbf{Q}$-factor via a CholeskyQR on $\mathbf{M}^{\text{pre}}$ was originally proposed elsewhere [13, 12]. Furthermore, the idea of using $\mathbf{R}^{\text{sk}}$ as the preconditioner traces back to the *sketch-and-precondition* paradigm for overdetermined least squares [108, 10, 97].[‡] If a well-studied condition called a *subspace embedding property* holds between $\mathbf{S}$ and range($\mathbf{M}$), then $\mathbf{M}^{\text{pre}}$ will be nearly-orthonormal [51, 110, 52, 88].

For any reasonable sketching operator, $\mathbf{S}$, the flop count of the algorithm outlined above will be $\mathcal{O}(mn^2)$. Whether or not practical speedup is observed depends greatly on the sketching distribution and on the method used to compute the product $\mathbf{SM}$. If a fast sketching operator is used with $n \leq d \leq m$, then the leading term in the resulting algorithm's flop count will be $3mn^2$, which is only mn^2 more than the standard CholeskyQR.

2.0.4 Chapter outline

`CQRRPT` takes the method from Section 2.0.3 and replaces the call to a QR function on $\mathbf{M}^{\text{sk}}$ with a call to a QRCP function on $\mathbf{M}^{\text{sk}}$. It uses the results from this call to QRCP to provide the pivot indices, an estimate of $\mathbf{M}$'s numerical rank, and the information for the preconditioner in the form of an upper-triangular matrix $\mathbf{R}^{\text{sk}}$. The numerical rank estimate is needed to deal with the possibility that $\mathbf{M}$ might be rank-deficient.

Section 2.1 formally introduces `CQRRPT` in Algorithm 1. This formalism emphasizes how the effect of randomness on `CQRRPT`'s behavior can be isolated in the choice of distribution used for the sketching operator. The core operations of `CQRRPT` are

[‡]In this context, $\mathbf{M}^{\text{pre}}$ was only formed implicitly and was accessed as a linear operator.

presented in Algorithm 2, which can be analyzed as a purely deterministic algorithm. Theorem 2.1 establishes the correctness of `CQRRPT`'s output, and Theorem 2.2 characterizes the spectrum of the preconditioned input matrix.

Section 2.2 concerns `CQRRPT`'s numerical stability, presenting a condensed view of [93, Section 3]. Its main result of this section states that `CQRRPT` can be implemented to provide numerical stability that is *unconditional* on numerical properties of the input matrix. Furthermore, this section explains how numerical rank selection can be performed in practice.

Section 2.3 empirically investigates pivot quality. It shows how easy-to-compute metrics of pivot quality compare when running the LAPACK default function (`GEQP3`) versus when running `CQRRPT` (based on applying `GEQP3` to $\mathbf{M}^{\text{sk}}$). The results show the interaction between *coherence*, which essentially represents a condition number for the act of sampling or embedding, and parameter choices for sparse sketching operators.

Section 2.4 empirically investigates the reconstruction error and orthogonality loss in `CQRRPT`, comparing the results to those from `GEQP3`. Note that this section was not originally included as part of [93].

Section 2.5 provides performance experiments with a RandLAPACK implementation of `CQRRPT`. The experiments use a machine with dual 24-core Xeon Gold 6248R CPUs, where Intel MKL and RandBLAS are the underlying linear algebra libraries. I first compare `CQRRPT` with alternative methods for both unpivoted and pivoted QR factorizations. `CQRRPT` handily beats competing methods for large matrices, and it has the advantage of a simpler representation of $\mathbf{Q}$. I then investigate how `CQRRPT`'s performance might be improved. Runtime profiling results show that computing QRCP of the sketch via `GEQP3` can exceed the cost of CholeskyQR several times over. This motivates an experiment where QRCP on the sketch is handled by HQRRP [90] instead of `GEQP3`.

Concluding remarks are given in Section 2.6.

2.0.5 Chapter-specific definitions and notation

Some necessary evils. Virtuous notation is readable and intuitive. Despite my best efforts, I have been unable to devise a wholly virtuous notation for this chapter. I have settled instead for notation that is readable and *internally consistent* for matrices with similar structures.

Leading columns of matrices subject to pivoting and QR. Suppose $\mathbf{X}$ is a matrix with n columns that is to be decomposed via QR after column pivoting by $\boldsymbol{J}$ (a permutation vector of $\{1, \ldots, n\}$). For each $k \in \{0, \ldots, n-1\}$, define the truncated, pivoted matrix

$$\mathbf{X}_k := \mathbf{X}(:, \boldsymbol{J}(0{:}k)).$$

Taking $k = n - 1$ allows us to refer to a pivoted matrix without any truncation.

Remark 5. Recall from Section 1.2 that pivot vectors in LAPACK store entries in a one-based index format used by Fortran and MATLAB.

I use this notation from the outset with the $m \times n$ matrix $\mathbf{X} = \mathbf{M}$ and the $d \times n$ matrix $\mathbf{X} = \mathbf{M}^{\text{sk}}$. Section 2.2 extends this convention to a matrix denoted by "$\mathbf{M}^{\text{pre}}$," which is decomposed via *unpivoted* QR.

Column-pivoted QR decompositions, and their factors. A QR decomposition consists of two conformable matrices: an orthonormal matrix called the $\mathbf{Q}$-*factor* and an upper-trapezoidal matrix called the $\mathbf{R}$-*factor*. I equip these factors with special indexing rules. Specifically, the first k columns of a $\mathbf{Q}$-factor "$\mathbf{Q}$" are denoted by $\mathbf{Q}_k$, and the first k rows of an $\mathbf{R}$-factor "$\mathbf{R}$" are denoted by $\mathbf{R}_k$.

This chapter features three matrices with important QR factors: a $d \times n$ pivoted matrix $\mathbf{M}_n^{\text{sk}}$, a matrix $\mathbf{M}^{\text{pre}}$ of dimensions $m \times k$ ($k \leq n$), and an $m \times n$ pivoted matrix $\mathbf{M}_n$. The QR factors of these matrices are distinguished from one another with superscripts, or the absence thereof (e.g., $\mathbf{R}^{\text{sk}}$, $\mathbf{R}^{\text{pre}}$, and $\mathbf{R}$).

Definition 2.1. *Let* $\mathbf{X}$ *denote a rank-*k *matrix with* n *columns. A* column-pivoted QR decomposition *of* $\mathbf{X}$ *consists of QR factors* $(\mathbf{Q}, \mathbf{R})$ *and a length-*n *permutation vector* $\boldsymbol{J}$ *such that the pivoted matrix* $\mathbf{X}_n = \mathbf{X}(:, \boldsymbol{J})$ *satisfies* $\mathbf{X}_n = \mathbf{Q}_k \mathbf{R}_k$.

Partitions of R-factors. The $\mathbf{R}$-factors are partitioned with notation that is similar to the celebrated paper of Gu and Eisenstat [65]. Specifically, to any $k \times n$ upper-trapezoidal matrix $\mathbf{R}$ and any $\ell \leq k$, I associate the distinguished submatrices

$$\mathbf{A}_\ell = \mathbf{R}(0{:}\ell,\ 0{:}\ell), \quad \mathbf{B}_\ell = \mathbf{R}(0{:}\ell,\ \ell{:}n), \quad \text{and} \quad \mathbf{C}_\ell = \mathbf{R}(\ell{:}k,\ \ell{:}n).$$

For fixed ℓ, these matrices let us cleanly express $\mathbf{R}$ as a 2×2 block matrix

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_\ell & \mathbf{B}_\ell \\ & \mathbf{C}_\ell \end{bmatrix}.$$

This notation helps in analyzing my algorithm's rank-revealing properties and numerical stability. I use it with $\mathbf{R}$-factors from the QR decompositions of $\mathbf{M}_n^{\text{sk}}$, $\mathbf{M}^{\text{pre}}$, and $\mathbf{M}_n$.

2.0.6 Chapter-specific experiments setup

For the experimental conventions that are general across all chapters of this dissertation, refer to Section 1.3.1.

Implementation resources. The version of the RandLAPACK software that was used in all experiments in this chapter is documented in [18]. The implementation of `CQRRPT` can be found in `/RandLAPACK/drivers/rl_cqrrpt.hh`. The code for constructing and dispatching the `CQRRPT` experiments can be found in `/benchmark/bench_CQRRPT/*`.

Hardware and software configuration. The detailed configurations of the machine that I used for conducting the experiments in this chapter are shown in Table 2.1.

Numerical properties of test matrices. Although `CQRRPT` is robust to the low-rank inputs (as explained in Section 2.2), this chapter is not concerned with studying its practical behavior on such matrices. As such, I use only full-rank input matrices in any of `CQRRPT`'s benchmarks.

All test matrices in this chapter were generated using RandBLAS. For the performance experiments, each matrix entry is independently sampled from the standard normal distribution. It should be noted that such matrices are appropriate for the performance tests as being representative of "typical" user inputs, and they have predictable numerical and performance properties, and give a representative behavior in pivot column interchanges.

I use numerically challenging matrices solely when discussing the quality of algorithms' output, as discussed in Sections 2.3 and 2.4.

2.1 Getting to know CQRRPT

Section 2.0.1 mentioned a two-phase approach to the analysis of randomized algorithms. In this section, I describe CQRRPT in a way that emphasizes this approach. I do this by differentiating between the full algorithm (which includes a random embedding step) and the algorithm's core (which is purely deterministic).

The full algorithm appears below. Its speed and reliability are affected decisively by the sketching distribution, and it has two parameters that control this distribution. The higher-level parameter is a *distribution family*; this is an association of matrix dimensions to probability distributions over matrices with those dimensions. The lower-level parameter, called the *embedding factor*, sets the size of the sketch in proportion to n.

CQRRPT's core appears in Algorithm 2. I characterize its behavior in Section 2.1.1 using the concept of *restricted singular values*. The restricted singular values of a $d \times m$ matrix $\mathbf{S}$ on a subspace $L \subset \mathbb{R}^m$ are the singular values of $\mathbf{SU}$ where $\mathbf{U}$ is any orthonormal matrix with range L.[§] The *restricted condition number* of $\mathbf{S}$ on L, denoted $\kappa(\mathbf{S}|L)$, is the ratio of its largest to smallest restricted singular values.

Section 2.1.2 explains how the distribution family and the embedding factor affect the probabilistic behavior of Algorithm 1. By the end of this section, it will be clear that standard results from random matrix theory can be used to probabilistically yet rigorously bound the restricted singular values of $\mathbf{S}$ on range($\mathbf{M}$). This will show that there are many ways of using CQRRPT to achieve different trade-offs between speed and reliability.

Remark 6. The analysis in this section is conducted in exact arithmetic. Such analysis may seem strange in the context of a CholeskyQR algorithm, but it plays a valuable role in plotting the course for the finite-precision analysis.

2.1.1 CQRRPT's deterministic core

The algorithm below relies on functions called "`qrcp`" and "`rank`." For now, the only required assumption is that `qrcp` always returns column-pivoted QR decompositions in the sense of Definition 2.1. The details of `rank` become important when considering finite-precision computations in Section 2.2. For the purposes of this section, this function is just a black-box that computes the exact rank of its input.

[§]Note that these singular values do not depend on the choice of orthonormal basis.

Table 2.1: Key configurations on hardware where testing was performed.

		Intel Xeon Gold 6248R (2x)
Cores per socket		24
Clock Speed	**Base**	3.00 GHz
	Boost	4.00 GHz
Cache sizes per socket	**L1**	1536 KB
	L2	24 MB
	L3	35.75 MB
RAM	**Type**	DDR4-2933
	Size	192 GB
BLAS & LAPACK		MKL 2023.2
GFLOPS in GEMM		1570

Algorithm 1 : CholeskyQR with randomization and pivoting for tall matrices

Required inputs. An $m \times n$ matrix $\mathbf{M}$.

Optional inputs. A distribution family $\mathscr{F}$ and an embedding factor $\gamma \geq 1$.

Outputs. QR factors of dimensions $m \times k$ and $k \times n$, and a length-n permutation vector. These comprise a column-pivoted QR decomposition of $\mathbf{M}$ in the sense of Definition 2.1 if and only if $\text{rank}(\mathbf{SM}) = \text{rank}(\mathbf{M})$; see Theorem 2.1.

1: **function** CQRRPT($\mathbf{M}, \gamma, \mathscr{F}$)
2: If $\mathscr{F}$ is not provided, set it to a default family of sparse sketching distributions.
3: If γ is not provided, set $\gamma = 1.25$.
4: Set $d = \lceil \gamma n \rceil$, and randomly sample a $d \times m$ matrix $\mathbf{S}$ from $\mathscr{F}_{d,m}$.
5: Compute $[\mathbf{Q}, \mathbf{R}, \boldsymbol{J}] =$ `cqrrpt_core`$(\mathbf{M}, \mathbf{S})$
6: **return** $\mathbf{Q}, \mathbf{R}, \boldsymbol{J}$

Remark 7. [93, Section 2.2 and Appendix A.2] explain how `CQRRPT` inherits the rank-revealing properties from `qrcp`.

There are two pressing questions for Algorithm 2.

1. Under what conditions does it actually obtain a decomposition of $\mathbf{M}$?
2. Just how "safe" is its use of CholeskyQR on Lines 6 through 8?

Section 2.2 discusses the answers to these questions in finite-precision arithmetic. Here, I present simpler answers under the assumption of computation in exact arithmetic. When interpreting them, it can be informative to use the fact that $\kappa(\mathbf{S}|\operatorname{range}(\mathbf{M}))$ is finite if and only if $\operatorname{rank}(\mathbf{SM}) = \operatorname{rank}(\mathbf{M})$.

Remark 8. Note that the assumption of $\operatorname{rank}(\mathbf{SM}) = \operatorname{rank}(\mathbf{M})$ used in Algorithm 1 follows trivially from Definition 2.2.

Theorem 2.1. *If* $\kappa(\mathbf{S}|\operatorname{range}(\mathbf{M}))$ *is finite, then* $[\mathbf{Q}_k, \mathbf{R}_k, \boldsymbol{J}] = \texttt{cqrrpt_core}(\mathbf{M}, \mathbf{S})$ *defines a column-pivoted QR decomposition of* $\mathbf{M}$ *in the sense of Definition 2.1.*

Proof. It is clear that $\mathbf{Q}_k$ is orthonormal and $\mathbf{R}_k$ is upper-trapezoidal. I need to show that if $\operatorname{rank}(\mathbf{SM}) = \operatorname{rank}(\mathbf{M})$ then $\boldsymbol{\Delta} = \mathbf{M}_n - \mathbf{Q}_k\mathbf{R}_k$ is zero. As a step towards this, note the identity $\mathbf{Q}_k\mathbf{R}_k = \mathbf{M}^{\text{pre}}\mathbf{R}_k^{\text{sk}}$, which implies $\operatorname{range}(\boldsymbol{\Delta}) \subset \operatorname{range}(\mathbf{M})$. Next, use the assumption that `qrcp` produces column-pivoted QR decompositions in sense of Definition 2.1 to find

$$\begin{aligned}\mathbf{S}\boldsymbol{\Delta} &= \mathbf{SM}_n - \mathbf{SM}^{\text{pre}}\mathbf{R}_k^{\text{sk}} \\ &= \mathbf{M}_n^{\text{sk}} - \mathbf{Q}_k^{\text{sk}}\mathbf{R}_k^{\text{sk}} = \mathbf{0}.\end{aligned}$$

Since $\operatorname{rank}(\mathbf{SM}) = \operatorname{rank}(\mathbf{M})$ implies $\ker(\mathbf{S}) \cap \operatorname{range}(\mathbf{M})$ is trivial, then $\boldsymbol{\Delta} = \mathbf{0}$. □

Theorem 2.2. *Let* $\mathbf{M}^{\text{pre}}$ *be as on Line 5 of* Algorithm 2 *for inputs* $\mathbf{M}$ *and* $\mathbf{S}$*. If* $\kappa(\mathbf{S}|\operatorname{range}(\mathbf{M}))$ *is finite, then the singular values of* $\mathbf{M}^{\text{pre}}$ *are the inverses of the restricted singular values of* $\mathbf{S}$ *on* $\operatorname{range}(\mathbf{M})$*.*

Proof. Let $\mathbf{U}$ be an $m \times k$ orthonormal matrix with the same range as $\mathbf{M}$. Define the set $\mathcal{U}_i = \{L \subset \mathbb{R}^k : L \text{ is a linear subspace and } \dim(L) = i\}$.

Since the singular values of a matrix are the square roots of the eigenvalues of its Gram matrix, and since the square root is monotonic, the classic min-max principle

for eigenvalues of Hermitian matrices directly translates to singular values of general matrices. This formulation of the min-max principle signifies that

$$\sigma_i\left(\mathbf{M}^{\text{pre}}\right) = \min_{X \in \mathcal{U}_i} \max_{\boldsymbol{x} \in X} \frac{\|\mathbf{M}^{\text{pre}}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} \quad \text{and} \quad \sigma_{k-i}\left(\mathbf{SU}\right) = \max_{Y \in \mathcal{U}_i} \min_{\boldsymbol{y} \in Y} \frac{\|\mathbf{SU}\boldsymbol{y}\|_2}{\|\boldsymbol{y}\|_2}.$$

The goal is to show that $\sigma_i\left(\mathbf{M}^{\text{pre}}\right) = \left(\sigma_{k-i}\left(\mathbf{SU}\right)\right)^{-1}$. To do this, assume the restricted singular values of $\mathbf{S}$ on range($\mathbf{M}$) are all nonzero. This ensures that range($\mathbf{M}^{\text{pre}}$) = range($\mathbf{M}$) and subsequently that there is an invertible matrix $\mathbf{T}$ where $\mathbf{M}^{\text{pre}} = \mathbf{UT}$. I also rely on the fact that the $d \times k$ matrix $\mathbf{SM}^{\text{pre}}$ is orthonormal. Combining these observations gives a chain of identities

$$\sigma_i\left(\mathbf{M}^{\text{pre}}\right) = \min_{X \in \mathcal{U}_i} \max_{\boldsymbol{x} \in X} \frac{\|\mathbf{M}^{\text{pre}}\boldsymbol{x}\|_2}{\|\boldsymbol{x}\|_2} = \min_{X \in \mathcal{U}_i} \max_{\boldsymbol{x} \in X} \frac{\|\mathbf{M}^{\text{pre}}\boldsymbol{x}\|_2}{\|\mathbf{SM}^{\text{pre}}\boldsymbol{x}\|_2} = \min_{X \in \mathcal{U}_i} \max_{\boldsymbol{x} \in X} \frac{\|\mathbf{UT}\boldsymbol{x}\|_2}{\|\mathbf{SUT}\boldsymbol{x}\|_2}.$$

Then, I apply a change of variables $\boldsymbol{y} = \mathbf{T}\boldsymbol{x}$ to get

$$\sigma_i\left(\mathbf{M}^{\text{pre}}\right) = \min_{X \in \mathcal{U}_i} \max_{\boldsymbol{x} \in X} \frac{\|\mathbf{T}\boldsymbol{x}\|_2}{\|\mathbf{SUT}\boldsymbol{x}\|_2} = \min_{Y \in \mathcal{U}_i} \max_{\boldsymbol{y} \in Y} \frac{\|\boldsymbol{y}\|_2}{\|\mathbf{SU}\boldsymbol{y}\|_2} = \left(\max_{Y \in \mathcal{U}_i} \min_{\boldsymbol{y} \in Y} \frac{\|\mathbf{SU}\boldsymbol{y}\|_2}{\|\boldsymbol{y}\|_2}\right)^{-1}$$

which completes the proof. □

Arithmetic complexity. The arithmetic complexity of Algorithm 2 depends on how $\mathbf{M}^{\text{sk}}$ is computed and then decomposed. Since there are many practical ways to handle the first of these operations, I shall simply say its cost in flops is C_{sk}.

Remark 9. Recall from Section 2.0.3 that for any reasonable operator $\mathbf{S}$, C_{sk} would be negligeable.

Other computations in `CQRRPT` can be evaluated as follows:

1. Factor $\mathbf{M}^{\text{sk}}$ by QRCP. The cost of this step depends on the algorithm used for QRCP. By default, I use LAPACK's `GEQP3`. With LAPACK's `GEQP3`, this operation would cost $4dnk - 2k^2(d+n) + 4k^3/3$ flops [62, Algorithm 5.4.1], where $k = \text{rank}(\mathbf{M}^{\text{sk}})$ and d is the user-defined embedding dimension of a sketch.

2. Apply the preconditioner to $\mathbf{M}$, to get $\mathbf{M}^{\text{pre}}$. This step is done via `TRSM`, so the cost is mk^2 [7, Page 120].

3. Run CholeskyQR on $\mathbf{M}^{\text{pre}}$. This operation is composed of `SYRK`, `POTRF`, and `TRSM` functions. This operation would cost $mk(k+1)+k^3/3+k^2/2+k/6+mk^2$ flops [7, Page 120].

4. Combine the triangular factors from QRCP on $\mathbf{M}^{\text{sk}}$ and CholeskyQR on $\mathbf{M}^{\text{pre}}$. This cost is negligible; an $\mathcal{O}(n^3)$ `TRMM`.

This results in a cumulative flop count of:

$$2mk^2 + mk(k+1) + 4dnk - 2k^2(d+n) + 5k^3/3 + k^2/2 + k/6 + C_{\text{sk}}. \tag{2.1}$$

If I plug $k = n$ into (2.1) and assume $d, n \in o(m)$, then the leading-order term in `CQRRPT`'s flop count is $3mn^2 + C_{\text{sk}}$. This compares favorably to the $4mn^2$ flops required to compute a Householder QR decomposition and then explicitly restore the $\mathbf{Q}$-factor with LAPACK's `ORGQR`. What's more, in certain applications it can suffice to represent $\mathbf{Q}$ as a composition of two elementary operators, $\mathbf{M}^{\text{pre}}$ and $(\mathbf{R}^{\text{pre}})^{-1}$. This comes with no sacrifices to the numerical stability, yet it decreases the leading term in `CQRRPT`'s arithmetic complexity to $2mn^2 + C_{\text{sk}}$.

Communication cost. `CQRRPT` lends itself well to distributed computation. Here it is prudent to choose the sketching distribution that allows for evaluation of Line 2 with the smallest possible value of d while retaining good statistical properties. This can be achieved in theory and practice with Gaussian sketching (see Section 2.1.2) and $\gamma = d/n \in [1.25, 5]$; see [101, §A.1.1].

Consider a popular setting where $\mathbf{M}$ is distributed block row-wise across $p \leq m/d$ processors, each of which has at least $2dn + n^2$ words of memory. Here, one implements Line 2 with local multiplications of blocks of $\mathbf{M}$ with the corresponding blocks of columns of $\mathbf{S}$ (using dn words of memory) and summing the contributions with an all-reduce operation (using another dn words of memory). From there, each processor performs QRCP on $\mathbf{M}^{\text{sk}}$, recovers $\mathbf{A}_k^{\text{sk}}$ from $\mathbf{R}^{\text{sk}}$, and applies its inverse to the corresponding local block of $\mathbf{M}$. When using the classical binomial tree version of all-reduce, the overall computation of $\mathbf{M}^{\text{pre}}$ requires one global synchronization, $2\log_2 p$ messages, $2dn\log_2 p$ communication volume, and two data passes. Adding the cost of the CholeskyQR step [104], the total cost of `CQRRPT` amounts to two global synchronizations, $4\log_2 p$ messages, $(2dn + n^2)\log_2 p$ communication volume,

and three data passes. Notably, if the recursive-halving version of all-reduce [117] was used, the communication volume can be improved to a mere $2dn + n^2$.

2.1.2 Probabilistic aspects of `CQRRPT`

From the perspective of Algorithm 1, the sketching operator $\mathbf{S}$ is sampled at random from a probability distribution, and so anything that $\mathbf{S}$ affects in Algorithm 2 becomes a random variable with some induced distribution. In particular, the probabilistic behavior of Algorithm 1 is determined by the induced distribution for the condition number of $\mathbf{M}^{\text{pre}}$. Therefore, to understand `CQRRPT`'s behavior, the following question must be explored.

> How can I bound the probability that $\kappa(\mathbf{M}^{\text{pre}})$ stays within a prescribed limit, *no matter the matrix* $\mathbf{M}$?

The answer to this question depends greatly on the distribution family $\mathscr{F}$ and the embedding factor γ used in Algorithm 1. Here, I give a range of answers based on different choices for these values.

Sketching distribution families: examples and intuition. For a fixed distribution family $\mathscr{F}$, use "$\mathscr{F}_{d,m}$" for the distribution in $\mathscr{F}$ over $d \times m$ matrices. Here are the structures of samples from $\mathscr{F}_{d,m}$ for three prominent distribution families.

- *Gaussian matrices.* The entries of $\mathbf{S}$ are iid Gaussian random variables with mean zero and variance $1/d$.
- *SASOs* (short-axis-sparse operators). The columns of $\mathbf{S}$ are independent. Each column has exactly s nonzeros (for a tuning parameter s) whose locations are chosen uniformly at random and whose values are chosen to be $\pm 1/\sqrt{s}$ with equal probability.[¶] In practice, it is common to keep ℓ between one and eight, even when d is on the order of tens of thousands.
- *SRHTs* (subsampled randomized Hadamard transforms). These are ordinarily only defined when $w = \log_2 m$ is an integer and are extended to general m by zero-padding the input. To absorb the zero-padding into the SRHT, define a $2^{\lceil w \rceil} \times m$ matrix $\mathbf{D}$ whose upper $m \times m$ block is a diagonal matrix populated by

[¶] For the etymology of these sketching operators, see [101].

independent Rademacher random variables and whose lower block is all zeros. An SRHT is then a composition of three operators: $\mathbf{S} = \sqrt{m/d} \cdot \mathbf{\Pi}[0{:}d, :]\mathbf{HD}$, where $\mathbf{H}$ is a Hadamard transform of order $2^{\lceil w \rceil}$ and $\mathbf{\Pi}$ is a random permutation matrix.

These families share key properties. First, if $\mathbf{S}$ is a random matrix sampled from $\mathscr{F}_{d,m}$, then its expected covariance matrix satisfies $\mathbb{E}[\mathbf{S}^{\mathrm{T}}\mathbf{S}] = \mathbf{I}_m$. Second, if $\mathbf{U}$ is an $m \times k$ orthonormal matrix and d/k is sufficiently large, then the sketched Gram matrix $\mathbf{U}^{\mathrm{T}}\mathbf{S}^{\mathrm{T}}\mathbf{S}\mathbf{U}$ concentrates strongly around $\mathbb{E}[\mathbf{U}^{\mathrm{T}}\mathbf{S}^{\mathrm{T}}\mathbf{S}\mathbf{U}] = \mathbf{I}_k$. To rephrase this second property: $\mathbf{SU}$ should concentrate strongly around the real Stiefel manifold $M_{d,k}$ of $d \times k$ orthonormal matrices.

To see the usefulness of these properties, consider Theorem 2.2. If range($\mathbf{U}$) = range($\mathbf{M}$) and $\mathbf{SU}$ has rank-k, then the singular values of $\mathbf{M}^{\mathrm{pre}}$ will equal those of the Moore-Penrose pseudo-inverse of $\mathbf{SU}$. Therefore the desirable property of $\mathbf{M}^{\mathrm{pre}}$ concentrating around $M_{m,k}$ is equivalent to $\mathbf{SU}$ concentrating around $M_{d,k}$.

Oblivious subspace embeddings. Let $\mathbf{U}$ be a matrix whose columns are an orthonormal basis for a linear subspace $L \subset \mathbb{R}^m$, and let $\mathbf{S}$ be a $d \times m$ matrix.

Definition 2.2. *I call* $\mathbf{S}$ *a* subspace embedding for L *with* distortion $\delta \in [0, 1]$ *if*

$$1 - \delta \leq \sigma_{\min}(\mathbf{SU})^2 \quad \textit{and} \quad \sigma_{\max}(\mathbf{SU})^2 \leq 1 + \delta. \tag{2.2}$$

Such a matrix is also called a δ-embedding *for* L.

Note that I always have $\kappa(\mathbf{S}|L) \leq \sqrt{(1+\delta)/(1-\delta)}$ when $\mathbf{S}$ is a δ-embedding for L. There is a long history of using subspace embeddings in randomized algorithms for least squares problems [110, 51, 52], including as a tool for finding preconditioners to solve least squares problems to higher accuracy [108, 10, 97].

Theory is available on how to choose d so that a sample from $\mathscr{F}_{d,m}$ will be a δ-embedding for any fixed linear subspace of a given dimension with high probability. One can use this theory to select d in practice for very well-behaved distribution families. For example, here is a representative result for Gaussians.

Remark 10. I state the following results with linear subspaces of dimension "n," since the dimension of range($\mathbf{M}$) is never larger than n.

Theorem 2.3. *Fix an n-dimensional linear subspace $L \subset \mathbb{R}^m$, along with some $\delta \in (0,1)$ and $\tau > 0$. If $\mathbf{S}$ is a $d \times m$ Gaussian operator with*

$$\frac{d}{n} \geq \left(\frac{1+\tau}{(1+\delta)^{1/2} - 1} \right)^2,$$

then it will be a δ-embedding for L with probability at least $1 - 2\exp(-n\tau^2/2)$.

The proof of Theorem 2.3 can be found in [93, Section 2.3.2]

I give results for SASOs and SRHTs below. These are often of interest since they are algorithmically more attractive than Gaussian matrices. However, the available bounds are quite pessimistic (they suggest that one take γ proportional to $\log n$, but taking $\gamma = 1.25$ suffices for practical purposes when n is large).

Theorem 2.4. *[32, Theorem 4.2] Fix an n-dimensional linear subspace $L \subset \mathbb{R}^m$, and any $B > 2$, $t \geq 1$, and $\delta < 1/2$. There are absolute constants c_1, c_2 where, upon taking*

$$\frac{d}{n} \geq c_1 t (B \log B) \frac{\log n}{\delta^2} \quad \text{and} \quad s \geq c_2 t (\log B) \frac{\log n}{\delta},$$

sampling a $d \times m$ SASO with s nonzeros per column provides a δ-embedding for L with probability at least $1 - B^{-t}$.

Theorem 2.5. *[14, Proposition 3.9] Let $L \subset \mathbb{R}^m$ be a linear subspace of dimension n. Let $\delta, p \in (0,1)$. If $\mathbf{S}$ is a $d \times m$ SRHT with $d \leq m$ and*

$$\frac{d}{n} \geq 2(\delta^2 - \delta^3/3)^{-1} \left(1 + \sqrt{\frac{8\log(6m/p)}{n}} \right)^2 \log(3n/p),$$

then it is a δ-embedding for L with probability at least $1 - p$.

Note that the analysis in [14, Proposition 3.9], cited above, builds on more fundamental results from [119, 30].

2.2 Numerical stability - condensed remarks

This section presents a condensed view of [93, Section 3], which analyzes `cqrrpt_core` (Algorithm 2) under the assumption that all algebraic operations in it and in its subroutines are performed in finite precision arithmetic. I refer the reader to [93, Section 3 & Appendix B.1] for the full elaboration on this topic.

Suppose for concreteness that

$$[\mathbf{Q}_k, \mathbf{R}_k, \boldsymbol{J}] = \texttt{cqrrpt_core}(\mathbf{M}, \mathbf{S}), \tag{2.3}$$

where $\mathbf{Q}_k$ is $m \times k$ and $\mathbf{R}_k$ is $k \times n$ and upper-triangular. In these terms, the analysis is concerned with bounding the *reconstruction error* $\|\mathbf{M}_n - \mathbf{Q}_k\mathbf{R}_k\|_{\mathrm{F}}/\|\mathbf{M}\|_{\mathrm{F}}$ and the *orthogonality loss* $\|\mathbf{Q}_k^{\mathrm{T}}\mathbf{Q}_k - \mathbf{I}_k\|_2$.

Proving bounds on these quantities requires assumptions on the `qrcp` and `rank` subroutines in Algorithm 2. So far, I have said very little about these points because they are ultimately design questions that have no single answer. As part of the finite-precision arithmetic analysis, the goal is to show that `qrcp` and `rank` can be implemented so that upon conditioning on $\mathbf{S}$ being a δ-embedding for range($\mathbf{M}$) (for some $\delta \leq 1/2$), the reconstruction error and orthogonality loss are bounded above by low-degree polynomials in (m, n). Such a result will directly imply that `CQRRPT`'s properties introduced in Section 2.1 are preserved under finite precision arithmetic.

New notation. Up until now, the $m \times k$ and $k \times k$ matrices $\mathbf{M}^{\mathrm{pre}}$ and $\mathbf{R}^{\mathrm{pre}}$ have been referred to without notational dependence on k. In this section, I use $\mathbf{M}_k^{\mathrm{pre}}$ and $\mathbf{R}_k^{\mathrm{pre}}$ for these matrices; their submatrices can be selected using notation consistent with Section 2.0.5. For example, I can speak of a parameter $\ell < k$ and use $\mathbf{M}_\ell^{\mathrm{pre}}$ in reference to the first ℓ columns of $\mathbf{M}_k^{\mathrm{pre}}$. Similarly, $\mathbf{A}_\ell^{\mathrm{pre}}$ can be used to denote the leading $\ell \times \ell$ submatrix of $\mathbf{R}_k^{\mathrm{pre}}$.

2.2.1 Summary

The challenge. I begin by emphasizing that since I want to decide how `rank` should be implemented, the value $k = \mathrm{rank}(\mathbf{M}^{\mathrm{sk}})$ on Line 4 of `cqrrpt_core` is really something I *can choose depending on the computational approach*.

That established, let us see how k affects reconstruction error and orthogonality loss. In the tradition of adding zero and applying the triangle inequality, one can obtain the following bound on reconstruction error committed in the preconditioning step:

$$\left\|\mathbf{M}_n - \mathbf{M}^{\text{pre}}\mathbf{R}_k^{\text{sk}}\right\|_{\text{F}} \leq \underbrace{\left\|\mathbf{M}_n - \mathbf{M}_k(\mathbf{A}_k^{\text{sk}})^{-1}\mathbf{R}_k^{\text{sk}}\right\|_{\text{F}}}_{\text{truncation error}} + \left\|(\mathbf{M}_k - \mathbf{M}^{\text{pre}}\mathbf{A}_k^{\text{sk}})(\mathbf{A}_k^{\text{sk}})^{-1}\mathbf{R}_k^{\text{sk}}\right\|_{\text{F}}. \tag{2.4}$$

Of the two terms in this upper bound, the *truncation error* is more opaque. To better understand it, suppose δ is the distortion of $\mathbf{S}$ for the range of $\mathbf{M}$. I claim that under mild assumptions for the accuracy of operations on Lines 2 and 3, and an assumption that $\mathbf{A}_k^{\text{sk}}$ is not too ill-conditioned, one can bound

$$\left\|\mathbf{M}_n - \mathbf{M}_k(\mathbf{A}_k^{\text{sk}})^{-1}\mathbf{R}_k^{\text{sk}}\right\|_{\text{F}} / \|\mathbf{M}\|_{\text{F}} \lesssim \sqrt{1+\delta}\left\|\mathbf{C}_k^{\text{sk}}\right\|_{\text{F}} / \left\|\mathbf{M}^{\text{sk}}\right\|_{\text{F}}. \tag{2.5}$$

Since $\|\mathbf{C}_k^{\text{sk}}\|_{\text{F}}$ is decreasing with k, this bound suggests that k should be large to keep reconstruction error under control.

Unfortunately, if $\mathbf{M}$ is ill-conditioned, then choosing k too large can pose severe problems. The basic reason for this stems from the fact that $\kappa(\mathbf{A}_n^{\text{sk}})$ can be as large as $\kappa(\mathbf{SM})$, which in turn can be as large as $\sqrt{\frac{1+\delta}{1-\delta}}\,\kappa(\mathbf{M})$. This suggests that $\mathbf{A}_k^{\text{sk}}$ can be ill-conditioned when k is large, even when δ is small. This matters since larger condition numbers for $\mathbf{A}_k^{\text{sk}}$ risk larger rounding errors at the step when the $m \times k$ matrix $\mathbf{M}^{\text{pre}}$ is formed. If this step is performed inaccurately, then $\kappa(\mathbf{M}^{\text{pre}})$ might be far from one even if δ is small, which risks orthogonality loss in $\mathbf{Q}_k$ that cannot be controlled by the choice of sketching distribution for Algorithm 1.

The tension described above raises an important question. Can I be certain that there *exists* a truncation index k so that both reconstruction error and orthogonality loss are kept near machine precision? [93, Theorem 10] states that the answer is positive, provided the QRCP of $\mathbf{M}^{\text{sk}}$ is (sufficiently) strongly rank-revealing.

2.2.2 Determining numerical rank in practice

I suggest a flexible two-stage approach to numerical rank estimation. The idea is that since orthogonality loss increases with truncation rank and reconstruction error (typically) decreases with truncation rank, it is reasonable to choose the largest rank where some estimate for the orthogonality loss is below a specified tolerance (say, $\epsilon_{\text{tol}} = 100\mathsf{u}$).

The first stage begins by finding a crude upper bound on the numerical rank of $\mathbf{R}^{\mathrm{sk}}$. For example, one can set

$$k_o = \min\{\ell \,:\, \|\mathbf{C}_\ell^{\mathrm{sk}}\|_{\mathrm{F}} \le \mathsf{u}s\}, \tag{2.6a}$$

where s is the maximum entry of $|\mathbf{R}^{\mathrm{sk}}|$. This bound is cheap to compute and ensures $\|\mathbf{C}_{k_o}^{\mathrm{sk}}\|_2 \le \mathsf{u}\|\mathbf{R}^{\mathrm{sk}}\|_2$. The rest of the first stage consists of forming the preconditioned matrix $\mathbf{M}_{k_o}^{\mathrm{pre}} = (\mathbf{M}_{k_o})(\mathbf{A}_{k_o}^{\mathrm{sk}})^{-1}$ and computing the Cholesky decomposition of $(\mathbf{M}_{k_o}^{\mathrm{pre}})^{\mathrm{T}}(\mathbf{M}_{k_o}^{\mathrm{pre}})$.

The second stage uses a function for estimating condition numbers of triangular matrices. Given such a function, `cond`, I estimate the orthogonality loss of choosing truncation rank ℓ by $\mathsf{u}\cdot(\mathtt{cond}(\mathbf{A}_\ell^{\mathrm{pre}}))^2$. The motivation for this is that if `cond` bounds condition numbers from above, then choosing ℓ to satisfy $\mathtt{cond}(\mathbf{A}_\ell^{\mathrm{pre}}) \le \sqrt{\epsilon_{\mathrm{tol}}/\mathsf{u}}$ bounds orthogonality loss by $\mathcal{O}(\epsilon_{\mathrm{tol}})$. Therefore, the formal goal of the second stage is to find

$$k = \max\{\ell \,:\, \mathtt{cond}(\mathbf{A}_\ell^{\mathrm{pre}}) \le \sqrt{\epsilon_{\mathrm{tol}}/\mathsf{u}}\,\}. \tag{2.6b}$$

When computing (2.6b), assume that $\mathtt{cond}(\mathbf{A}_{\ell+1}^{\mathrm{pre}}) \ge \mathtt{cond}(\mathbf{A}_\ell^{\mathrm{pre}})$. This assumption holds for the true condition number, as can be seen by applying the eigenvalue interlacing theorem to the Gram matrices of $\mathbf{A}_{\ell+1}^{\mathrm{pre}}$ and $\mathbf{A}_\ell^{\mathrm{pre}}$ (the latter being a submatrix of the former). This assumption is useful because it lets us compute (2.6b) by binary search over ℓ in $\{0,\ldots,k_o-1\}$. Even if this assumption does not hold, the only risk in deciding k by binary search is underestimating numerical rank.

Remark 11 (What if Cholesky fails in stage one?). Set $\mathbf{G} = (\mathbf{M}_n^{\mathrm{pre}})^{\mathrm{T}}(\mathbf{M}_n^{\mathrm{pre}})$. Consider an index k_o where $\mathbf{G}(0{:}k_o, 0{:}k_o)$ is positive definite, but $\mathbf{G}(0{:}(k_o+1), 0{:}(k_o+1))$ is not. Running LAPACK's `POTRF` function on $\mathbf{G}$ will produce an error. However, the leading k_o-by-k_o submatrix of the output $\mathbf{R}$ factor will be well-formed and satisfy $\mathbf{R}(0{:}k_o, 0{:}k_o)^{\mathrm{T}}\mathbf{R}(0{:}k_o, 0{:}k_o) = \mathbf{G}(0{:}k_o, 0{:}k_o)$. Therefore, if `POTRF` fails at Line 7 and returns an error code k, then $k_o = k-1$ can be taken as an initial estimate for numerical rank.

2.3 Pivot quality experiments

This section gives experimental comparisons of pivot quality using the LAPACK default (GEQP3) versus those produced by CQRRPT *based on* that default. Of

course, the results of such a comparison depend heavily on the distribution family $\mathscr{F}$ and the embedding factor γ that determines the distribution of the sketching operator. Therefore, I take this as an opportunity to show how one might set $\mathscr{F}$ and γ in practice.

Background on leverage scores and coherence. Section 2.1.2 presented results on how γ can be chosen to achieve oblivious subspace embedding properties for various sketching families. The relevant result for SASOs, Theorem 2.4, provides worst-case bounds that are valuable in theoretical analysis. However, some subspaces are "easier to sketch" with SASOs than this result would suggest, in the sense that the subspace embedding property can reliably be obtained with a far smaller embedding factor or sparsity parameter.

Leverage scores are a useful concept for understanding when a subspace might be easy or hard to accurately sketch with a given distribution. They quantify the extent to which a low-dimensional subspace aligns with coordinate subspaces [88, 50, 39].

Definition 2.3. *Let* $\mathbf{U}$ *be an* $m \times n$ *orthonormal matrix. The* i^{th} leverage score *of* range($\mathbf{U}$) *is the squared row-norm* $\|\mathbf{U}[i,:]\|_2^2$.

When using sketching operators such as SASOs or SRHTs, it is standard to summarize leverage scores with a concept called *coherence*. This is essentially a condition number for random sampling or embedding algorithms [88, 50, 39]. Formally, the coherence of an $m \times n$ matrix $\mathbf{M}$ is m times the largest leverage score of range($\mathbf{M}$). It is well-documented in the literature that matrices with higher coherence are harder to sketch.

Pivot quality metrics. I use two pivot quality metrics. The first is the Frobenius norms of the matrices $\mathbf{C}_\ell$ in block 2-by-2 partitions of $\mathbf{R}$. This has the natural interpretation as the norm of a rank-ℓ approximation of $\mathbf{M}_n - \mathbf{Q}_\ell \mathbf{R}_\ell$; I plot this metric as ratios $\|\mathbf{C}_\ell^{\text{qp3}}\|_{\text{F}} / \|\mathbf{C}_\ell^{\text{cqrrpt}}\|_{\text{F}}$. The second pivot quality metric is the ratios of $r_{ii} := |\mathbf{R}(i,I)|$ to the singular values of $\mathbf{M}$. If $\mathbf{R}$ comes from `GEQP3` then this ratio can be quite bad in the worst case. Letting σ_i denote the i^{th} singular value of $\mathbf{M}$, this only guarantees that $\phi_i := r_{ii}/\sigma_i$ is between $(n(n+1)/2)^{-1/2}$ and 2^{n-1} [72]. Since there is a chance for large deviations, I plot r_{ii}/σ_i for `CQRRPT` and `GEQP3` separately (rather than plotting the ratio $r_{ii}^{\text{qp3}}/r_{ii}^{\text{cqrrpt}}$).

2.3.1 Example low-coherence matrices

Here, I consider tall $m \times n$ matrices with $m = 2^{17} = 131{,}072$ and $n = 2000$ whose spectrum falls into one of the two following categories:

- Matrices for which the first ten percent of their singular values are all equal to one and the rest are decaying polynomially down to $1/\kappa(\mathbf{M}) = 10^{-10}$.
- Matrices with a four-step "staircase-shaped" spectrum. The first quarter of the singular values are 1, the next quarter are $8 \cdot 10^{-10}$, the quarter after that are $4 \cdot 10^{-10}$, and the final quarter are all 10^{-10}.

The singular vectors of these matrices were generated by orthogonalizing the columns of matrices with iid Gaussian entries. Generating the left singular vectors in this way is a standard method for generating low-coherence matrices.

Fig. 2.1 visualizes metrics of `CQRRPT`'s pivot quality for these matrices, when it is configured to use maximally aggressive dimension reduction via SASOs ($\gamma = 1$ and one nonzero per column). For both matrices, the ratio of `GEQP3`'s residual-norm metric to that of `CQRRPT` is close to 1. Although it is noteworthy that the deviations of this ratio from 1 are more pronounced at ranks that coincide with rapid drops in the singular values. For the second pivot quality metric, observe that the curves for `CQRRPT` and `GEQP3` are extremely similar (in fact, visually coincident) for the two types of matrices.

2.3.2 An example high-coherence matrix

Here, I consider a matrix $\mathbf{M}$ of the same dimensions as before, $m \times n = 131{,}072 \times 2000$, constructed in three steps. The first step is to vertically stack $c = \lfloor m/n \rfloor$ copies of the $n \times n$ identity matrix and one copy of the first $m - cn$ rows of the $n \times n$ identity. The second step is to select n rows of $\mathbf{M}$ at random and multiply them by 10^{10}. Finally, the third step is to multiply $\mathbf{M}$ on the right by a random $n \times n$ orthogonal matrix. The left singular vectors are not explicitly generated in this matrix to ensure its high coherence.

Column one of Fig. 2.2 shows pivot quality results when applying `CQRRPT` to $\mathbf{M}$ with overly-aggressive dimension reduction; column two of Fig. 2.2 shows the analogous data for only modestly more expensive dimension reduction. The figures' combined message is twofold: that there is good reason to use $\gamma > 1$ and more than

Algorithm 2 : `Core computations in CQRRPT`

Input: A matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, and a sketching operator $\mathbf{S} \in \mathbb{R}^{d \times m}$ where $n \leq d \leq m$.

function CQRRPT_CORE($\mathbf{M}, \mathbf{S}$)
  Sketch $\mathbf{M}^{\text{sk}} = \mathbf{S}\mathbf{M}$
  Decompose $[\mathbf{Q}^{\text{sk}}, \mathbf{R}^{\text{sk}}, \boldsymbol{J}] = \texttt{qrcp}(\mathbf{M}^{\text{sk}})$
  // The n-vector $\boldsymbol{J}$ is returned in full, regardless of subsequent steps
  Determine $k = \texttt{rank}(\mathbf{R}^{\text{sk}})$
  Precondition $\mathbf{M}^{\text{pre}} = \mathbf{M}_k(\mathbf{A}_k^{\text{sk}})^{-1}$
  // Recall the notation $\mathbf{M}_k := \mathbf{M}(:, \boldsymbol{J}(0{:}k))$ and $\mathbf{A}_k^{\text{sk}} := \mathbf{R}^{\text{sk}}(0{:}k, 0{:}k)$
  Compute $\mathbf{G} = (\mathbf{M}^{\text{pre}})^{\text{T}}(\mathbf{M}^{\text{pre}})$
  Decompose $\mathbf{R}^{\text{pre}} = \texttt{chol}(\mathbf{G})$
  Set $\mathbf{Q}_k = (\mathbf{M}^{\text{pre}})(\mathbf{R}^{\text{pre}})^{-1}$
  Undo preconditioning $\mathbf{R}_k = \mathbf{R}^{\text{pre}}\mathbf{R}_k^{\text{sk}}$
  // Recall the notation $\mathbf{R}_k^{\text{sk}} := \mathbf{R}^{\text{sk}}(0{:}k, :)$
  return $\mathbf{Q}_k$, $\mathbf{R}_k$, $\boldsymbol{J}$

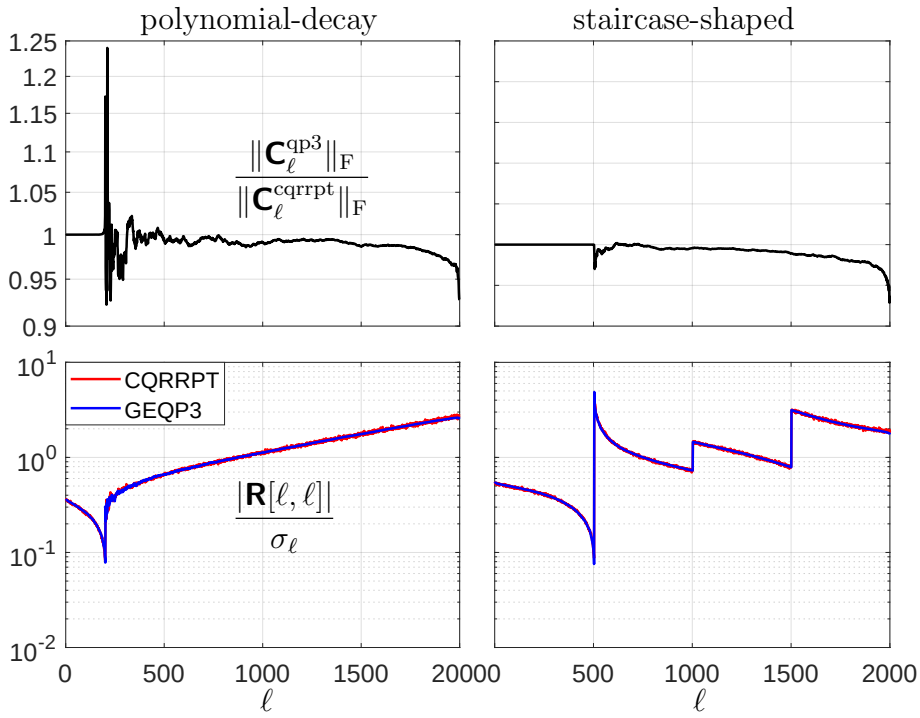


Figure 2.1: Pivot quality results for low-coherence matrices with two types of spectral decay.

one nonzero per column in $\mathbf{S}$, and that extensive parameter tuning is *not* needed to achieve reliable results from `CQRRPT` at low computational cost.

2.4 `CQRRPT` output quality

Recall that Section 2.3 discussed the "quality of pivots," i.e., the *quality of reconstruction* provided by `CQRRPT`. While the results presented in Section 2.3.1 and Section 2.3.2 conveyed the message that the reconstruction error of `CQRRPT` matches that of `GEQP3` and linked the theoretical results from Section 2.1.2 to an empirical setting, one could still criticize the original manuscript [93] for the lack of rigor in assessing how "good" the results output by `CQRRPT` are in practice. Specifically, Fig. 2.1 and Fig. 2.2 only present the similarity of `CQRRPT` and `GEQP3`'s results visually, hence relying on the notorious "eyeball norm" test. Moreover, Section 2.3 only discussed the reconstruction error and not the orthogonality loss in the $\mathbf{Q}$-factor, which, as discussed in Section 2.2, is crucial for `CQRRPT`. Lastly, Section 2.3 gives no perspective on the empirically observed variance in the results from different runs of `CQRRPT`. This section aims to address each of the described limitations of the existing work.

This section presents table Table 2.2 reporting the relative reconstruction error and orthogonality loss across `GEQP3` and `CQRRPT` algorithm runs on matrices with low and high coherence, featured previously in Section 2.3.1 and Section 2.3.2. The test matrices are of size $m \times n$, with $m = \{16{,}000, 32{,}000, 64{,}000\}$ and $n = \{50, 8{,}000\}$. I showcase the average error attained by `CQRRPT` across 10 runs for each input size and type, along with the error ranges observed, using the notation $x \pm y$, where x is the average error and y is the error range. The results for `GEQP3` are only captured once. In the `CQRRPT` runs, the embedding factor, γ, is set to the default value of 1.25; the number of nonzeros per column in the sketching operator is set to the default value of 4.

Remark 12. Note that the results shown do not claim 16-digit accuracy in all cases; they simply highlight that the two methods achieve the same number of correct digits.

Table 2.2 shows that for all types of inputs tested, `GEQP3` and `CQRRPT` achieve similar accuracy results (specifically, the algorithms achieve the same number of digits of accuracy). Moreover, in all test cases, the error ranges in `CQRRPT` are much smaller than the average error, and, as such, can be disregarded.

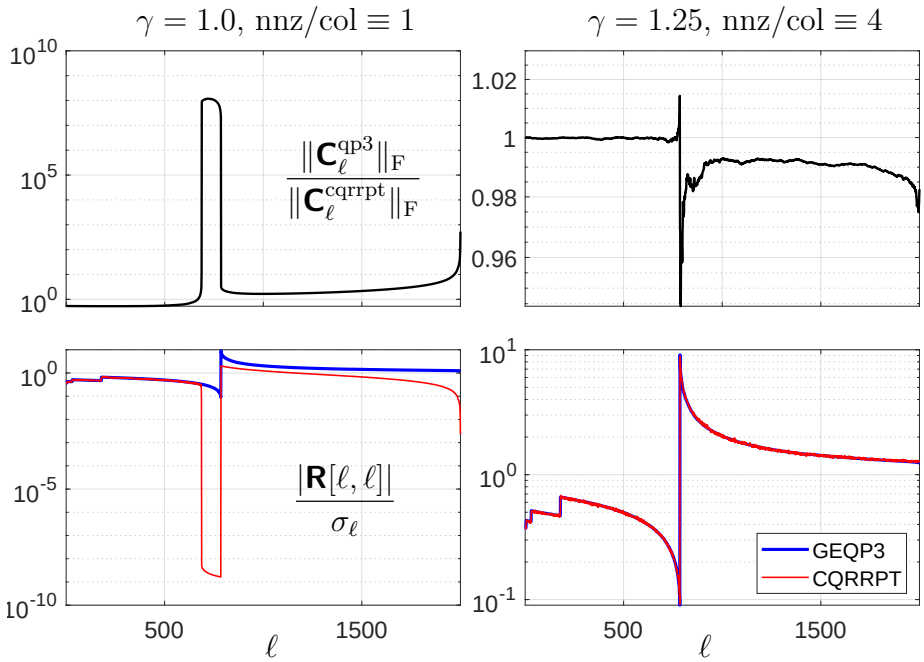


Figure 2.2: Pivot quality results for a high-coherence matrix under two choices of sketching distribution parameters.

Table 2.2: Reconstruction error and orthogonality loss in `GEQP3` and `CQRRPT` with varying column sizes, captured across tall matrices of varying types and numbers of input rows.

	m	spectrum($\mathbf{M}$)	$n = 50$		$n = 8000$	
			$\frac{\|\mathbf{M}(:, \boldsymbol{J}) - \mathbf{QR}\|_{\mathrm{F}}}{\|\mathbf{M}\|_{\mathrm{F}}}$	$\frac{\|\mathbf{I} - \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\|_{\mathrm{F}}}{\sqrt{n}}$	$\frac{\|\mathbf{M}(:, \boldsymbol{J}) - \mathbf{QR}\|_{\mathrm{F}}}{\|\mathbf{M}\|_{\mathrm{F}}}$	$\frac{\|\mathbf{I} - \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\|_{\mathrm{F}}}{\sqrt{n}}$
`GEQP3`	16000	polynomial	2e-16	3e-16	9e-16	9e-16
		staircase	2e-16	3e-16	1e-15	9e-16
		spiked	2e-16	2e-16	1e-15	8e-16
	32000	polynomial	2e-16	2e-16	1e-15	6e-16
		staircase	2e-16	3e-16	1e-15	6e-16
		spiked	2e-16	2e-16	1e-15	7e-16
	64000	polynomial	2e-16	3e-16	1e-15	5e-16
		staircase	2e-16	3e-16	1e-15	5e-16
		spiked	2e-16	2e-16	1e-15	6e-16
`CQRRPT`	16000	polynomial	3e-16 ± 1e-17	8e-16 ± 7e-17	1e-15 ± 4e-19	2e-15 ± 8e-18
		staircase	3e-16 ± 1e-17	8e-16 ± 1e-16	1e-15 ± 3e-19	2e-15 ± 9e-18
		spiked	3e-16 ± 2e-17	8e-16 ± 1e-16	1e-15 ± 2e-18	2e-15 ± 1e-17
	32000	polynomial	3e-16 ± 7e-18	9e-16 ± 9e-17	1e-15 ± 2e-19	2e-15 ± 7e-18
		staircase	3e-16 ± 1e-17	8e-16 ± 1e-16	1e-15 ± 3e-19	2e-15 ± 8e-18
		spiked	3e-16 ± 1e-17	8e-16 ± 1e-16	1e-15 ± 1e-18	2e-15 ± 1e-17
	64000	polynomial	3e-16 ± 2e-17	8e-16 ± 1e-16	1e-15 ± 3e-19	2e-15 ± 1e-17
		staircase	3e-16 ± 2e-17	8e-16 ± 1e-16	1e-15 ± 4e-19	2e-15 ± 1e-17
		spiked	3e-16 ± 1e-17	8e-16 ± 1e-16	1e-15 ± 4e-18	3e-15 ± 2e-17

2.5 Algorithm benchmarking

The main set of experiments in this section compares `CQRRPT` to other algorithms for QR and QRCP. Specifically, I compare my algorithm to the following.

- `GEQRF` - standard unpivoted Householder QR.
- `GEQR` - unpivoted QR, which dispatches either a specialized algorithm for QR of tall and skinny matrices *or* a general-purpose algorithm according to implementation-specific logic. The selected algorithm is nominally based on a prediction of which algorithm will be more efficient for the given matrix. Intel MKL documentation does not promise that `GEQR` makes the optimal decision in this regard.
- `GEQP3` - standard pivoted QR.
- `GEQPT` - composed of performing `GEQR` on an input matrix to get the factors $\mathbf{Q}_1$ and $\mathbf{R}_1$ and then getting data $(\mathbf{Q}_2, \mathbf{R}, \boldsymbol{J})$ by running `GEQP3` on $\mathbf{R}_1$; the final matrix $\mathbf{Q}$ is defined implicitly as the product of $\mathbf{Q}_1$ and $\mathbf{Q}_2$.
- `sCholQR3` - shifted CholeskyQR3, my implementation of [58, Algorithm 4.2], using the Frobenius norm on Line 3 of said algorithm.

Remark 13. Among the algorithms listed, only `sCholQR3` outputs the $\mathbf{Q}$ factor explicitly, in the same manner as `CQRRPT`.

Varying matrix sizes. The first set of experiments was run on $m \times n$ matrices with $m = 65{,}536$ (Fig. 2.3, left) and $m = 131{,}072$ (Fig. 2.3, right), with n varying as the powers of two from 512 to 8192. The embedding factor, γ, is set to the default value of 1.25; the number of nonzeros per column in the sketching operator is set to the default value of 4.

The results show that with an increase in input matrix size, `CQRRPT` outperforms alternative pivoted schemes and has either comparable or superior performance to unpivoted schemes. The plots show an order of magnitude acceleration over `GEQP3`, and accelerations of up to 2x over `GEQR` and `GEQPT`, and 1.6x over `GEQRF`, for matrices with 131,072 rows. On a plot with 65,536 rows, the performance of `CQRRPT` declines after 2048 columns due to the fact that the matrix is not tall enough.

Note that GEQR fails to match the performance of GEQRF and consequently causes poor performance in GEQPT. This is likely due to the fact that the matrix sizes that I use in the experiments are not considered "tall enough" by the internal metric of the MKL implementation of GEQR. For experiments with matrices where m is several orders of magnitude larger than n, I refer the readers to Section 2.5.2.

Varying number of threads. Table 2.3 depicts the performance scaling of various algorithms with the change in number of threads used. For the dimensions of the test matrix used here, CQRRPT remains the fastest algorithm for any number of threads.

It is worth noting that CQRRPT has another possible performance advantage over the alternative algorithms that come from the output format that CQRRPT uses. Namely, having an explicit representation of a $\mathbf{Q}$-factor ensures that the factor can be applied via a fast GEMM function as opposed to a slower ORMQR, which is used to apply an implicitly-stored $\mathbf{Q}$-factor. I acknowledge the existence of applications where having an implicitly stored $\mathbf{Q}$-factor is a necessity (for example, when the full square $\mathbf{Q}$ is required). In the context of such applications, the output format of CQRRPT would be disadvantageous.

2.5.1 Profiling and optimizing CQRRPT

As seen from Algorithm 2, CQRRPT consists of a handful of subcomponents. Understanding how each such subcomponent affects overall runtime for varying input parameters is important for potential future optimizations of CQRRPT. To give perspective on this, I present the runtime breakdown of CQRRPT in Fig. 2.4, highlighting the percentage of CQRRPT's runtime occupied by each of its respective subroutines. As before, the experiments are conducted on $m \times n$ iid random test matrix with $m = 131{,}072$ and n as powers of two from 32 to 16,384. I also set the embedding factor, γ, to the default value of 1.25.

An immediate observation to be made here is that QRCP and CholeskyQR become the most time-consuming subroutines as the matrix size increases. Additionally, QRCP clearly becomes the main computational bottleneck. As the embedding dimension parameter greatly affects the timing of QRCP, it is important to understand how the choice of this parameter affects CQRRPT's overall wall clock time.

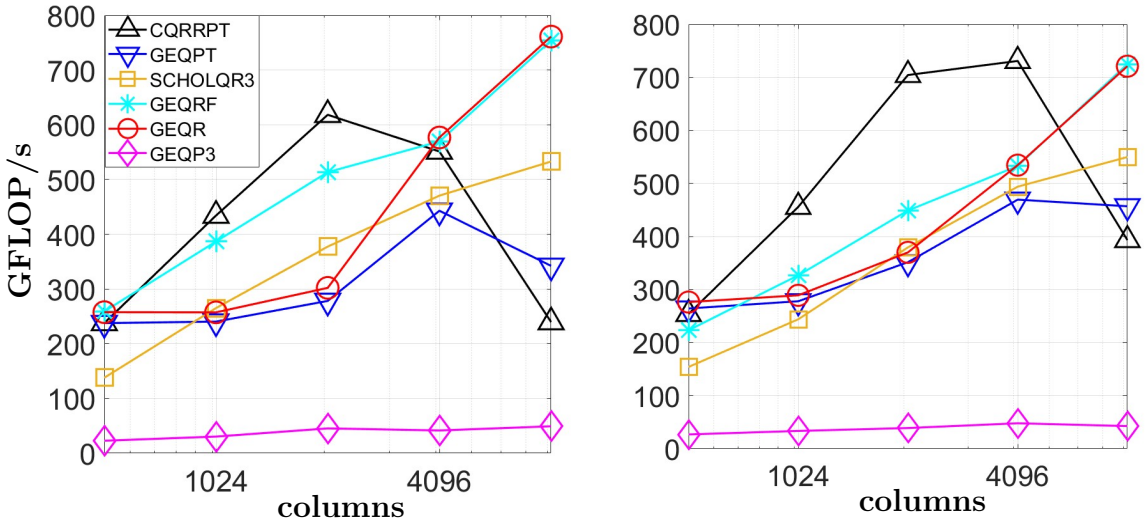


Figure 2.3: QR schemes performance comparisons for matrices with fixed numbers of rows (2^{16} = 65,536, and 2^{17} = 131,072 respectively) and varying numbers of columns $(512, \ldots, 8192)$. `CQRRPT` remains the fastest algorithm until the `QRCP` on the sketch becomes too expensive, as discussed further in Fig. 2.4.

Table 2.3: Canonical GFLOP/s scaling with the number of threads used, given $\mathbf{M} \in \mathbb{R}^{131{,}072 \times 2048}$. At this particular input matrix size, `CQRRPT` outperforms all other listed algorithms.

Threads	`CQRRPT`	`GEQPT`	`sCholQR3`	`GEQRF`	`GEQR`	`GEQP3`
8	250	159	147	121	166	21
16	490	286	276	274	300	25
24	577	351	316	343	369	29
36	677	372	388	411	393	50
48	704	351	379	448	370	39

Effect of varying the embedding dimension parameter. Let's take a look at the effect of varying the embedding dimension in practice. I use iid random input matrices of sizes $131{,}072 \times n$ for $n \in \{1024, 2048, 4096\}$ with the ratio $\gamma = d/n$ varying from 1 to 4 in steps of 0.5. Results are presented in Fig. 2.5.

Note that for a smaller column size n, increasing the parameter γ has little to no effect on `CQRRPT`'s overall runtime, as the portion of runtime dedicated to performing the QRCP on a sketch is smaller. However, as the figure suggests, for a larger $n = 4096$, the effect of increasing γ is rather noticeable. Recalling the results from Section 2.3.1, using small values of γ can reliably produce accurate results for matrices of low coherence. Therefore, a good starting point for the value of this parameter would be somewhere between one and two in such cases.

Performing `HQRRP` on the sketch. At this point, the reader has been repeatedly shown that `CQRRPT`'s method for QRCP of $\mathbf{M}^{\text{sk}}$ can decisively impact its runtime when n is large. Here, I explore the possibility of accelerating this operation by replacing `CQRRPT`'s call to `GEQP3` with a call to `HQRRP` (Householder QR Factorization With Randomization for Pivoting, see [90]).

I incorporated an `HQRRP` implementation into RandLAPACK and ran experiments comparing `HQRRP` to `GEQP3` directly. In these experiments, I use square matrices of sizes $n \times n$ for $n \in \{1000, 2000, \ldots, 10000\}$ with varying numbers of OpenMP threads used. An important tuning parameter to consider in `HQRRP` benchmarking is the block size. Upon experimenting with different `HQRRP` block sizes, I have concluded that on the particular system that I am utilizing, using the block size of 32 results in the best performance for `HQRRP`. Using other block sizes in $\{8, 16, 64, 128, 256\}$ resulted in worse performance than what I report in this paper. Results are presented in Fig. 2.6.

As seen in Fig. 2.6, `HQRRP` achieves its peak performance with 24 threads for large matrix sizes, undoubtedly beating the performance of `GEQP3`. However, for small input matrices, using a large number of threads results in `HQRRP` failing to outperform `GEQP3`. When 48 threads are used, the performance of `HQRRP` stagnates, similar to other QR factorization algorithms; see Section 2.5.2 for an explanation of why this happens.

Fig. 2.6 allows us to conclude that it is preferred to use 8 threads when running `HQRRP` on the matrices with $n \leq 6000$; using 24 threads would be optimal for matrices of larger sizes. Additionally, using 24 threads should result in the best performance

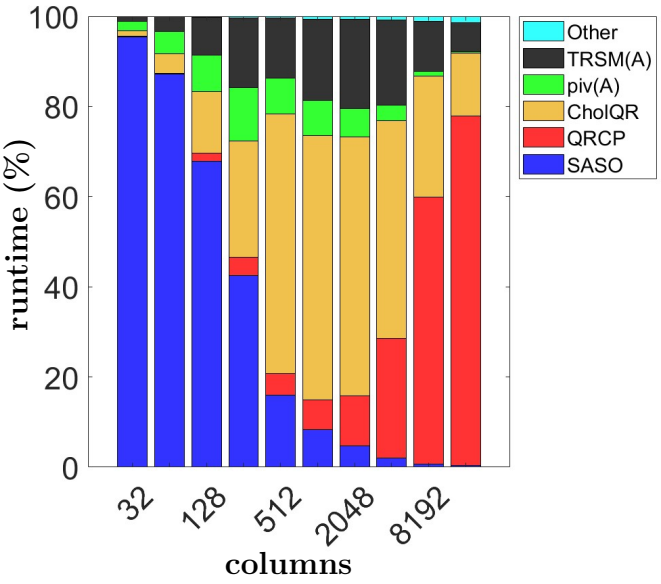


Figure 2.4: Percentages of `CQRRPT`'s runtime, occupied by its respective subroutines. Note that the cost of sketching becomes negligible for larger matrices. Note also that when $d \geq n$, the cost of applying QRCP to the $d \times n$ sketch grows as $\Omega(n^3)$. By contrast, the cost of applying CholeskyQR to the $m \times n$ preconditioned matrix grows as $\mathcal{O}(mn^2)$. Therefore, it is reasonable that QRCP consumes a larger fraction of runtime as n increases.

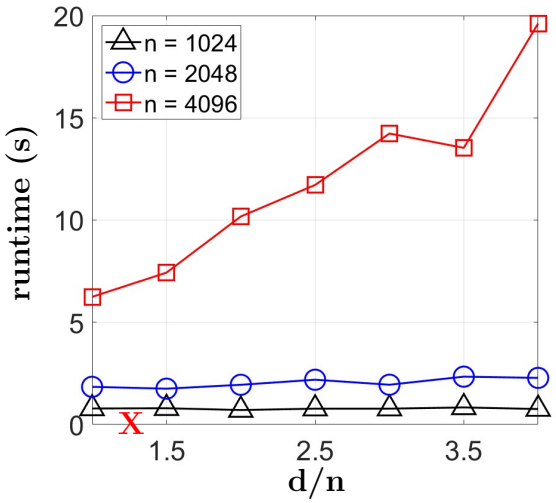


Figure 2.5: Effect of varying the embedding factor $\gamma \in \{1, 1.5, 2, ..., 4\}$ for matrices of sizes $131072 \times \{1024, 2048, 4096\}$. Runtime represents the wall clock time for the full execution of `CQRRPT`. An increase of the embedding dimension has a larger effect on wider matrices, as QRCP becomes more expensive with the increased number of columns, as shown in Fig. 2.4. Note that the default value of $\gamma = 1.25$ is marked with **X**; performance in this case can be inferred by interpolating between the first two data points of each series.

of GEQP3. Assuming CQRRPT used ideal parameter choices for HQRRP and GEQP3 on the given machine, I would expect the HQRRP version to outperform the GEQP3 version when $n \geq 3000$.

2.5.2 CQRRPT on extremely tall matrices

The matrices used in Section 2.5 experiments were tall, but not as tall as some readers might anticipate. Therefore, here I consider matrices with one million rows and $\{32, 64, \ldots, 16{,}384\}$ columns. Fig. 2.7 presents performance results for the algorithms considered previously. It shows that for very thin matrices ($n \leq 256$), CQRRPT may be inefficient compared to alternative algorithms. This can be understood with performance profiling data in Fig. 2.8. Specifically, for these extremely tall and thin matrices, the runtime is dominated by sketching, and the time to pivot $\mathbf{M}$ before preconditioning takes nearly as long as CholeskyQR. The experiments were run on the machine described in Table 2.1.

Remark 14. Fig. 2.7 does not depict GEQR and GEQPT results for the number of columns larger than 2048. I initially suspected that this issue may be related to an internal overflow in some Intel MKL function that is out of my control. In reality, however, this was related to my not setting the dblas_int parameter in BLAS++ during configuration, which caused certain routines to be configured for 32-bit integers. At the time of this dissertation being written, I no longer had access to the machine specified in Table 2.1, and as such, was not able to remake Fig. 2.7 properly. Instead, in Section 3.9, I present a new set of CQRRPT experimental results, inspired by the project described in Chapter 3.

2.6 Chapter conclusion

This chapter introduces CQRRPT, a randomized algorithm for computing a QRCP factorization of a tall matrix. The algorithm consists of two ingredients: obtaining the pivot order and an approximate $\mathbf{R}$-factor from a small random sketch, and retrieving the complete QRCP factorization by a preconditioned CholeskyQR. In comparison to standard QR algorithms, such as Householder QR, TSQR, and their pivoted (possibly randomized) derivatives, CQRRPT requires $\frac{4}{3}$x less flops[‖] and is better suited to modern

[‖] to compute QRCP factorization in explicit form

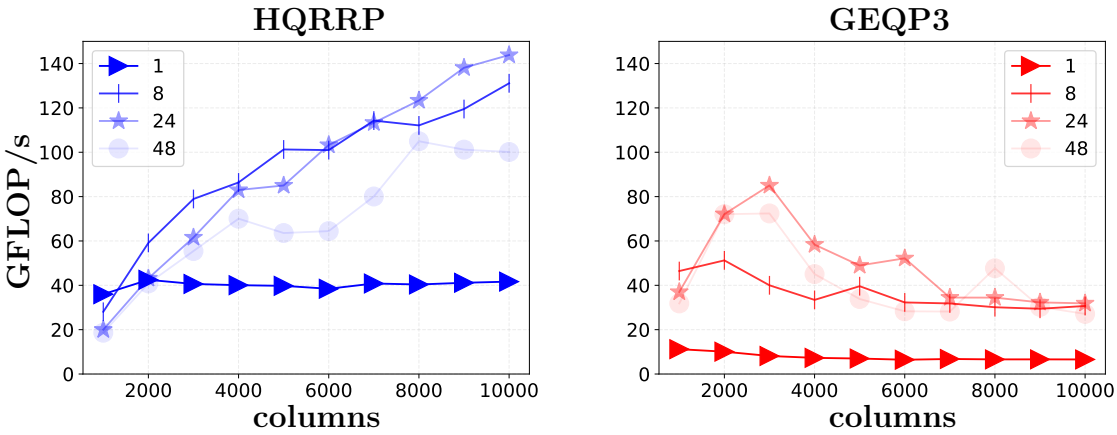


Figure 2.6: Canonical flop rates for `HQRRP` vs `GEQP3` with `OMP_THREADS` in $\{1, 8, 24, 48\}$. The experimental speedup of `HQRRP` over `GEQP3` for the larger matrix sizes is significant, but it does not match the speedup reported in [90]; this may be due to a variety of factors (primarily, the machine used for benchmarking).

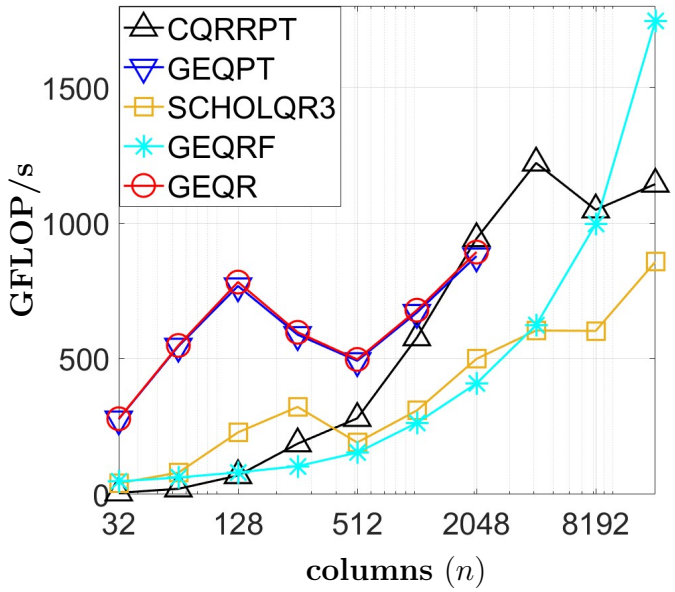


Figure 2.7: QR schemes performance comparisons for matrices with one million rows and varying numbers of columns ($\{32, 64, \ldots, 16{,}384\}$).

computational architectures due to fewer global synchronization points, fewer data passes, and/or more straightforward reduction operator. Notably, `CQRRPT` provides numerical stability even for rank-deficient matrices, in contrast to existing sketched QR and CholeskyQR methods. This makes the algorithm an appealing tool for orthogonalization.

I established rigorous conditions for the random sketch required by `CQRRPT` to attain the rank-revealing property and numerical stability (a detailed explanation of these results is discussed in the original published work, [93]). These conditions can be met by employing a sketching distribution that possesses an oblivious subspace embedding property. The robustness and performance advantages of `CQRRPT` were confirmed through extensive numerical experiments of a RandLAPACK implementation based on RandBLAS. In particular, I present flop rate improvements of up to 10x compared to the standard pivoted QR routine `GEQP3` and 1.4x compared to the unpivoted QR routine `GEQRF` from LAPACK.

This chapter also highlights several promising research directions. First, the stability requirements and the efficiency of `CQRRPT` in certain scenarios can be improved by using more sophisticated sketching operators and schemes for computing the sketch-orthonormal preconditioned matrix. The performance of `CQRRPT` in modern computing environments such as GPUs, distributed systems, and multi-(or low-)precision arithmetic systems also remains to be explored. Furthermore, `CQRRPT` can be extended to be applicable to matrices with any aspect ratio by integrating ideas from this paper with ideas from [122, 54, 89, 90] for `HQRRP` and the related randomized and communication-avoiding QR techniques.

The latter direction is explored as part of Chapter 3.

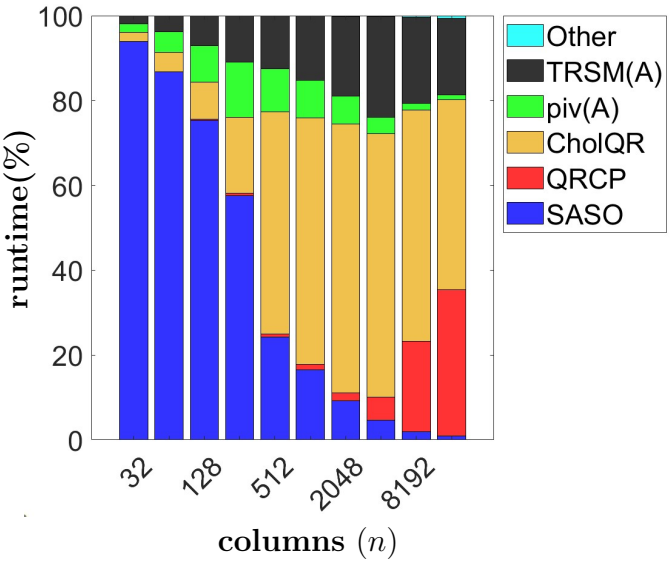


Figure 2.8: Percentages of runtime used by `CQRRPT's` subroutines, for matrices with one million rows and the indicated number of columns. Note that although CholQR occupies the dominant portion of runtime across the larger ($n \geq 512$) matrix sizes, its runtime is primarily comprised of vector-vector operations for the smaller matrix sizes. As the matrix size increases, `TRSM` routine gradually achieves a `GEMM`-like performance level.

Chapter 3

Anatomy of High-Performance Column-Pivoted QR Decomposition

In Chapter 2, I introduced a novel QRCP algorithm called "Cholesky QR with Randomization and Pivoting for Tall matrices" (`CQRRPT`). Despite its exceptional features, the main limitation of this scheme lies in its name: `CQRRPT` is only applicable to rectangular matrices, where the number of rows is larger than the number of columns. This limitation of `CQRRPT` disqualifies it from candidacy for a definitive "solution" to the problem of designing a QRCP algorithm.

This chapter introduces an algorithmic framework, referred to as *BQRRP*,* which stands for "Blocked QR with Randomization and Pivoting." BQRRP serves as a natural evolution of `CQRRPT`, encapsulating much of `CQRRPT`'s core features, while resolving its main aforementioned limitation.

This chapter does not dwell too much on the theoretical properties of the algorithmic framework I propose. Rather, it concentrates on providing many details on the practical aspects of what goes into a high-performance QRCP implementation. As such, while this work is particularly beneficial for numerical software engineers, it is accessible to a broader audience who may find value in exploring it at their own pace.

*pronounced "bee-crip"

The contribution. In the framework I describe, one implements a `BQRRP` instantiation by selecting a set of subroutines, most suitable for a system on which `BQRRP` is to be used. The main contributions of this work are: (a) a detailed description of the design decisions in developing the BQRRP framework, including the elaboration on most suitable practical choices for BQRRP subroutines on two modern systems; and (b) CPU and GPU implementations of `BQRRP` that allow users to configure the algorithm in a way that is most suitable for their needs, provided as part of the RandLAPACK library.

As I show in Section 3.7, an implementation of `BQRRP_CPU` not only outperforms `HQRRP` (described in Section 1.1.1), but also it is up to two orders of magnitude faster than the standard pivoted QR (`GEQP3`) implementation available in MKL 2025 [76]. `BQRRP_GPU` exhibits excellent throughput and shows reasonable performance relative to an unpivoted QR (`GEQRF`) implementation available in NVIDIA's cuSOLVER 11.6.1.9 [105]. This makes `BQRRP_GPU` a strong candidate for adoption as a standard general QRCP method by GPU linear algebra library vendors.

The dominant cost of `BQRRP_CPU` comes from a routine that applies an implicitly-stored orthonormal matrix to a portion of the input data in a loop, which is elaborated on in Section 3.1.4. On a GPU, in addition to this bottleneck, there is the added complication in the form of the cost of permuting the columns of a portion of the input matrix at every iteration of the main loop. This cost can be mitigated by employing advanced pivoting strategies that allow for additional levels of parallelism, such as the "parallel pivoting" approach. I discuss this in Section 3.3. The rest of the major operations in `BQRRP` are all highly efficient, reaching the Level 3 BLAS performance.

`BQRRP_CPU` is designed to be storage-efficient, applying all of its subsequent operations in place (modulo comparatively tiny workspace buffers). Furthermore, the output format of both `BQRRP_CPU` and `BQRRP_GPU` is identical to that of `GEQP3`.

Whether the approach to QRCP I develop remains relevant in the long run is an open question. However, even in the worst-case scenario, I emphasize that the value of this work extends beyond delivering a modern high-performance algorithm for QRCP. It also lies in establishing a solid foundation for analyzing the performance and implementation details of future QRCP methods, as well as exposing the underlying structure of such algorithms, which can be invaluable for future developers if the scourge of slow QRCP returns.

3.0.1 Chapter outline

The rest of this section is dedicated to describing the notation that is unique to this chapter (Section 3.0.2), as well as the specialized setup of the experiments (Section 3.0.3) used throughout this chapter.

Section 3.1 then starts by formally introducing the generalized view of the BQRRP algorithmic framework in Algorithm 3. The formalism I chose emphasizes that the `BQRRP` instantiation is defined by the choices of its core subroutines. The choices for such subroutines are presented in Sections 3.1.1 to 3.1.5. The subroutine details there are aided by a series of pseudocode algorithms and CPU performance plots.

Section 3.2 presents a step-by-step guide to how `BQRRP_CPU` is constructed in RandLAPACK, with particular attention to its in-place storage feature. The discussion is aided by a detailed storage visualization in Fig. 3.4.

Section 3.3 discusses how composing a GPU version of `BQRRP` differs from constructing the previously-described CPU version, making references to Algorithm 3 and Sections 3.1.1 to 3.1.5.

Section 3.4 shows details on how each subroutine that comprises the CPU and GPU versions of `BQRRP` affects the overall performance of the algorithm, presenting runtime breakdown plots. This is done to help identify specific performance bottlenecks inside the algorithm.

Section 3.5 provides empirical investigations of the quality of pivots, returned by `BQRRP`. It shows how easy-to-compute metrics of pivot quality compare when running LAPACK's default function `GEQP3` versus when running the versions of `BQRRP`, defined previously in Section 3.2. The results are shown for input matrices that are notoriously difficult to process by QRCP.

Section 3.6 empirically investigates the reconstruction error and orthogonality loss in `BQRRP`, comparing the results to those from `GEQP3`. Note that this section was not originally included as part of [94].

Section 3.7 provides performance experiments with both GPU and CPU implementations of `BQRRP` in RandLAPACK.

Section 3.8 showcases a detailed investigation of the performance of `HQRRP`.

Section 3.9 revisits the algorithm discussed as part of Chapter 2 and investigates its performance on a newer Intel machine.

Concluding remarks are given in Section 3.10.

3.0.2 Chapter-specific definitions and notation

This section describes definitions and notation that are specific to this chapter. For the general notation rules, refer to Section 1.2.

Preliminaries. The symbol b is used to refer to the block size parameter used in a given algorithm; b is expected to be small relative to the matrix size, i.e., $b \ll \min\{m, n\}$. The symbol ℓ denotes the estimated rank of a given matrix. The symbol k denotes the block rank, or the rank of the given submatrix, $k \leq b$.

Given an iteration $i \in 0{:}\lceil n/b \rceil$ of an arbitrary algorithm that parses the given matrix in increments of block size b, I use $s = i \times b$ to denote the start of the current row/column block range; $r = \min\{m, (i+1)b\}$ denotes the (exclusive) end of the current row block range; and $c = \min\{n, (i+1)b\}$ denotes the (exclusive) end of the current column block range.

`BQRRP` (`fixed-width` font) is used when referring to *any* practical algorithm developed from the BQRRP (standard font) algorithmic framework. `BQRRP_CPU` and `BQRRP_GPU` are used to specifically denote the respective CPU and GPU versions of `BQRRP`.

`BQRRP` intended output format. A `BQRRP` algorithm intends to provide output in a format that is identical to the standard LAPACK QRCP routine, `GEQP3`. Assuming that a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ is passed as input into `GEQP3`, the output format is described as follows:

- Vector $\boldsymbol{\tau}$ of length $\min\{m, n\}$ that holds the scalar factors of the elementary reflectors.
- Modified in-place $\mathbf{M}$ with its on and above-diagonal portion occupied by an explicit upper-trapezoidal $\mathbf{R}$ of size $\min\{m, n\} \times n$. The below-diagonal portion of output $\mathbf{M}$ stores Householder vectors $\boldsymbol{v}_i$ for $i = 0{:}\min\{m, n\}$, which, together with the vector $\boldsymbol{\tau}$, compactly represents the orthonormal matrix $\mathbf{Q}$ as a product of $\min\{m, n\}$ elementary reflectors:

$$\mathbf{Q} = \mathbf{H}_1 \mathbf{H}_2 \ldots \mathbf{H}_{\min\{m,n\}}$$
$$\mathbf{H}_i = \mathbf{I} - \tau \boldsymbol{v}_i \boldsymbol{v}_i^{\mathrm{T}}.$$

- Permutation vector $\boldsymbol{J}$ of size n such that if $\boldsymbol{J}(j) = i + 1$, then the j^{th} column of $\mathbf{M}(:, \boldsymbol{J})$ was the i^{th} column of $\mathbf{M}$ (as explained in Section 1.2).

I refer back to the above output format throughout this chapter when talking about the "intended" output format for `BQRRP`. The described format of the $\mathbf{Q}$ factor is referred to as the *implicit economical* storage format. By contrast, some of the algorithms described in this chapter use an *explicit economical* storage format, where $\mathbf{Q}$ is comprised of $\min\{m, n\}$ explicitly-defined orthogonal column vectors.

3.0.3 Chapter-specific experiments setup

Each of the further sections of this chapter is interlaced with the practical performance experiments. For that reason, I present upfront the details of the hardware and software setup of the experiments. For the experimental conventions that are general across all chapters of this dissertation, refer to Section 1.3.1.

Implementation resources. The version of the RandLAPACK software that was used in all experiments in this chapter is documented in [17]. The code for `BQRRP` can be found in `/RandLAPACK/drivers/rl_bqrrp*`. The code for constructing and dispatching the experiments can be found in `/benchmark/bench_BQRRP/*`, as well as `/test/drivers/bench_bqrrp*`. An automated script for re-running all of the experiments shown in this paper, as well as the MATLAB plotting scripts for replicating this chapter's plots, can be found in [24].

Hardware and software configuration. The detailed configurations of the machines that I used for conducting the CPU experiments are shown in Table 3.1.

The system setup used for the GPU experiments can be found in Table 3.2. Since the GPU listed in Table 3.2 has "only" 80 GB of memory, I was unable to run double-precision experiments with the input matrices of certain sizes on this GPU.

Numerical properties of test matrices. Although `BQRRP` can, of course, process the low-rank inputs with no trouble (as explained in Section 3.1), this chapter is not concerned with studying its practical behavior on such matrices. As such, I use only full-rank input matrices in any of `BQRRP`'s benchmarks. All test matrices in this chapter were generated using RandBLAS. For the performance experiments, each matrix entry is independently sampled from the standard normal distribution.

Table 3.1: Key details of the hardware and software configuration of the platforms where CPU testing was performed. Note that I am using MKL on both Intel and AMD systems, since it performs better than AOCL 5.0 on AMD hardware. This observation is discussed in Section 3.8.1.

		Intel Xeon 8462Y+ (2×)	**AMD EPYC 9734 (2×)**
Cores per socket		32	112
Maximum hardware threads		128	448
Clock Speed	**Base**	2.80 GHz	2.2 GHz
	Boost	4.10 GHz	3.0 GHz
Cache sizes per socket	**L1**	80 KB	64 KB
	L2	2 MB	1 MB
	L3	60 MB	256 MB
RAM		DDR5 1TB	
FP64 Peak Performance		5.4 TeraFLOPs	6.5 TeraFLOPs
BLAS & LAPACK		MKL 2025.0	
Compiler		GCC 13.2.0	
CMake		3.31.4	
OS		Red Hat Enterprise Linux 8.9	

Table 3.2: Key details of the hardware and software configuration of the platform where GPU testing was performed.

		NVIDIA Hopper (H100)
CUDA cores		16,896
Memory	**Type**	HBM3
	Size	80 GB
	Bandwidth	3.35 TB/s
BLAS & LAPACK		cuBLAS 12.4.5.8 & cuSOLVER 11.6.1.9
FP64 Peak Performance		60 TeraFLOPs
CUDA NVCC		12.4.1
CMake		3.27

It should be noted that such matrices are appropriate for the performance tests as being representative of "typical" user inputs, and they have predictable numerical and performance properties, and give a representative behavior in pivot column interchanges.

I use numerically challenging matrices solely when discussing the quality of algorithms' output, as discussed in Sections 3.5 and 3.6.

Sizes of test matrices. The experiments in this chapter are focused on large matrices where the performance gap between Level-2 and Level-3 BLAS-based algorithms is readily apparent. Simultaneously, I want to ensure that the experiments I set up terminate in *reasonable* time. Hence, I do not conduct runs on input matrices of the absolute largest sizes that can fit on the systems accessible to me. Once I finish analyzing the established versions of the `BQRRP` algorithm for the larger matrix sizes, I present the investigation of the performance of various methods when applied to input matrices of a wide range of sizes, using various numbers of OpenMP threads, in Section 4.2.

There are some known performance concerns when it comes to selecting the specific sizes of the input matrices. Specifically, matrices with column sizes being powers of 2 experience increased demand for the main memory bandwidth based on how the modern cache memory works: elements from nearby columns are mapped to the exact cache line because power-of-2 memory addresses have all lower bits set to 0. Note that all modern libraries use a packed storage format for an efficient in-cache matrix multiply kernel, but they still have to read and write the cached data from/to the main memory, causing false sharing of cache lines. Due to a much-simplified structure of caches, GPUs do not experience this problem, and thus, one needs to compromise on the matrix size selection for comparable results between the types of computing devices. Because of this, I chose to run the experiments on matrices with dimensions drawn from the two sets: powers of two and multiples of ten. Additionally, in my main performance experiments with various QR and QRCP algorithms, I focus on square input matrices. This choice provides a meaningful baseline for performance comparisons, while ensuring a balanced computational workload and even distribution of work across threads. Nonetheless, I do present the performance results captured on both tall and wide matrices in Section 3.7.1.

3.1 The framework

As stated in Section 3.0.2 the intended format for a `BQRRP` algorithm is to be in line with that of standard LAPACK QRCP, `GEQP3`. Algorithm 3 gives pseudocode toward this end. For clarity of presentation, it uses a *simplified* representation of orthogonal factors from QR factorizations, stating that such factors represent a *square* matrix using some number of elementary reflectors (implicit economical storage format). The pseudocode is valid in a setting where variables are passed by value rather than by reference, and where functions can return nontrivial data structures. Details on how a `BQRRP` algorithm can be properly implemented in-place and when working with raw pointers are deferred to Section 3.2.

The pseudocode in Algorithm 3 has two required inputs: the matrix $\mathbf{M}$ and an integer block size b. It relies on five essential helper functions inside its main loop. These helper functions are applied to various submatrices whose bounds are computed at step 6.

- `qrcp_wide` in step 7 represents a column-pivoted QR factorization method, suitable for wide matrices.
- `tri_rank` in step 8 computes some notion of numerical rank of an input triangular matrix.
- `col_perm` in steps 9, 10, 11 is responsible for permuting the columns of a given matrix in accordance with the indices stored in a given vector.
- `qr_tall` in step 12 performs a tall unpivoted QR factorization.
- `apply_trans_q` in steps 15 and 17 applies the transpose $\mathbf{Q}$-factor output by `qr_tall` to a given matrix.

Sections 3.1.1 to 3.1.5 describe possibilities of how each of these functions might be implemented. I discuss specific implementations targeting the CPU architectures from Table 3.1.

While `BQRRP` is a randomized algorithm, it uses randomness only once. Before entering the main loop, it generates a matrix whose entries are independent mean-zero variance-one Gaussian random variables.[†] This matrix is applied $\mathbf{M}$ to obtain

[†]The unit-variance requirement is not essential, so long as all entries are sampled from the same mean-zero Gaussian distribution.

Algorithm 3 : Blocked QR with Randomization and Pivoting

Required input: An m-by-n matrix $\mathbf{M}$; and an integer block size parameter b.

Optional input: A scalar γ that sets the size of the sketch relative to b ($\gamma \geq 1$).

Output: Numerical rank ℓ, orthonormal $m \times m$ matrix $\mathbf{Q}$ based on ℓ elementary reflectors, upper-trapezoidal $\ell \times n$ matrix $\mathbf{R}$, and a column permutation vector $\boldsymbol{J}$ of length n.

function BQRRP($\mathbf{M}, b, \gamma$)
  Set $d = \lceil \gamma \cdot b \rceil$ and sample a d-by-n sketching operator $\mathbf{S}$ from a Gaussian distribution
  Allocate empty $\mathbf{Q}$, $\mathbf{R}$; $\boldsymbol{J} = 1{:}(n+1)$
  Sketch $\mathbf{M}^{\text{sk}} = \mathbf{S}\mathbf{M}$
  for $i = 0 : \lceil n/b \rceil$ **do**
    $s = i \cdot b$ is the start of the current row/col block;
    $c = \min\{n, (i+1)b\}$ is the (exclusive) end column of the column block;
    $r = \min\{m, (i+1)b\}$ is the (exclusive) end row of the row block;
    Decompose $[\sim, \mathbf{R}^{\text{sk}}, \boldsymbol{J}^{\text{sk}}] = \texttt{qrcp_wide}(\mathbf{M}^{\text{sk}}(:, s{:}))$
    // $\boldsymbol{J}^{\text{sk}}$ is a permutation vector of length $n - s$
    Determine $k = \texttt{tri_rank}(\mathbf{R}^{\text{sk}})$
    // $k \leq \min\{b, n-s, m-s\}$
    Permute $\mathbf{R}(0{:}s, s{:}) = \texttt{col_perm}(\mathbf{R}(0{:}s, s{:}), \boldsymbol{J}^{\text{sk}})$
    // The rectangular portion of the **computed** rows is permuted (no-op at $i = 0$)
    Permute $\mathbf{M}(s{:}, s{:}) = \texttt{col_perm}(\mathbf{M}(s{:}, s{:}), \boldsymbol{J}^{\text{sk}})$
    Update $\boldsymbol{J} = \texttt{col_perm}(\boldsymbol{J}(s{:}), \boldsymbol{J}^{\text{sk}})$
    Decompose $[\mathbf{Q}^{\text{curr}}, \mathbf{R}_{11}] = \texttt{qr_tall}(\mathbf{M}(s{:}, s{:}c), k)$
    // $\mathbf{Q}^{\text{curr}}$ uses k reflectors, implicitly represents $m - s$ columns and $\mathbf{R}_{11}$ is k-by-b
    Update $\mathbf{R}(s{:}(s+k), s{:}c) = \mathbf{R}_{11}$
    if $k \neq \min\{b, (n-s), (m-s)\}$ **then**
      Perform $[\sim, \mathbf{R}_{12}] = \texttt{apply_trans_q}(\mathbf{Q}^{\text{curr}}, \mathbf{M}(s{:}k, c{:}))$
      // Output $\mathbf{R}_{12}$ is $k \times (n - c)$
    else
      Perform $[\mathbf{M}^{\text{to_update}}, \mathbf{R}_{12}] = \texttt{apply_trans_q}(\mathbf{Q}^{\text{curr}}, \mathbf{M}(s{:}, c{:}))$
      // Output $\mathbf{M}^{\text{to_update}}$ is $(m-r)$-by-$(n-c)$, $\mathbf{R}_{12}$ is b-by-$(n-c)$
      Update $\mathbf{M}(r{:}, c{:}) = \mathbf{M}^{\text{to_update}}$
    Update $\mathbf{R}(s{:}(s+k), c{:}) = \mathbf{R}_{12}$
    Update $\mathbf{Q}(s{:}r, s{:}(s+k)) = \mathbf{Q}^{\text{curr}}$
    if $i = n/b$ or $k \neq \min\{b, (n-s), (m-s)\}$ **then**
      $\ell = s + k$
      break
    Update $\mathbf{M}^{\text{sk}}(:, c{:}) = \begin{bmatrix} \mathbf{R}_{12}^{\text{sk}} - \mathbf{R}_{11}^{\text{sk}}(\mathbf{R}_{11})^{-1}\mathbf{R}_{12} \\ \mathbf{R}_{22}^{\text{sk}} \end{bmatrix}$
  return ℓ, $\mathbf{Q}$, $\mathbf{R}$, $\boldsymbol{J}$

the *sketch*. The number of rows in the sketch is $d = \lceil \gamma b \rceil$, where γ is the *sampling factor*. The natural default value of γ depends on the implementation of `qcrp_wide` (in the recommended implementation, the only reasonable value is $\gamma = 1$). As `BQRRP` iterates, the sketch is updated deterministically with a method proposed by Duersch and Gu [54, Section 4].

Remark 15. In Chapter 2, a fast sparse operator was used to prevent sketching from becoming a computational bottleneck. In the context of `BQRRP`, Gaussian sketching has no risk of being a bottleneck operation. This is because `BQRRP` sketches in the *sampling regime* rather than the *embedding regime*, to borrow terms from [101, Section 2.2].

Note that Sections 3.1.1 to 3.1.5 do not dwell too much on the edge cases of Algorithm 3 (m, n not divisible by b, small square inputs in the described pseudocodes, etc.), instead describing the default formulation of the subroutines that go into Algorithm 3 and their performance on the most relevant inputs.

3.1.1 A practical wide QRCP selection

At iteration $i \in 0{:}\lceil n/b \rceil$, step 7 in Algorithm 3 uses a `qrcp_wide` function, applied to a wide (for most i) sketched matrix $\mathbf{M}^{\text{sk}} \in \mathbb{R}^{d \times (n-ib)}$. One could implement this by calling the standard LAPACK QRCP function, `GEQP3`. In my high-performance implementation of `BQRRP`, I employ an approach that uses the LU factorization with partial pivoting, `GETRF`, to retrieve the pivot vector $\boldsymbol{J}^{\text{sk}}$ and then unpivoted QR, `GEQRF`, to find the matrix $\mathbf{R}^{\text{sk}}$ needed in the sample updating step (step 24). Algorithm 4 shows how such a method is implemented.

Step 2 in Algorithm 4 does not use an in-place transpose on purpose. Since the matrix $\mathbf{M}^{\text{sk}}$ that is to be used by the `qrcp_wide` function in step 4 of Algorithm 3 is marginally smaller than $\mathbf{M}$, allocating a $d \times n$ buffer for $\mathbf{M}^{\text{sk_trans}}$ is more efficient than performing an additional in-place transpose to restore $\mathbf{M}^{\text{sk}}$ at the end of the algorithm.

Remark 16. Note that a portion of $\mathbf{R}^{\text{sk}}$ can be used in step 12 for preconditioning the next panel of $\mathbf{M}$ (see details in Section 3.1.3).

Permutation formats. Step 4 in Algorithm 4 is crucial, since the format in which a pivoted LU represents the permutation vector $\boldsymbol{J}$ is different from the pivoted QR

format, and hence the format conversion procedure is required. In the context of pivoted LU factorization, row i of the input matrix was interchanged with row $\boldsymbol{J}^{\mathrm{lu}}(i)$. The format conversion consists of first creating a vector $\boldsymbol{J}^{\mathrm{qr}}$ of length n with entries from 1 to n and then serially swapping elements in it according to the entries in $\boldsymbol{J}^{\mathrm{lu}}$. Simply put, for element at index $i \in 0{:}n$, element at $\boldsymbol{J}^{\mathrm{qr}}(i)$ is to swap positions with the element at $\boldsymbol{J}^{\mathrm{qr}}(\boldsymbol{J}^{\mathrm{lu}}(i) - 1)$.

Remark 17. When implementing the pivot translation procedure in practice, it is important to remember that pivot vectors in LAPACK store entries in a one-based index format used by Fortran and MATLAB (as stated previously in Section 1.2).

Wide QRCP and sketching. When Algorithm 4 is in use in `BQRRP`, the first b components of $\boldsymbol{J}^{\mathrm{qr}}$ are the same for $\gamma = 1$ and $\gamma > 1$. Therefore the rank-revealing properties of `BQRRP` using Algorithm 4 for `qrcp_wide` cannot be improved by using $\gamma > 1$.

Candidates' performance. Observe a practical comparison of the performance of the two approaches in Fig. 3.1. The performance superiority of Algorithm 4 to the standard QRCP approach makes it the best option to be used in a `BQRRP` implementation.

3.1.2 Numerical rank selection

In order to understand the role of `tri_rank` at step 8, it is useful to pretend that all computations in `BQRRP` and its subroutines are performed in exact arithmetic.

If the current block is not full-rank ($k \neq \min\{b, (n-s), (m-s)\}$), then the current iteration of `BQRRP` will be the final one. To see why this is reasonable, suppose `tri_rank` returns the exact rank of its input matrix. One can show that, conditional on an event which occurs with probability 1, `BQRRP` returns $\ell = \mathrm{rank}(\mathbf{M})$ and a full decomposition of $\mathbf{M}$. If the update at Step 24 were still performed, $\mathbf{M}^{\mathrm{sk}}(:, c{:})$ would be the zero matrix.

Of course, it is unrealistic to assume black-box access to a method to compute the exact rank of a floating-point matrix. This raises the question of whether, setting aside rounding errors, a full decomposition of $\mathbf{M}$ can be ensured if I allow for overestimation of rank. The answer is that *it can be*; it is valid for `tri_rank` to simply return the dimension p of the input $p \times p$ triangular matrix. This choice can be used with

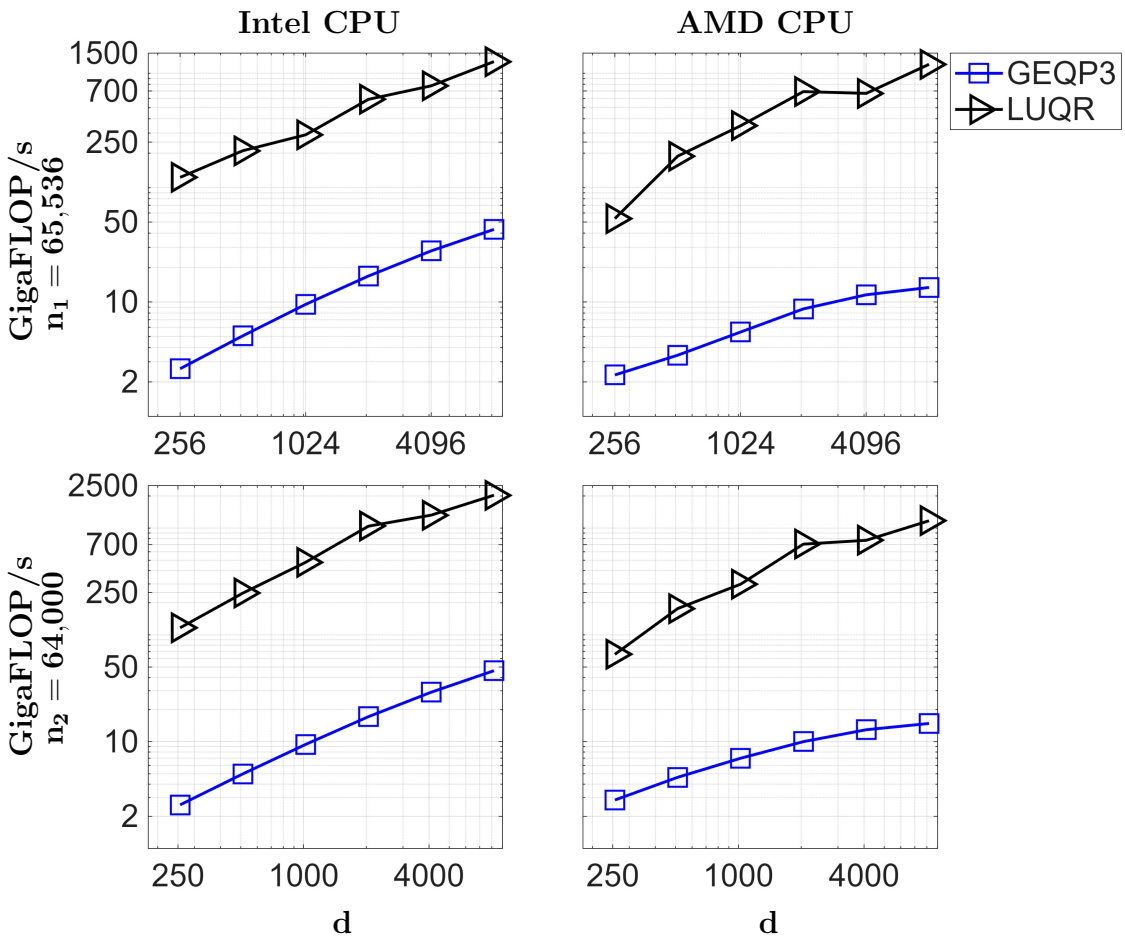


Figure 3.1: Performance of the candidate methods for `qrcp_wide` function in step 7 of Algorithm 3, captured on Intel and AMD systems (see Table 3.1). The performance is measured via the *canonical* FLOP rate, relying on the FLOP count of the standard LAPACK QR function (`GEQRF`). Experiments were conducted on matrices of size $d_1 \times n_1$ and $d_2 \times n_2$, with $n_1 = 65{,}536$ and $d_1 \in 256 \cdot \{1, 2, 4, \ldots, 32\}$, and $n_2 = 64{,}000$ and $d_2 \in 250 \cdot \{1, 2, 4, \ldots, 32\}$. On both systems, the performance of `LUQR` scheme (Algorithm 4) is superior to `GEQP3`, for all choices of d.

minor modifications to the rest of `BQRRP` and would ensure that `BQRRP` mimics `GEQP3` as closely as possible. It would be sufficient to use a different updating formula at step 24 that remains well-posed if $\mathbf{R}_{11}$ is singular; two such methods are described in [54, §4]. It would also be sufficient to keep the ill-posed update, while ensuring that `qrcp_wide` returns $\boldsymbol{J}^{\text{sk}} = (1, 2, \ldots, n - s + 1)$ when $\mathbf{M}^{\text{sk}}$ contains infs or NaNs. Since $\boldsymbol{J}^{\text{sk}}$ only affects the pivot decisions used in $\mathbf{M}$, a stable Householder QR method can safely be applied in `qr_tall`, without being impacted by $\mathbf{M}^{\text{sk}}$ becoming ill-formed.

By default, the `BQRRP` implementation in RandLAPACK uses a naive rank estimator that was previously described in Section 2.2.2, specifically, in Eq. (2.6a). This strategy is needed to deal with "bad" inputs (such as, for example, matrices consisting of all zeros), and for implementations of `qr_tall` based on Cholesky QR.

Remark 18. Furthermore, a full two-stage rank estimation approach is required for the implementations of `qr_tall` based on Cholesky QR to operate robustly, as the Householder reconstruction routine, described in Section 3.1.3, can fail if the input matrix is not orthonormal.

More sophisticated strategies would be needed if `BQRRP` were intended as a drop-in replacement for LAPACK 3.12's `GEQP3RK` function for truncated QRCP of low-rank matrices. Such strategies are beyond the scope of this work. In particular, as stated earlier in Section 3.0.3, all performance experiments I conduct are on matrices of full numerical rank.

3.1.3 Tall QR selection

Step 12 in Algorithm 3 is concerned with performing unpivoted QR on a (generally) tall submatrix of the input matrix that has been permuted via a permutation vector, computed according to the description in Section 3.1.1. Thus, any QR factorization method that can handle tall matrices is suitable here. The resulting $\mathbf{Q}$-factor must follow the format outlined in Section 3.0.2. If the selected QR method produces $\mathbf{Q}$ in a different format, it should be converted to the required representation. Additionally, at iteration $i \in 0{:}\lceil n/b \rceil$, the tall QR is to be performed on a matrix of size $(m - ib)$-by-b (except possibly in the last iteration if n is not evenly divisible by b), regardless of whether the block has full rank k (estimated with the procedure described in Section 3.1.2). Despite the fact that b reflectors would be computed in that case, only k of them would be used outside of this step.

Available methods. LAPACK offers several algorithms suitable for tall QR. The natural choice in this context is `LATSQR` [45]: a specialized Householder QR factorization for tall matrices. Alternatively, one could use a standard Householder QR factorization, `GEQRF` [42]. As the third option, `GEQRT` [43], is a blocked algorithm based on compact WY representation [27]. Finally, LAPACK's `GEQR` calls either `LATSQR` or `GEQRT`, depending on the size of the input matrix and the output from LAPACK's `ILAENV` inquiry function [44]. In addition to listing the standard LAPACK algorithms, I consider the use of *Cholesky QR* in this context, following the approach from Chapter 2. Refer to Section 2.0.2 for the background on Cholesky QR.

Remark 19. In principle, one could use a version of the Gram-Schmidt method in order to obtain an *explicit* version of the $\mathbf{Q}$-factor. While this may be a viable option in some settings, it would increase space requirements by m^2 words, and it would not meet the intended output format requirements.

Cholesky QR dependencies. Although Cholesky QR can be safe to use in the context of `BQRRP`, its use produces the following complication: steps 15 and 17 require access to the fully-formed m-by-m representation of the orthonormal factor computed at the current iteration, while Cholesky QR only outputs an *explicit* economical representation $\mathbf{Q}^{\text{chol}} \in \mathbb{R}^{m \times k}$. The full representation can be acquired by using a specialized LAPACK routine `ORHR_COL`. This routine transforms an explicit economical $\mathbf{Q}^{\text{chol}}$ output by Cholesky QR into an implicit representation via *Householder reconstruction*. The description of the basic approach for performing the Householder reconstruction can be found in [15, Algorithms 5, 6]. An advanced recursive implementation of `ORHR_COL` is described in its respective Netlib LAPACK documentation [77]; see also [68].

Algorithm 5 shows a practical implementation of Cholesky QR and its dependencies, offering a method that can be used in step 12.

Candidates' performance. Fig. 3.2 illustrates how the use of preconditioning and Householder reconstruction affect the performance of Cholesky QR, relative to alternative methods for tall QR factorization. Fig. 3.2 shows that `GEQRF` is the best alternative method for tall QR factorization across both systems. Additionally, favoring the use of Cholesky QR on the Intel system for the input matrices of select sizes can be a reasonable option.

Algorithm 4 : A practical wide QRCP

Input: a submatrix $\mathbf{M}^{\text{sk}} \in \mathbb{R}^{d\times(n-ib)}$, where $d = \lceil \gamma b \rceil \geq b$.

function QRCP_PRACTICAL($\mathbf{M}^{\text{sk}}$)
  Allocate $\mathbf{M}^{\text{sk_trans}} = \texttt{transpose}(\mathbf{M}^{\text{sk}})$
  Compute $[\sim, \sim, \boldsymbol{J}^{\text{lu}}] = \texttt{lu}(\mathbf{M}^{\text{sk_trans}})$
  // Done via standard row-pivoted LU factorization, GETRF
  Convert $\boldsymbol{J}^{\text{qr}} = \texttt{piv_transform}(\boldsymbol{J}^{\text{lu}})$
  Permute $\mathbf{M}^{\text{sk}} = \texttt{col_perm}(\mathbf{M}^{\text{sk}}, \boldsymbol{J}^{\text{qr}})$
  Compute $[\mathbf{Q}^{\text{sk}}, \mathbf{R}^{\text{sk}}] = \texttt{qr}(\mathbf{M}^{\text{sk}})$
  // Done via standard unpivoted QR factorization, GEQRF
  return $\mathbf{Q}^{\text{sk}}$, $\mathbf{R}^{\text{sk}}$, $\boldsymbol{J}^{\text{qr}}$

Algorithm 5 : Cholesky QR + dependencies in the context of `qr_tall`

Input: Iteration $i \in 0{:}\lceil n/b \rceil$ (from Algorithm 3); matrix $\mathbf{M}^{\text{perm}} \in \mathbb{R}^{(m-ib)\times b}$, where $b \ll m$; upper-triangular $\mathbf{R}^{\text{sk}} \in \mathbb{R}^{d\times(n-ib)}$ (output from step 7 in Algorithm 3), where $d = \lceil \gamma b \rceil \geq \gamma b \geq b$ when $\gamma \geq 1$; block rank $k \leq b$.

function CHOLQR_DEPS($\mathbf{M}^{\text{perm}}, \mathbf{R}^{\text{sk}}, k$)
  Truncate and precondition $\mathbf{M}^{\text{pre}} = \mathbf{M}^{\text{perm}}(:, 0{:}k)(\mathbf{R}^{\text{sk}})^{-1}$
  Decompose $[\mathbf{Q}^{\text{chol}}, \mathbf{R}^{\text{chol}}] = \texttt{cholqr}(\mathbf{M}^{\text{pre}})$
  // $\mathbf{Q}^{\text{chol}}$ is explicit $(m-ib)$-by-k; $\mathbf{R}^{\text{chol}}$ is k-by-k
  Reconstruct $[\mathbf{Q}^{\text{curr}}, \boldsymbol{D}] = \texttt{householder_reconstruct}(\mathbf{Q}^{\text{chol}})$
  // Using ORHR_COL; $\mathbf{Q}^{\text{curr}}$ uses k reflectors, represents $m-ib$; $\boldsymbol{D}$ is a sign vector
  Compute $\mathbf{R}_{11} = \mathbf{R}^{\text{chol}}\texttt{diag}(\boldsymbol{D})\mathbf{R}^{\text{sk}}(0{:}k, 0{:}b)$
  // Undoing the preconditioning; $\mathbf{R}_{11}$ is $k \times b$
  return $\mathbf{Q}^{\text{curr}}$, $\mathbf{R}_{11}$

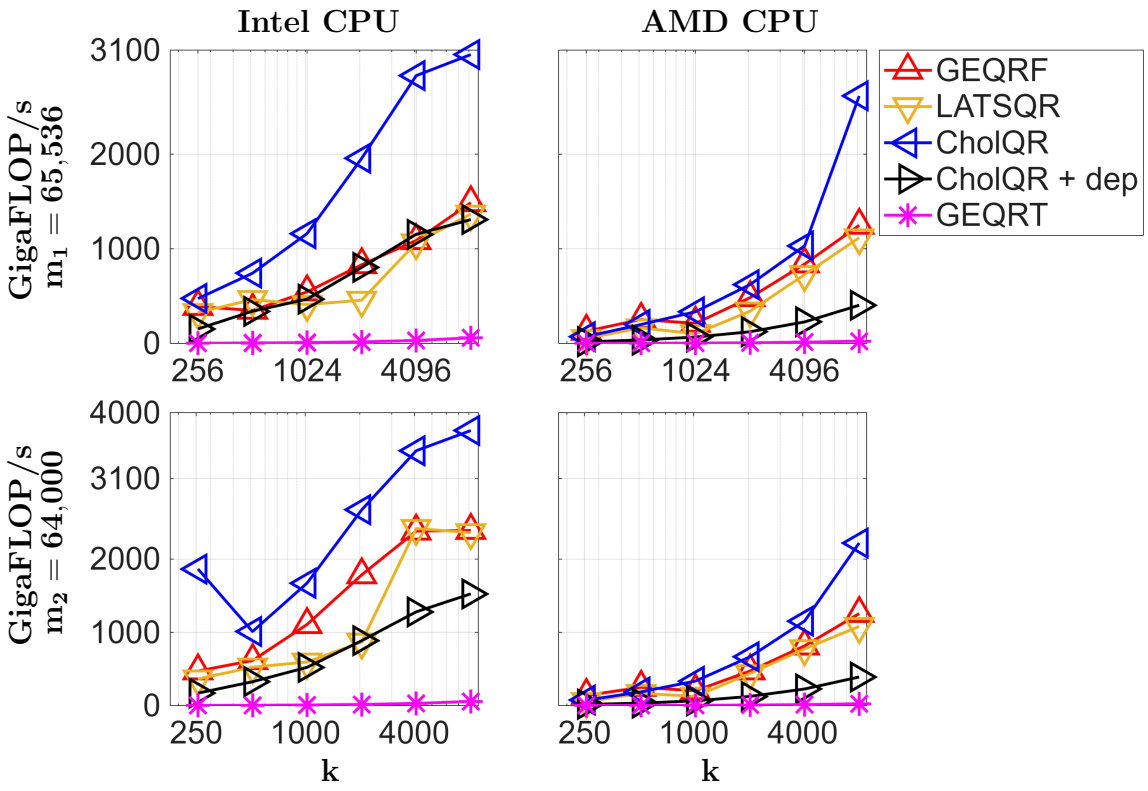


Figure 3.2: Performance of the candidate methods for `qr_tall` function in step 12 of Algorithm 3, captured on Intel and AMD systems (see Table 3.1). The performance is measured via *canonical* FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m_1 \times k_1$ and $m_2 \times k_2$, with $m_1 = 65{,}536$ and $k_1 \in 256 \cdot \{1, 2, 4, \ldots, 32\}$, and $m_2 = 64{,}000$ and $k_2 \in 250 \cdot \{1, 2, 4, \ldots, 32\}$. Across all systems, observe that Cholesky QR outperforms all alternative methods; however, the version of Cholesky QR with all of its dependencies (Algorithm 5) is only competitive on the Intel system with input data of dimensions that are powers of two. Otherwise, on both Intel and AMD systems, `GEQRF` shows the best performance out of the practical alternatives.

Remark 20. In theory, the performance of `GEQRT` should be comparable to that of `GEQRF`, as the primary distinction between the two is that `GEQRT` provides access to the $\mathbf{T}$ matrix, which encodes the sequence of upper triangular block reflectors. However, as shown in Fig. 3.2, `GEQRT` performs significantly worse than all other tall QR implementations. Varying the internal block size had no noticeable impact on performance. I suspect this is because `GEQRT` is not as heavily optimized in MKL 2025.0 as other QR routines.

Output $\mathbf{Q}$ representation. Only `GEQRF` outputs the representation of $\mathbf{Q}$ that precisely matches the intended format (described in Section 3.0.2). The format of $\mathbf{Q}$ produced by `ORHR_COL` differs from the intended one in the following way: instead of generating a vector with scalar factors of the elementary reflectors $\boldsymbol{\tau}$, it produces an upper-trapezoidal matrix $\mathbf{T}$ that represents a sequence of upper triangular block reflectors stored compactly. Each block in $\mathbf{T}$ is of size $n_b \times b$ (except possibly in the last iteration if n is not evenly divisible by b), with n_b being the block size parameter. The scalar factors of the elementary reflectors, originally intended for $\boldsymbol{\tau}$, are stored along the diagonals of the block matrix $\mathbf{T}$.

For details of the representations used by `GEQR`, `LATSQR`, and `GEQRT`, refer to the official LAPACK documentation.

3.1.4 Selection of methods for applying the Q-factor

In `BQRRP`, steps 15 and 17 represent an application of a transpose of a full representation of the current iterations's $\mathbf{Q}$ factor, $(\mathbf{Q}^{\text{curr}})^{\text{T}}$ (obtained as described in Section 3.1.3) to a pivoted trailing portion of the current iteration's matrix $\mathbf{M}$. Whether line 15 or 17 is executed depends on whether the current block rank, estimated as outlined in Section 3.1.2, equals the number of columns in the current block. The distinction between these lines lies in what is updated during the current iteration of the main `BQRRP` loop. One approach updates only a portion of the output $\mathbf{R}$-factor (in this case, the current iteration is the final one). The other approach updates both the $\mathbf{R}$-factor and the trailing portion of the input matrix, enabling the algorithm to proceed with further iterations. In an actual implementation (i.e., one where the decomposition is performed in place), no logical branching is needed to distinguish between lines 15 and 17. The difference is only a matter of how many rows of $\mathbf{M}$ are involved in the computation.

Available methods. There are two main methods for applying $(\mathbf{Q}^{\text{curr}})^{\mathrm{T}}$ to a given matrix; the choice of the right method relates to the decision made in the tall QR step.

The first available method is LAPACK's `ORMQR` function [41], which takes reflectors produced by `GEQRF` or `GEQP3`. Recall that Section 3.1.3 states that the `GEQP3`-compatible representation of $\mathbf{Q}$ must be obtained regardless of the algorithm used in the tall QR step. Therefore `ORMQR` is always an option after obtaining the `GEQP3`-compatible representation. The cost of `ORMQR` is $4nmk - 2nk^2 + 3nk$ FLOPs [7, Page 122].

The second option to use in this context is the `GEMQRT` function [40]. The input format of `GEMQRT` matches the output format of `ORHR_COL`. Since Cholesky QR must use `ORHR_COL`, this implies pairing Cholesky QR with `GEMQRT` can be an efficient choice. Alternatively, one could pair `GEMQRT` with any tall QR method, first ensuring that the representation of the output $\mathbf{Q}$-factor is made compatible with the input representation of `GEMQRT`.

One more alternative to the methods described above is avoiding the trailing update altogether, as described in [54, Algorithm 5.1]. This approach is, however, only applicable to low-rank data, and in an implementation, it would further increase the complexity of a rather nontrivial `BQRRP` scheme.

Candidates' performance. The relative performance of the alternative ways of applying an orthonormal factor $(\mathbf{Q}^{\text{curr}})^{\mathrm{T}}$ to an arbitrary matrix of size m-by-$(n-b)$ is shown in Fig. 3.3. The performance superiority of `ORMQR` function to the alternatives in the majority of the explored cases makes it the best option to be used in a `BQRRP` implementation.

Remark 21. Similar to the underperformance of `GEQRT`, which I attributed to limited optimization in MKL 2025.0 `GEMQRT` also exhibits subpar performance compared to `ORMQR`, as shown in Fig. 3.3, despite the expectation that their performance should be comparable.

3.1.5 Selection of implementation for column permutation

A `BQRRP` algorithm requires a conceptually trivial, yet crucially important kernel: a function for permuting columns of a given matrix in accordance with the pivot vector output from `qrcp_wide` at step 7. This kernel is used in Algorithm 3 at step

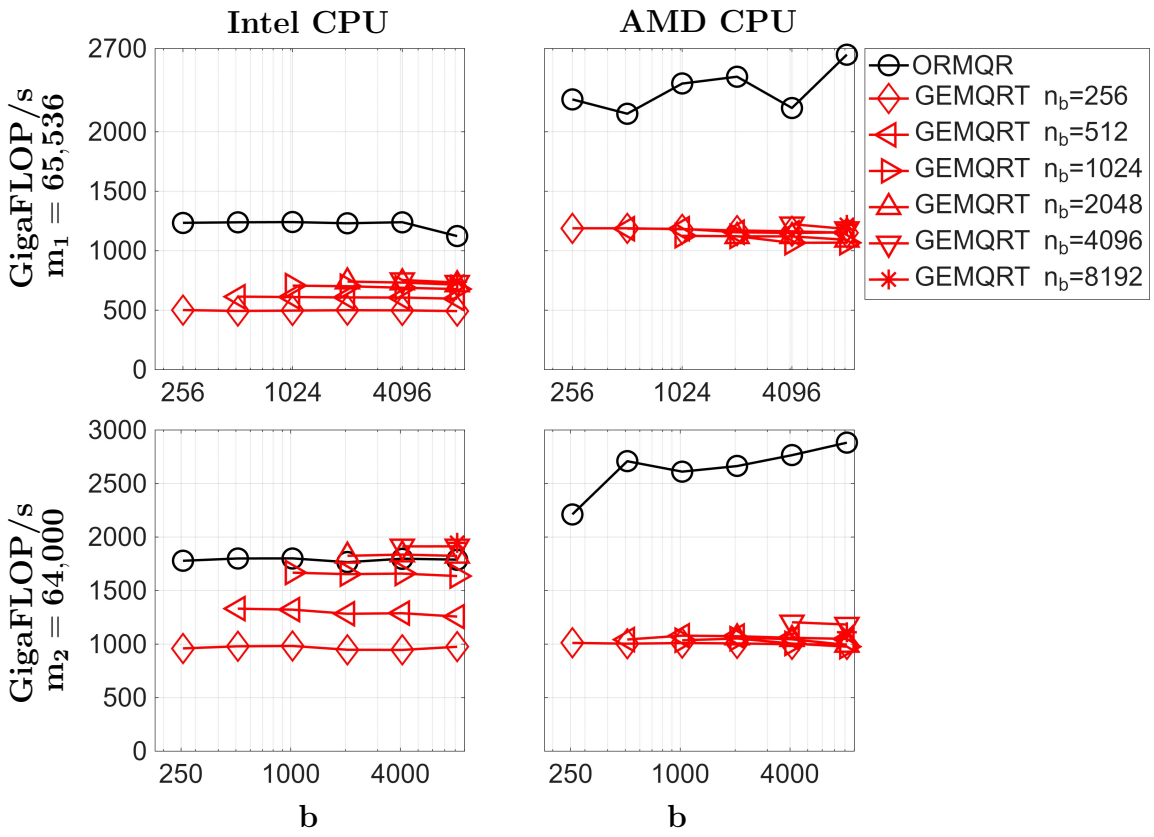


Figure 3.3: Performance of the candidate methods for `apply_trans_q` function in steps 15 and 17 of Algorithm 3, captured on Intel and AMD system (see Table 3.1). The performance is measured via *canonical* FLOP rate, relying on the FLOP count of the LAPACK function (`ORMQR`). Experiments were conducted using orthonormal matrices represented by $b_{1,2}$ column vectors of length $m_{1,2}$, applied to general matrices of size $m_{1,2}$-by-$(n_{1,2} - b_{1,2})$, where $m_1 = n_1 = 65{,}536$ and $b_1 \in 256 \cdot \{1, 2, 4, \ldots, 32\}$, and $m_2 = n_2 = 64{,}000$ and $b_2 \in 250 \cdot \{1, 2, 4, \ldots, 32\}$. As such, the conducted experiment mimics the action in step 17 in Algorithm 3 at iteration one. In most cases across both systems, the performance of `ORMQR` function is superior to that of `GEMQRT` for all choices of b and n_b.

9 for permuting the columns in the rectangular portion of the **R**-factor, at step 10 for permuting the columns of a submatrix of the input matrix, and at step 11 for updating the permutation vector.

In the context of pivoted QR factorization that generates a pivot vector $\boldsymbol{J}^{\text{qr}}$ indicating that if $\boldsymbol{J}^{\text{qr}}(j) = i + 1$, then the j^{th} column of $\mathbf{M}(:, \boldsymbol{J}^{\text{qr}})$ was the i^{th} column of $\mathbf{M}$. This representation of the pivot vector can be referred to as "permutation format." These are stored with one-based indexing for consistency with LAPACK. The approach for permuting the columns of a given matrix in accordance with $\boldsymbol{J}^{\text{qr}}$ is described in Algorithm 6.

There are two important things to note about Algorithm 6. First, as seen from step 6, the permutation vector $\boldsymbol{J}^{\text{qr}}$ is updated at every iteration of the main loop. In the context of a `BQRRP` algorithm, $\boldsymbol{J}^{\text{qr}}$ needs to be preserved after column permutation is done, and hence copying the permutation vector is required. Second, the idea behind how the permutations are performed implies that the same column can be moved several times. This prevents us from parallelizing the main loop in this algorithm (hence the keyword "`sequential`" in its name).

In principle, the sequential nature of Algorithm 6 could cause a performance bottleneck in a `BQRRP` algorithm. However, I did not initially anticipate this happening when running `BQRRP` on a CPU, as it is considered latency-tolerant hardware with a very low level of parallelism required to saturate the available memory bandwidth. The column permutation is inherently a data-intensive operation with no floating-point computation involved, and thus its main hardware bottleneck is the maximum rate at which the matrix elements can be transferred between their main memory locations. Historically, CPUs continue to suffer from decreasing bandwidth per compute cores, and thus, very few of them are needed to saturate the local memory controller in a socket. Specialized vector load and store instructions inside the `swap` function (and other Level 1 BLAS) allow one to take advantage of all memory channels, even with the majority of the cores remaining idle. The remaining issue may stem from the sequential nature of the loop in Algorithm 6. However, modern CPUs feature latency-hiding mechanisms like out-of-order execution, register renaming across an extended shadow register file, branch prediction, and data prefetching. These hardware resources allow for multiple iterations of the loop to be unrolled at runtime onto the CPU units, waiting for the moment when the `swap` operation finishes. If the non-temporal data moving instructions are used for swapping, then

there is no interference between all levels of cache memory and the matrix column elements that have to be moved through only a few vector registers. In summary, the utilization of the CPU components is evenly distributed, and the only bottleneck is the main memory bandwidth during column swapping.

3.2 Practical implementation of BQRRP and storage management

The ideas from Section 3.1 can be consolidated in the form of the two visions for BQRRP algorithms: one relying on Cholesky QR (represented by Algorithm 5 and **Q** matrix storage format conversion); and another relying on Householder QR in step 12. In both cases, ORMQR is the function of choice in steps 15 and 17, due to its reliable performance and straightforward input format requirements. Both approaches would rely on Algorithm 4 due to its unarguably superior efficiency compared to GEQP3.

From the storage standpoint, both approaches preserve the central feature of BQRRP_CPU, namely, the fact that it can be implemented in a way that all the major computations take place in the space occupied by the given input matrix (disregarding the use of some relatively small workspace buffers). To explain how this can be achieved, this section provides a step-by-step breakdown of how each operation in the two versions of BQRRP_CPU is implemented and how the data in each operation is stored.

The rest of this section is split into subsections that correspond to specific steps (or collections of steps) in Algorithm 3. I also make references to the previous sections (Sections 3.1.1 to 3.1.4). I emphasize that my description matches the way BQRRP is implemented as part of RandLAPACK. The aforementioned two versions of BQRRP_CPU are implicitly defined through the choices that the user makes when configuring the algorithm. To aid the upcoming detailed description, I present in Fig. 3.4 the visualization of how all the major components of BQRRP are stored at iteration i.

3.2.1 Input and output specification

This section is concerned with steps 1 and 25 in Algorithm 3.

Algorithm 6 : Sequential approach to column permutation

Input: A matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, a pivot vector $\boldsymbol{J}^{\text{qr}}$ of length n produced by a black-box `qrcp` function.

1: **function** COL_PERM_SEQUENTIAL($\mathbf{M}, \boldsymbol{J}^{\text{qr}}$)
2: **for** $i = 0 : n$ **do**
3: $j = \boldsymbol{J}^{\text{qr}}(i) - 1$
4: $\texttt{swap}(\mathbf{M}(:, i), \mathbf{M}(:, j))$
// Swap entirety of two columns in $\mathbf{M}$
5: $\texttt{idx} = \texttt{find}(\boldsymbol{J}^{\text{qr}}, i + 1)$
// Find the index of an element with value i
6: $\boldsymbol{J}^{\text{qr}}(\texttt{idx}) = j + 1$
7: **return** $\mathbf{M}$

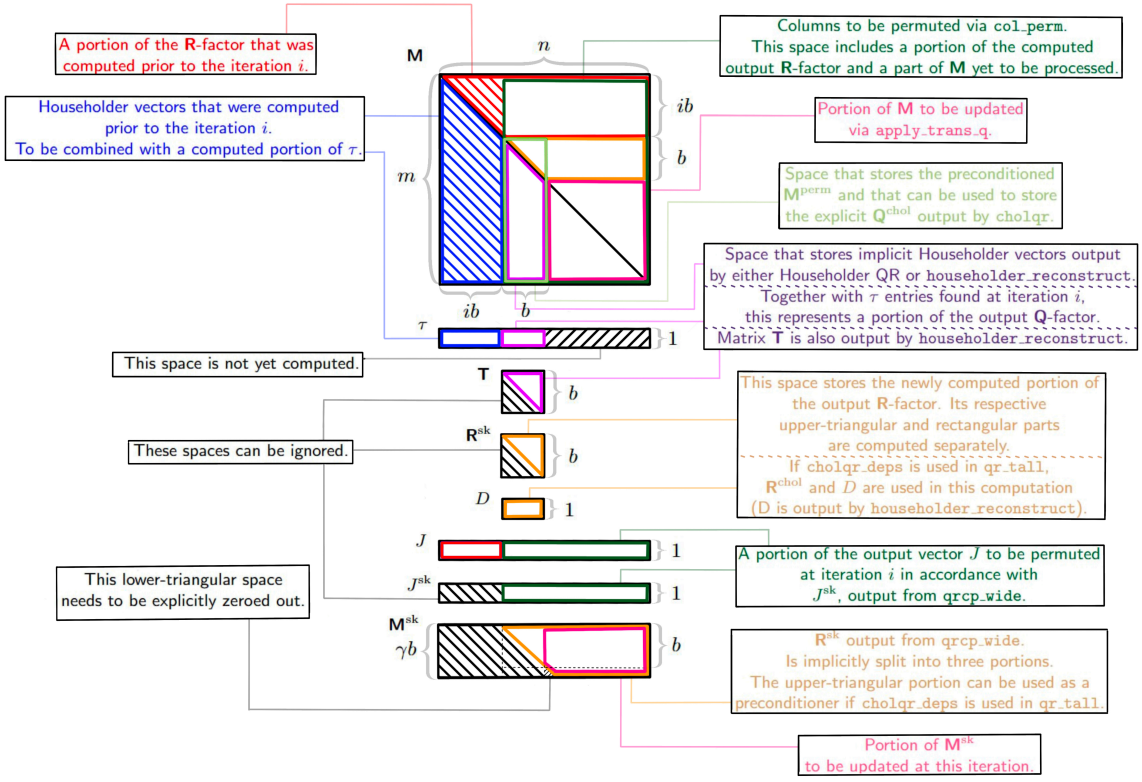


Figure 3.4: A visualization of how all the major components of `BQRRP` are stored at iteration $i \in 0{:}\lceil n/b \rceil$ of its main loop. Refer to pseudocode Algorithm 3 for the parameter and buffer names. `BQRRP` would require a maximum of $dm + 2dn + 2b^2 + 4n + b$ additional words of memory for the internal workspace buffers. This includes buffers for $\mathbf{M}^{\text{sk_trans}}$ (size $n \times d$, $d = \gamma b$), $\boldsymbol{D}$ (size b) and $\boldsymbol{J}^{\text{lu}}$ (of length n), not depicted in the figure. These storage costs are modest, if not negligible, considering the relative sizes of parameters used in practical settings: $b \leq d \ll n$.

On input, BQRRP has access to the following: a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, with no specific requirements on how m and n relate, two empty buffers of length n to store the permutation vector $\boldsymbol{J}$ and a vector with scalar factors of the elementary reflectors $\boldsymbol{\tau}$, an integer block size parameter $b \ll n$, and a scalar γ that sets the sketch size relative to b (if using LU-based QRCP on the sketch $\mathbf{M}^{\text{sk}}$, $\gamma = 1.0$). Additionally, BQRRP receives instructions on which internal subroutine decisions to take; in this case, the choice is to use Householder QR or Cholesky QR in step 12. If Cholesky QR is chosen, the user can supply an optional integer parameter n_b, defining the number of column block reflectors to be used in ORHR_COL.

On output, BQRRP returns the following: the inferred numerical rank ℓ; the $\mathbf{Q}$-factor represented by ℓ Householder reflectors that are stored implicitly and occupy the portion of the space below the diagonal (that initially contained the input matrix $\mathbf{M}$) together with a vector $\boldsymbol{\tau}$ of length ℓ; an upper-trapezoidal $\mathbf{R}$ factor sized $\ell \times n$ and stored in the upper-triangular portion of $\mathbf{M}$'s space; and a permutation vector $\boldsymbol{J}$ of length n.

3.2.2 Initial sketching details

This section is concerned with steps 2-4 in Algorithm 3.

At step 2 of Algorithm 3, a sketching operator $\mathbf{S} \in \mathbb{R}^{d \times m}$ (where $d = \lceil \gamma b \rceil$) is constructed with independent mean-zero variance-one Gaussian random variables as its entries.[‡] The generation of this sketching operator is governed by RandBLAS; the memory required to store $\mathbf{S}$ will be automatically allocated upon the operator construction and deallocated when the operator application function returns. In the in-place implementation of BQRRP, step 3 is, naturally, not explicitly performed. At step 4, a sketch resulting from applying an operator $\mathbf{S}$ to the input $\mathbf{M}$ is stored in a $d \times n$ buffer $\mathbf{M}^{\text{sk}}$. A portion of this space will further be used to store an updated sketch at every iteration of BQRRP's main loop.

3.2.3 Block partitions in BQRRP

This section is concerned with steps 5 and 6 in Algorithm 3.

[‡]Variance $1/d$ could also be used. This would have the effect of making the expected squared norm of each column in $\mathbf{SM}$ match the squared norm of the corresponding column in $\mathbf{M}$.

This section, and its corresponding steps in Algorithm 3, re-emphasize the notation that was outlined in Section 3.0.2. In Sections 3.2.4-3.2.7, I refer to the current iteration of the Algorithm 3 main loop as $i \in 0{:}\lceil n/b \rceil$. I use $s = i \times b$ to denote the start of the current row/column block range, $r = \min\{m, (i+1)b\}$ to denote the (exclusive) end of the current row block range, and $c = \min\{n, (i+1)b\}$ to denote the (exclusive) end of the current column block range (the same notation used in step 6 of Algorithm 3). This ensures that the final iteration is well-defined even if neither m or n are evenly divisible by b.

3.2.4 Processing the sketch and column permutation

This section is concerned with steps 7-11 in Algorithm 3.

QRCP on the sketch. At step 7, performing QRCP on the sketch as described in Algorithm 4 requires an $n \times d$ buffer for transposing $\mathbf{M}^{\text{sk}}$ (the full buffer size is only needed at $i = 0$). The important data computed in this step at the iteration i is stored as follows: the upper-trapezoidal $\mathbf{R}^{\text{sk}} \in \mathbb{R}^{d \times (n-s)}$ is stored at $\mathbf{M}^{\text{sk}}(:, s{:})$; and a vector $\boldsymbol{J}^{\text{sk}} \in \mathbb{R}^{(n-s)}$ is stored in a buffer of length n. In the further elaboration, $\mathbf{R}^{\text{sk}}$ is partitioned into three components: upper-triangular $\mathbf{R}^{\text{sk}}_{11} = \mathbf{M}^{\text{sk}}(0{:}b, s{:}c)$; rectangular $\mathbf{R}^{\text{sk}}_{12} = \mathbf{M}^{\text{sk}}(0{:}b, c{:})$; and an upper-trapezoidal $\mathbf{R}^{\text{sk}}_{22} = \mathbf{M}^{\text{sk}}(b{:}, c{:})$.

My implementation of `BQRRP` allows for alternatively performing QRCP via `GEQP3`, in which case a buffer for transposing $\mathbf{M}^{\text{sk}}$ is not needed.

Block rank computation. In step 8, if the block rank k (computed as described in Section 3.1.2) is not equal to $\min\{b, (n-s), (m-s)\}$ (when the current block is not full rank), then the given iteration would be the final one, as the termination criteria at step 21 states. In that case, the trailing portion of the matrix $\mathbf{M}$ is not updated at step 18.

If `BQRRP` is set to use Cholesky QR in step 12, then a rank estimate is employed to ensure no infinite or not-a-number values appear when the preconditioning is performed.

Combining column permutations. Since the intended output format of `BQRRP` matches that of `GEQP3`, the output $\mathbf{R}$ factor will be stored in the upper-triangular portion of $\mathbf{M}$'s memory space. At iteration $i \geq 1$, step 9 permutes the trailing

columns in the *computed rectangular* portion of the $\mathbf{R}$-factor, implying that $\mathbf{M}(0{:}s, s{:})$ is to be permuted in accordance with $\boldsymbol{J}^{\text{sk}}$. Meanwhile, step 10 requires $\mathbf{M}(s{:}, s{:})$ to be permuted by the same vector. Therefore, steps 9 and 10 can be combined into permuting $\mathbf{M}(:, s{:})$ at every iteration of the `BQRRP`'s main loop.

Remark 22. If Algorithm 6 is used for permuting columns, a copy of $\boldsymbol{J}^{\text{sk}}$ is required.

After the columns of $\mathbf{M}(:, s{:})$ have been permuted, an early termination check can be performed by verifying whether the first column in $\mathbf{M}(:, s{:})$ consists of all zeros. In that case, `BQRRP` terminates immediately. If at iteration i, $k \neq \min\{b, n-s, m-s\}$, then `BQRRP` should avoid permuting $\mathbf{M}(s+k{:}, c{:})$, since this trailing portion of the input matrix would not be used (i.e., the trailing update to $\mathbf{M}$ would be avoided in step 15). This was not shown in Algorithm 3 for simplicity of presentation.

Updating the permutation vector. By step 11, the pivot vector $\boldsymbol{J}^{\text{sk}}$, computed at the current iteration, has been used for all the necessary internal operations. It can now be incorporated into the vector $\boldsymbol{J}$ that is to be output by `BQRRP`. When i is 0, this is done by copying $\boldsymbol{J}^{\text{sk}}$ into $\boldsymbol{J}$; at any subsequent iteration, the trailing portion of $\boldsymbol{J}$, $\boldsymbol{J}(s{:})$, will be permuted in accordance with $\boldsymbol{J}^{\text{sk}}$ via (a vector version of) Algorithm 6. Since $\boldsymbol{J}^{\text{sk}}$ is not needed after this step, one does not have to create a copy of it in Algorithm 6.

3.2.5 Panel QR factorizations details

This section is concerned with steps 12 and 13 in Algorithm 3.

The internal details of step 12 depend on the user's choice of tall QR function. As said before, in my implementation of `BQRRP`, I allow users to choose between `GEQRF` and the Cholesky QR-based approach.

Householder QR on a panel. When using `GEQRF` in step 12, the factorization is performed on all columns of $\mathbf{M}^{\text{perm}} = \mathbf{M}(s{:}, s{:}c)$, regardless of whether k, the current column block rank, is equal to $k_{\max} = \min\{b, (n-s), (m-s)\}$. This is because $\mathbf{R}_{11}$ needs to have $(c-s) = \min\{b, (n-s)\}$ columns. Despite the fact that $(c-s)$ reflectors are computed in that case, only k of them are used *outside* of this step. Using `GEQRF` on $\mathbf{M}^{\text{perm}}$ in this step requires no additional storage. This function produces $\mathbf{Q}^{\text{curr}}$, represented by $(c-s)$ reflectors of length $(m-s)$ (stored in the portion of $\mathbf{M}(s{:}, s{:}c)$

below the diagonal) and the vector of scalar factors of the elementary reflectors (stored in $\boldsymbol{\tau}(s{:}c)$). The explicit $\mathbf{R}_{11}$ of size $(r-s)\times(c-s)$ is stored right above the $\mathbf{Q}^{\text{curr}}$, in the upper-triangular portion of $\mathbf{M}(s{:}r, s{:}c)$. Note that in this case, step 13 is performed implicitly, since $\mathbf{R}_{11}$ will be stored where it needs to be on output from `GEQRF`.

Cholesky QR on a panel. Using the Cholesky QR approach in step 12 is comprised of four parts: preconditioning; performing Cholesky QR on the preconditioned matrix; implicit $\mathbf{Q}^{\text{curr}}$ reconstruction; and computing $\mathbf{R}_{11}$. All of these are done in accordance with Algorithm 5.

The preconditioning is to be done on a portion of the matrix $\mathbf{M}^{\text{perm}}$. I perform $\mathbf{M}^{\text{pre}} = \mathbf{M}^{\text{perm}}(:, 0{:}k)(\mathbf{R}^{\text{sk}}_{11}(0{:}k, 0{:}k))^{-1}$, using $\mathbf{R}^{\text{sk}}_{11}$ stored in $\mathbf{M}^{\text{sk}}(0{:}b, s{:}c)$. This step requires no workspace buffers, and the matrix $\mathbf{M}^{\text{pre}} \in \mathbb{R}^{(m-s)\times k}$ is stored at $\mathbf{M}(s{:}, s{:}(s+k))$ after the preconditioning. The next step is to perform Cholesky QR on $\mathbf{M}^{\text{pre}}$.

As explained in Section 3.1.3, Cholesky QR is comprised of: computing the Gram matrix (`SYRK`), performing the Cholesky factorization (`POTRF`), and forming the explicit economical version of the orthonormal factor (`TRSM`). These functions require (at most) a single $b \times b$ workspace buffer to store the Gram matrix and the output upper-triangular $\mathbf{R}^{\text{chol}}$ factor, both of size $k \times k$, $k \leq \min\{b, (n-s), (m-s)\}$. The explicit orthonormal factor $\mathbf{Q}^{\text{chol}} \in \mathbb{R}^{(m-s)\times k}$ is stored at $\mathbf{M}(s{:}, s{:}(s+k))$, in place of $\mathbf{M}^{\text{pre}}$.

`ORHR_COL` used for the Householder reconstruction returns an implicit representation of $\mathbf{Q}^{\text{curr}}$ in the form of k reflectors and an upper-trapezoidal matrix $\mathbf{T}$ of size $n_b \times k$. The k reflectors fit exactly in place of the portion of $\mathbf{Q}^{\text{chol}}$ below the diagonal of $\mathbf{M}(s{:}, s{:}(s+k))$, while $\mathbf{T}$ requires an additional buffer of size $n_b \times b$. Furthermore, `ORHR_COL` produces a sign vector $\boldsymbol{D}$ of length k that requires a storage buffer of length b.

Computing $\mathbf{R}_{11}$ requires first applying the entries from the sign vector $\boldsymbol{D}$ to the columns of $\mathbf{R}^{\text{chol}}$ (this is done so that $\mathbf{R}^{\text{chol}}$ is in line with $\mathbf{Q}^{\text{curr}}$, computed by `ORHR_COL`). The next step amounts to undoing the preconditioning on $\mathbf{R}^{\text{chol}}$ via multiplying it by an upper-triangular $\mathbf{R}^{\text{sk}}_{11}(0{:}k, 0{:}(c-s))$, stored at $\mathbf{M}^{\text{sk}}(0{:}k, s{:}c)$. The

product is temporarily stored in place of $\mathbf{R}^{\text{chol}}$ and then is copied into the upper-triangular portion of $\mathbf{M}(s{:}(s+k), s{:}c)$, placing it right above the implicitly-stored Householder reflectors (step 13).

Remark 23. This additional copy is currently unavoidable because no standard BLAS function for multiplying two triangular matrices exists (although one could implement `trtrmm`).

For the output format of `BQRRP` to match that described in Section 3.0.2, one would need to copy the entries from the block diagonals of $\mathbf{T}$ into $\boldsymbol{\tau}(s{:}(s+k))$.

3.2.6 Applying transposed Q and updating factors

This section is concerned with steps 14-20 in Algorithm 3.

Step 15 or 17 perform operations described in Section 3.1.4. Depending on whether the estimated block rank k matches the column block size b, I apply a transpose of $\mathbf{Q}^{\text{curr}}$ (from the Householder representation in $\mathbf{M}(s{:}, s{:}(s+k))$ and $\boldsymbol{\tau}(s{:}(s+k))$), to either the first k or to all $(m-s)$ rows of $\mathbf{M}(s{:}, c{:})$.

Regardless of whether step 15 or 17 is executed, the first k rows of $\mathbf{M}(s{:}(s+k), c{:})$ represents $\mathbf{R}_{12}$ (this implicitly fulfills step 19). If step 17 is executed, then the remaining rows $\mathbf{M}(r{:}, c{:})$ represent the "working submatrix" of the matrix $\mathbf{M}$ at the next iteration (this implicitly fulfills step 18). Otherwise, the trailing $\mathbf{M}$ update is avoided.

In `BQRRP`, step 20 is performed implicitly, as the matrix $\mathbf{Q}$ is constructed from the Householder vectors, stored below the main diagonal of $\mathbf{M}$, and the scalar factors of the elementary reflectors stored in $\boldsymbol{\tau}$.

3.2.7 Algorithm termination and sample update

This section is concerned with steps 21-24 in Algorithm 3.

Termination criteria. The termination criteria check (step 21) simply amounts to verifying whether the maximum number of iterations has been exceeded (i reached $\lceil n/b \rceil - 1$) or whether the block rank k does not match the column block size $\min\{b, n-s, m-s\}$. Before the termination, the rank parameter ℓ is updated to $s+k$ (step 22).

Sample update. If the maximum number of iterations has not been reached at this point, the sketch $\mathbf{M}^{\text{sk}}$ is updated at step 24. This step first computes the expression $\mathbf{R}_{11}^{\text{sk}}(\mathbf{R}_{11})^{-1}$, where $\mathbf{R}_{11}^{\text{sk}}$ is stored in the upper-triangular portion of $\mathbf{M}^{\text{sk}}(0{:}b, s{:}c)$ and $\mathbf{R}_{11}$ is stored in the upper-triangular portion of $\mathbf{M}(s{:}r, s{:}c)$. The result is written into the space of $\mathbf{R}_{11}^{\text{sk}}$, at $\mathbf{M}^{\text{sk}}(0{:}b, s{:}c)$; I explicitly zero out the entries in this space below the diagonal. Next, the full expression $\mathbf{R}_{12}^{\text{sk}} - \mathbf{R}_{11}^{\text{sk}}(\mathbf{R}_{11})^{-1}\mathbf{R}_{12}$ is computed. The matrix $\mathbf{R}_{12}^{\text{sk}}$ is stored in $\mathbf{M}^{\text{sk}}(0{:}b, c{:})$ and $\mathbf{R}_{12}$ is stored at $\mathbf{M}(s{:}r, c{:})$. The result of this expression is placed into the space of $\mathbf{R}_{12}^{\text{sk}}$, at $\mathbf{M}^{\text{sk}}(0{:}b, c{:})$. The upper-trapezoidal matrix $\mathbf{R}_{22}^{\text{sk}}$ is already stored at its intended location, in $\mathbf{M}^{\text{sk}}(b{:}(d-b), c{:})$, but one needs to make sure that the entries are zeroed out below the diagonal. After the sketch update is completed, the "working submatrix" of $\mathbf{M}^{\text{sk}}$ is implicitly located at $\mathbf{M}^{\text{sk}}(:, c{:})$.

3.3 Considerations specific to implementations targeting GPU accelerators

3.3.1 Limited LAPACK functionality

Implementing a GPU version of the `BQRRP` algorithm involves several challenges, primarily due to the limited availability of GPU variants of most LAPACK and BLAS-level functions. Specifically, NVIDIA cuSOLVER lacks support for the full range of LAPACK functions. This issue, however, is rather understandable, as the latest version of LAPACK 3.12.0 includes over 2000 functions, many of which are not widely used. Nevertheless, the lack of a wide range of LAPACK functions limits the choices for the internal subroutines in a `BQRRP_GPU`.

For the purpose of constructing `BQRRP_GPU`, the most notable function that cuSOLVER does not offer is `ORHR_COL`, which is required when using Cholesky QR in the `qr_tall` subroutine. In order to provide some investigation of Cholesky QR methods for `qr_tall`, I developed a basic CUDA implementation of [15, Algorithm 5]. Note that a significant performance improvement could be achieved by the use of the Cholesky QR approach in the `qr_tall` step in Algorithm 8 if cuSOLVER offered a recursive LU-based implementation of `ORHR_COL` similar to the one used by LAPACK. It is also worth noting that there is no standard QRCP algorithm offered in

cuSOLVER. Therefore, my `BQRRP_GPU` implementation is forced to use the LU-based Algorithm 4 for `qrcp_wide`.

3.3.2 Column permutation

A major difference between the CPU and GPU versions of `BQRRP` is the importance of a high-performance implementation of a function for permuting columns of a given matrix on the GPU in accordance with a pivot vector.

Hardware considerations. Recall that Algorithm 6 illustrated a sequential approach to permuting columns. Despite this approach being sequential, I did not anticipate it having a major negative effect on the overall performance of `BQRRP_CPU`, as described in Section 3.1.5. By contrast, on a GPU, there are very few hardware resources devoted to dealing with inherently sequential instruction streams or even tolerating the high latency of the main memory transactions. The initially presented sequential loop for pivoting is anathema to the parallel GPU hardware.

The alternative approach. The widely-used alternative solution is the "parallel pivoting" strategy, shown in Algorithm 7. By contrast to Algorithm 6, Algorithm 7 does not swap columns within the memory space of a single matrix. Instead, it copies columns from one space into another at the new location from the permutation vector, which allows for parallelizing the main loop, significantly increasing the performance of this column permutation approach. The performance gain offered by Algorithm 7 comes at the cost of increased space usage, as it requires a copy of the input $\mathbf{M}$.

Managing data copies in Algorithm 7. In order to minimize the number of explicit copies performed as part of using Algorithm 7 in practice, one could swap the pointers that point to the memory spaces of $\mathbf{M}$ and $\mathbf{M}^{\text{cpy}}$ before performing column permutation. This approach is safe as long as either the *entire* matrix $\mathbf{M}$ is permuted in Algorithm 7, or the non-permuted part of $\mathbf{M}$ is not used outside of Algorithm 7. As stated in Section 3.1.5, the column permutation is used in steps 9, 10, and 11 in Algorithm 3 (where steps 9 and 10 can be merged) and in step 5 of Algorithm 4. As such, permutations are to be applied to the portions of matrices $\mathbf{M}^{\text{sk}}$ and $\mathbf{M}$, as well as a vector $\boldsymbol{J}^{\text{sk}}$ at every iteration of `BQRRP_GPU`.

Remark 24. To avoid overcomplicating the notation, the description below does not account for cases where n is not evenly divisible by b.

Since the full $\mathbf{M}^{\text{sk}}$ is not used outside of `BQRRP_GPU`, the pointer-swapping strategy can be used without any memory safety concerns. This is, however, not the case with $\mathbf{M}$ and $\boldsymbol{J}^{\text{sk}}$, as both of their entire memory spaces will have to store correct data on output from the `BQRRP_GPU`. To illustrate the complication with the pointer-swapping strategy, suppose the main loop in `BQRRP_GPU` is executing starting with $i = 0$, where index 0 is considered "even." Then, at the end of the 0^{th} iteration, the space of $\mathbf{M}^{\text{cpy}}$, will contain the correct data to be a part of the output of `BQRRP_GPU` since the pointer swapping took place. At iteration 1, the space $\mathbf{M}^{\text{cpy}}$ will contain the first b properly-computed columns, the rest of the correct entries will be contained in $\mathbf{M}(:, b{:})$. Continuing with that logic, one may conclude that the space $\mathbf{M}^{\text{cpy}}$ will contain the "correct" entries in $\mathbf{M}^{\text{cpy}}(:, ib{:}(i+1)b)$ for all even $i \leq n/b$, and the space $\mathbf{M}$ will contain the correct entries for all odd i in $\mathbf{M}(:, ib{:}(i+1)b)$ when `BQRRP_GPU` terminates. If `BQRRP_GPU` terminated at an even iteration, the pointers to $\mathbf{M}$ and $\mathbf{M}^{\text{cpy}}$ will need to be swapped back around. Regardless of the final parity status of the loop's index, the "correct" columns will need to be copied from $\mathbf{M}^{\text{cpy}}$ to $\mathbf{M}$. Note that if `BQRRP_GPU` was to terminate early at an even iteration, the trailing entries in range $(i+1)b$ to m in $\mathbf{M}^{\text{cpy}}$ will need to be copied into $\mathbf{M}$.

As per the vector $\boldsymbol{J}$, since it is not permuted at iteration 0, the space of $\boldsymbol{J}^{\text{cpy}}$ would contain the "correct" entries in $\boldsymbol{J}^{\text{cpy}}(ib{:}(i+1)b)$ for all odd $i \leq n/b$, while the space $\boldsymbol{J}$ will contain the correct entries for all even i in $\boldsymbol{J}(ib{:}(i+1)b)$ when `BQRRP_GPU` terminates. Otherwise, the vector $\boldsymbol{J}$ is to be processed similarly to the matrix $\mathbf{M}$.

3.3.3 The view of a practical `BQRRP_GPU`

Considering the constraints of designing a GPU version of `BQRRP` described in Sections 3.3.1 and 3.3.2, I adjust the view of the `BQRRP_GPU` algorithms from the two implicit CPU versions of `BQRRP` described at the beginning of Section 3.2.

All in all, I stick to the vision of `BQRRP_GPU` being represented implicitly by the two algorithms (defined by the user choices during algorithm configuration), where one version relies on Cholesky QR and its dependencies, and the other version relies on Householder QR in step 12. Cholesky QR is paired with the *sequential* approach to `ORHR_COL`, as I decided that implementing a high-performance GPU version of `ORHR_COL` is beyond the scope of this work. In steps 15 and 17, I use cuSOLVER's

ORMQR function. Furthermore, I decided to sacrifice storage for performance by using Algorithm 7 (and the pointer-swapping logic) for column permutation in my implementation of BQRRP_GPU.

At the time of writing, the current version 1.0.1 of RandBLAS does not offer GPU support. Because of that, steps 2 and 4 are performed outside of BQRRP_GPU, and $\mathbf{M}^{\text{sk}}$ is provided as an input into the algorithm.

3.4 Performance profiling of major computational kernels

Having established how the two views of BQRRP_CPU and BQRRP_GPU are constructed, in Sections 3.2 and 3.3, it is important to analyze how different parts of these algorithms affect the overall algorithm performance, in order to identify the bottlenecks for future optimization. In this section, I omit depicting the results for matrices with dimensions that are multiples of ten, as their performance profiling plots are nearly identical to those for matrices with dimensions that are powers of two.

CPU performance breakdown. I first present in Fig. 3.5 the algorithm runtime breakdown results for the two CPU versions of BQRRP. The figure depicts the percentage of runtime that is occupied by a given component kernel of the two versions of BQRRP_CPU on the y-axis. The experiments were conducted on square test matrices with 65,536 rows and columns and the block size parameter b varying as powers of two from 256 to 2048 (x-axis).

An immediate observation from Fig. 3.5 is that apply_trans_q function is the major bottleneck in the three out of four BQRRP formations under consideration, despite the fact that I used the best available function for this step (per the results in Section 3.1.4). Observe also that col_perm_sequential and ORHR_COL are proportionally much slower on the AMD system than the Intel system. It is plausible that the slowdown in the former function is more visible in the AMD system, given its up to 448 threads available. Moreover, the only parallelism within col_perm_sequential stems from the BLAS SWAP function. Since MKL is used on the AMD system (as noted in Section 3.0.3), the underlying BLAS functions may not be fully optimized for AMD hardware, which could result in suboptimal performance.

Algorithm 7 : Column-parallel approach to column permutation

Input: A matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, a pivot vector $\boldsymbol{J}^{\text{qr}}$ output by a black-box `qrcp` function.

1: **function** COL_PERM_PARALLEL($\mathbf{M}, \boldsymbol{J}^{\text{qr}}$)
2: $\quad \mathbf{M}^{\text{cpy}} = \texttt{copy}(\mathbf{M})$
3: $\quad$ **for** $i = 0 : n$ **do**
4: $\quad\quad j = \boldsymbol{J}^{\text{qr}}(i)$
5: $\quad\quad \mathbf{M}(:, i) = \mathbf{M}^{\text{cpy}}(:, j)$
6: **return** $\mathbf{M}$

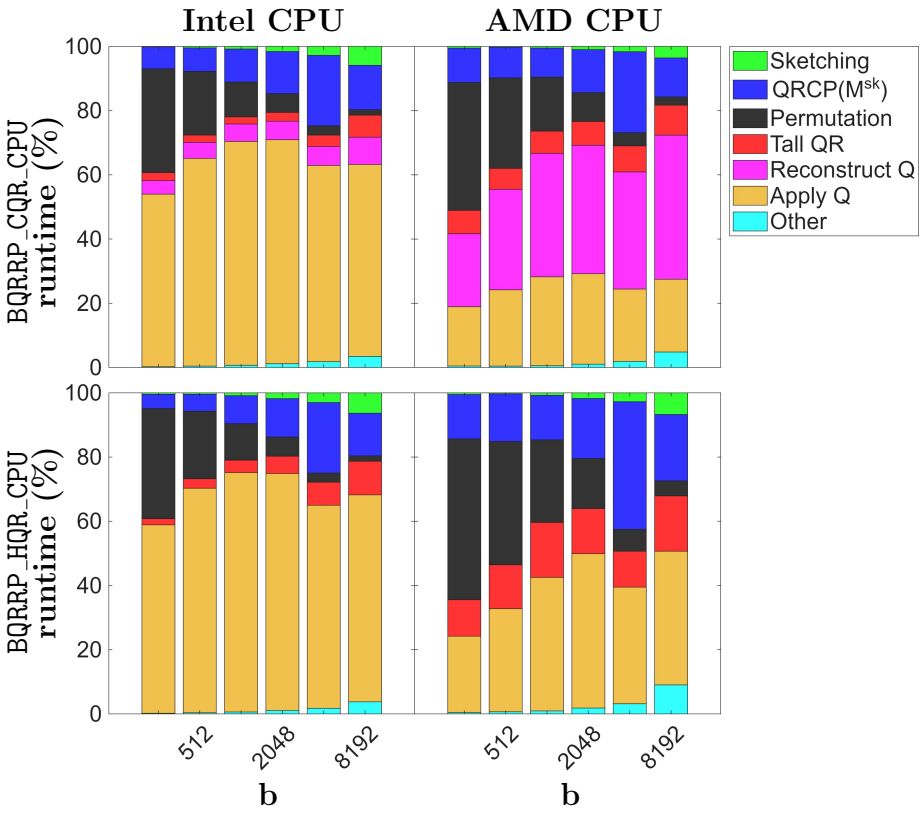


Figure 3.5: Percentages of `BQRRP_CPU` runtime, occupied by its respective subroutines. The top row represents `BQRRP_CPU` with Cholesky QR on a panel, and separately shows the percentage of runtime occupied by the preconditioned Cholesky QR and Householder restoration. The bottom row represents `BQRRP_CPU` with Householder QR on a panel. The results are captured on an Intel CPU (left) and an AMD CPU (right) (see Table 3.1), with matrices of size $m \times m$ and `BQRRP` block size b, where $m = 65{,}536$ and $b \in \{256, 512, 1024, 2048, 4096, 8192\}$. Observe that in all plots, `apply_trans_q` (Section 3.1.4) is among the most costly subroutines.

I do not have an explanation for the slow performance of `ORHR_COL` on the AMD system.

Remark 25. The results in Fig. 3.5 raise the question: could one have predicted this plot based solely on the operation counts of the core subroutines in `BQRRP`? The short answer is *no*, as evidenced by the expense of column permutation, which requires a total of $O(mn)$ operations across the entire algorithm. Furthermore, since `BQRRP` is a blocked algorithm, the operation count for each subroutine varies across iterations; adding these operation counts would lead to complex expressions that obscure more than they clarify.

GPU performance breakdown. In Fig. 3.6, I present the runtime breakdown of the two implicit versions of `BQRRP_GPU` to see any bottlenecks in their respective subroutines. Any given implementation of `BQRRP_GPU` uses a single GPU stream, and hence timing its computational kernels does not involve any complications. The figure depicts the percentage of runtime that is occupied by a given subcomponent of `BQRRP_GPU` on the y-axis. The experiments were conducted on square test matrices with 32,768 rows and columns and the block size parameter b varying as powers of two from 32 to 2048 (x-axis).

The glaring issue with the top plot can be seen as the block size increases: the fact that my simple implementation of `ORHR_COL` dominates the runtime. This suggests that the simplest approach to `ORHR_COL` is simply not viable in practice. As such, the version of `BQRRP_GPU` with Cholesky QR on a panel is expected to have worse overall performance than that with Householder QR. As can be seen from the bottom plot in Fig. 3.6, whenever a naive implementation of `ORHR_COL` is not an issue, the main algorithm bottleneck is `ORMQR`, just like in the CPU version of `BQRRP`. Both plots in Fig. 3.6 show that when smaller block sizes are in use, permuting columns in the matrix $\mathbf{M}$ is rather costly.

3.5 Pivot quality on the Kahan matrix

This section gives experimental comparisons of pivot quality using LAPACK's default QRCP subroutine `GEQP3`, compared to those produced by `BQRRP`, configured to use Algorithm 4 in Step 7. For a QR factorization with column pivoting, the pivot quality

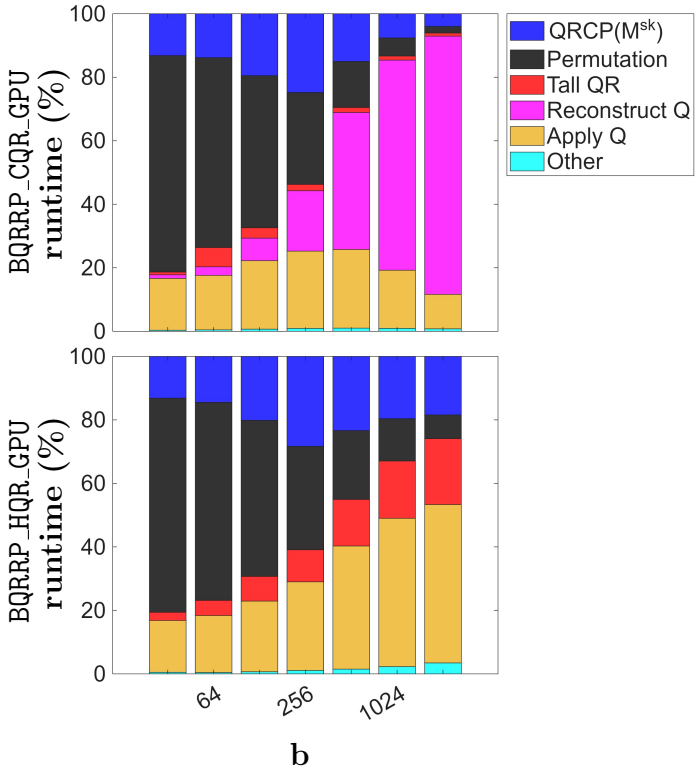


Figure 3.6: Percentages of `BQRRP_GPU` runtime, occupied by its respective subroutines. The top row represents `BQRRP_GPU` with Cholesky QR on a panel, and separately shows the percentage of runtime occupied by the preconditioned Cholesky QR and Householder restoration. The bottom row represents `BQRRP_GPU` with Householder QR on a panel. The results are captured on an NVIDIA H100 GPU (see Table 3.2), with matrices of size $m \times m$ and `BQRRP` block size b, where $m = 32{,}768$ and $b \in \{32, 64, 128, 256, 512, 1024, 2048\}$. In the algorithm with Cholesky QR on a panel, observe that my naive approach for `ORHR_COL_GPU` appears to be too costly. In the algorithm with Householder QR on a panel, observe that the main bottleneck is the `apply_trans_q` function, similar to the CPU results.

directly coincides with the *reconstruction quality* of the factors that a given algorithm produces. I use the pivot quality metrics described in Section 2.3.

The quality of pivots produced by `BQRRP` naturally depends on the type of matrix used as input. Rather than exhaustively testing `BQRRP` with various commonly used matrices in QRCP verification schemes, I focus directly on a challenging case: the Kahan matrix, which is notoriously difficult for QRCP to handle. The Kahan matrix is known for having small differences in column norms, hence potentially causing pivoting to fail, which results in inaccurate factorizations. From the description in [98], one can parameterize the Kahan matrix by n, p, θ, by taking $\alpha = \sin(\theta)$, $\beta = -\cos(\theta)$, and

$$\mathbf{M} = \underbrace{\begin{bmatrix} 1 & & & \\ & \alpha^1 & & \\ & & \ddots & \\ & & & \alpha^{n-1} \end{bmatrix}}_{\text{diagonal}} \underbrace{\begin{bmatrix} \beta & 1 & \cdots & 1 \\ & \beta & \ddots & \vdots \\ & & \ddots & 1 \\ & & & \beta \end{bmatrix}}_{\text{upper-triangular}} + \epsilon_{\text{mach}} \cdot p \underbrace{\begin{bmatrix} n & & & \\ & n-1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}}_{\text{diagonal}} \tag{3.1}$$

Fig. 3.7 shows the spectrum of a Kahan matrix (with default choices for the values of the tuning parameters), obtained by the Jacobi SVD function, `GESVD`. Fig. 3.8 shows `BQRRP` pivot quality compared against `GEQP3` pivot quality in the two aforementioned metrics. The experiment was conducted using two different `BQRRP` block sizes.

3.6 `BQRRP` output quality

This section mirrors the content of Section 2.4 for `BQRRP` and shares the same underlying sentiment of providing rigorous empirical results on the quality of output for the developed.

I present Table 3.3 that reports the relative reconstruction error and orthogonality loss across `GEQP3` and `BQRRP` algorithm runs on matrices with low and high coherence, featured previously in Section 2.3.1 and Section 2.3.2. The matrices are of dimensions $m \times n$, with $m = n\{16{,}000, 64{,}000, 128{,}000\}$; I use `BQRRP` block size $b = \{250, 8{,}000\}$. I showcase the average error attained by `BQRRP` across 10 runs for each input size and type, along with the error ranges observed; the results for `GEQP3` are only captured once. Remarks made in Section 2.4 apply here as well.

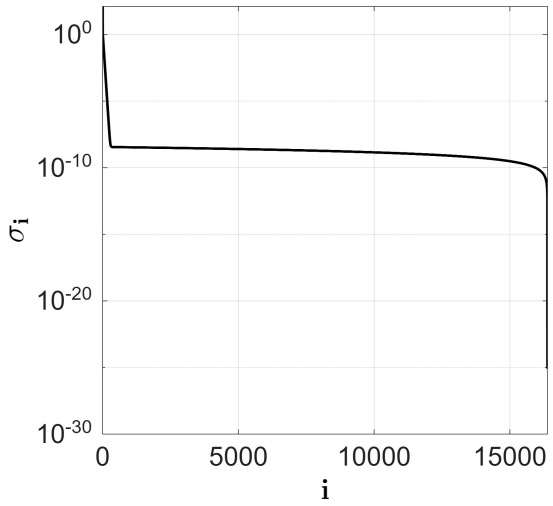


Figure 3.7: Spectrum of Kahan matrix of order 16,384, generated as described in Eq. (3.1), using $p = 1000$ and $\theta = 1.2$. The spectrum was obtained using LAPACK's most accurate SVD function, `GESVD`. Note that the trailing singular values fall below the double-precision machine ϵ.

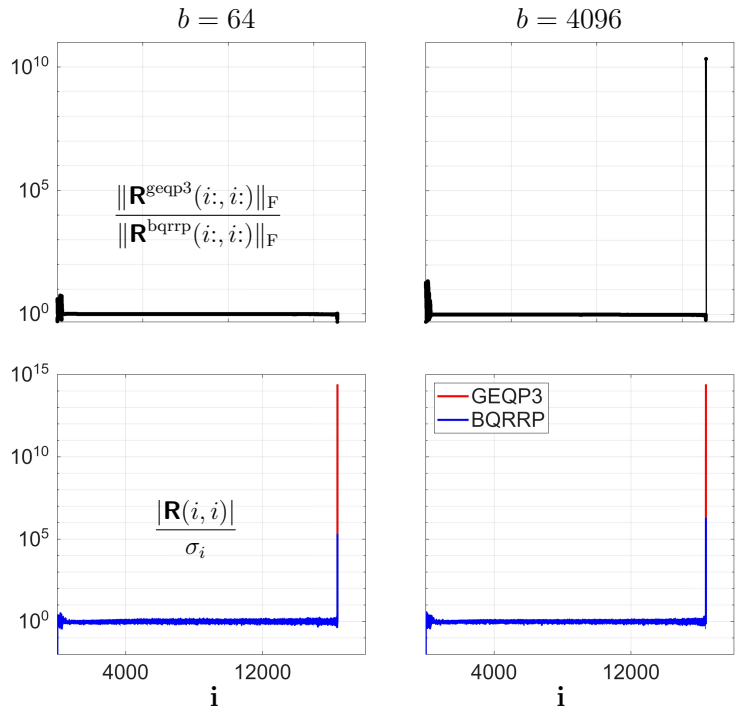


Figure 3.8: Pivot quality results for `BQRRP` with block sizes 64 and 4096 show that block size has limited impact on pivot quality. The residual-norm ratio is generally similar to that of `GEQP3`, diverging mainly near sharp singular value drops, especially for $b = 4096$. The second metric shows near-identical behavior, except at the final singular value.

Table 3.3: Reconstruction error and orthogonality loss in `GEQP3` and `BQRRP` with varying `BQRRP` block sizes, captured across input square matrices of varying sizes and types.

	$m = n$	spectrum($\mathbf{M}$)	$\frac{\|\mathbf{M}(:, \boldsymbol{J}) - \mathbf{QR}\|_{\mathrm{F}}}{\|\mathbf{M}\|_{\mathrm{F}}}$	$\frac{\|\mathbf{I} - \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\|_{\mathrm{F}}}{\sqrt{n}}$	$\frac{\|\mathbf{M}(:, \boldsymbol{J}) - \mathbf{QR}\|_{\mathrm{F}}}{\|\mathbf{M}\|_{\mathrm{F}}}$	$\frac{\|\mathbf{I} - \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\|_{\mathrm{F}}}{\sqrt{n}}$
`GEQP3`	16000	polynomial	1e-15	1e-15	–	–
		staircase	1e-15	1e-15	–	–
		spiked	1e-15	1e-15	–	–
	32000	polynomial	1e-15	1e-15	–	–
		staircase	1e-15	1e-15	–	–
		spiked	1e-15	1e-15	–	–
	64000	polynomial	1e-15	2e-15	–	–
		staircase	2e-15	2e-15	–	–
		spiked	2e-15	1e-15	–	–
			`BQRRP` $b = 250$		`BQRRP` $b = 8000$	
`BQRRP`	16000	polynomial	1e-15 ± 5e-18	1e-15 ± 1e-18	9e-16 ± 5e-18	1e-15 ± 1e-18
		staircase	1e-15 ± 3e-18	1e-15 ± 1e-18	1e-15 ± 2e-18	1e-15 ± 1e-18
		spiked	1e-15 ± 1e-18	1e-15 ± 1e-18	1e-15 ± 1e-18	1e-15 ± 8e-19
	32000	polynomial	1e-15 ± 2e-18	1e-15 ± 1e-18	1e-15 ± 3e-18	1e-15 ± 8e-19
		staircase	1e-15 ± 1e-18	1e-15 ± 1e-18	1e-15 ± 2e-18	1e-15 ± 9e-19
		spiked	1e-15 ± 8e-19	1e-15 ± 1e-18	1e-15 ± 1e-18	1e-15 ± 1e-18
	64000	polynomial	1e-15 ± 1e-18	2e-15 ± 9e-19	1e-15 ± 2e-18	2e-15 ± 5e-19
		staircase	1e-15 ± 7e-19	2e-15 ± 7e-19	1e-15 ± 5e-19	2e-15 ± 6e-19
		spiked	1e-15 ± 3e-19	2e-15 ± 6e-19	8e-16 ± 2e-16	1e-15 ± 7e-18

3.7 Performance results

The experiments that are conducted in this section compare the performance of the following QR and QRCP algorithms:

- `BQRRP_CQR` – a version of `BQRRP_CPU` that uses Cholesky QR (and its dependencies) on a panel, `ORMQR` for the updating step, and Algorithm 4 in Step Algorithm 3.
- `BQRRP_HQR` – a version of `BQRRP_CPU` that uses Householder QR on a panel, `ORMQR` for the updating step, and Algorithm 4 as a black-box `qrcp_wide`.
- `GEQRF` – standard unpivoted Householder QR.

Forthcoming Section 3.7.1 presents the results of benchmarking on CPUs; the following section Section 3.7.2 discusses the results on GPUs. The sampling factor, γ, is set to the default value of 1.0 in all experiments (since this is the only reasonable choice if Algorithm 4 is in use at step 7). In all experiments, the performance is measured via *canonical* FLOP rate, relying on the FLOP count of `GEQRF`.

3.7.1 CPU algorithms performance

In addition to the algorithms listed above, CPU experiments also involve benchmarking `GEQP3` – standard pivoted QR, and `HQRRP` – randomized pivoted QR algorithm from [90]. I ported an LAPACK-compatible implementation [75] of HQRRP into RandLAPACK (found in `/RandLAPACK/drivers/rl_hqrrp.hh`) for easier benchmarking.

The first set of experimental results is shown in Fig. 3.9. Algorithms were run on $m_{1,2} \times m_{1,2}$ matrices using block sizes $b_{1,2}$, with $m_1 = 65{,}536$ (Fig. 3.9, row one) and $m_2 = 64{,}000$ (Fig. 3.9, row two), with block size in `BQRRP_CPU` and `HQRRP` varying as the powers of two of multiples of ten as $b_1 = 256 \cdot \{1, 2, 4, \ldots, 32\}$ and $b_2 = 250 \cdot \{1, 2, 4, \ldots, 32\}$.

I set the `HQRRP` block sizes to the same values as `BQRRP` block sizes, given that the internal logic of `HQRRP` is extremely similar to that of `BQRRP`. This allows us to explore how similar block size decisions impact both algorithms. A detailed exploration of the optimal `HQRRP` block size is provided in Section 3.8.2.

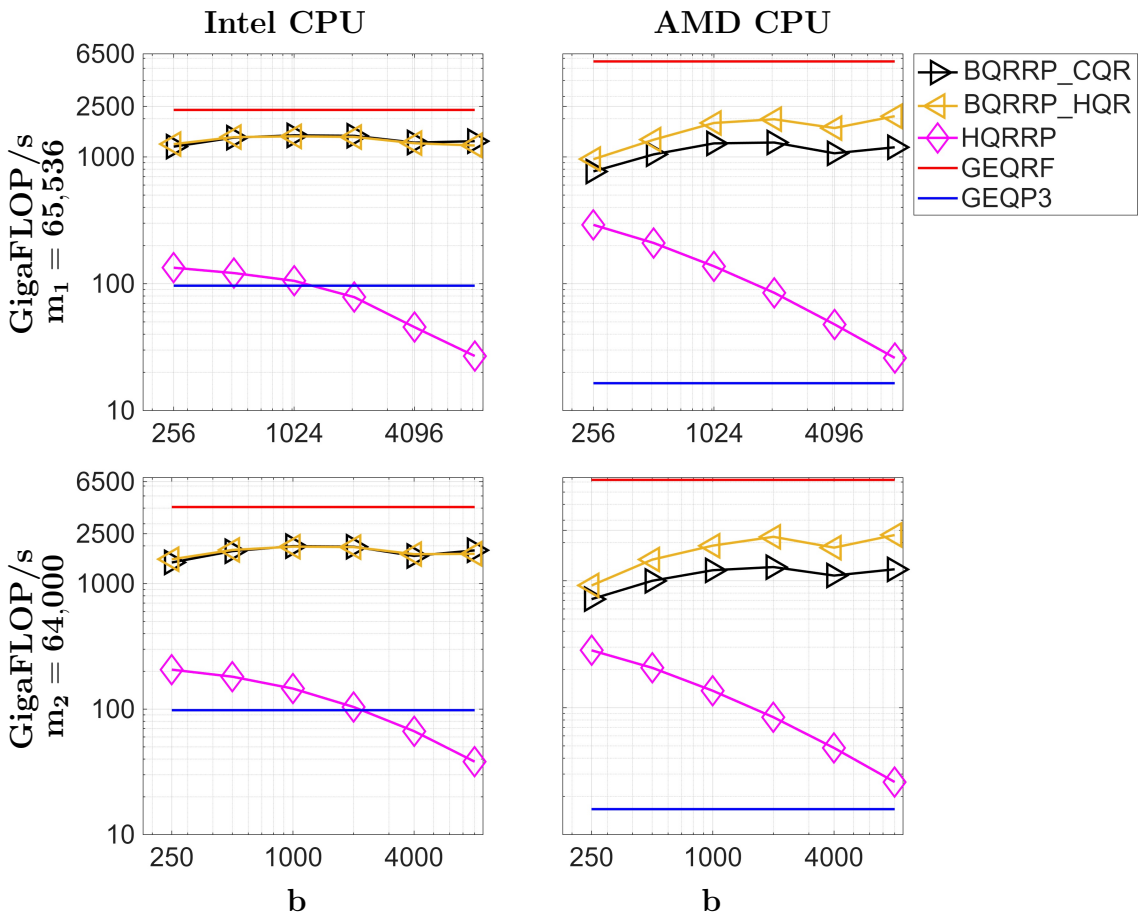


Figure 3.9: Performance of various QR and QRCP methods, captured on an Intel and AMD systems (see Table 3.1). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m_1 \times m_1$ and $m_2 \times m_2$ with `BQRRP` and `HQRRP` block sizes b_1 and b_2, where $m_1 = 65{,}536$ and $m_2 = 64{,}000$, $b_1 \in \{256, 512, 1024, 2048, 4096\}$ and $b_2 \in \{250, 500, 1000, 2000, 4000\}$. The execution of `GEQRF` and `GEQP3` does not depend on the block size parameter b, and hence the performance is depicted as constant. Observe that on an Intel system, both versions of `BQRRP_CPU` exhibit comparable performance. On an AMD system, `BQRRP_CPU` with Householder QR is the fastest method.

Fig. 3.9 shows that on an Intel system, BQRRP_CPU with Cholesky QR on a panel has near-identical performance to that with Householder QR on a panel. BQRRP_CQR and BQRRP_HQR are up to 19× faster than the standard pivoted QR, GEQP3 (as seen in the bottom left plot in Fig. 3.9), and they achieve up to 60% of performance of the unpivoted QR, GEQRF (as seen in the top left plot in Fig. 3.9). Furthermore, BQRRP_CPU algorithms are 7×–20× faster than HQRRP, depending on the block size used.

On an AMD system, BQRRP_CPU with Householder QR on a panel is the fastest QRCP method. BQRRP_HQR is up to 148× faster than the standard pivoted QR, GEQP3 (as seen in the bottom right plot in Fig. 3.9), and it achieves up to 35% of performance of the unpivoted QR, GEQRF (as seen in the top right plot in Fig. 3.9). Furthermore, it is 3×–148× faster than HQRRP, depending on the block size used.

Thread scaling results. Fig. 3.10 depicts thread scaling results for the QR and QRCP schemes run on $m \times m$ matrices with $m \in \{8{,}000,\ 16{,}000,\ 32{,}000\}$. In these plots, I set the block size parameter in BQRRP_CQR and BQRRP_HQR to $b = m/32$. This is because Fig. 3.9 shows that this block size setting generally yields the best BQRRP performance across all experiments.

Fig. 3.10 shows that BQRRP exhibits good thread scaling. For a 32,000 × 32,000 matrix, BQRRP's performance improves by a factor of 20× on the Intel system and 37× on the AMD system when scaling from 1 to 128 threads. In comparison, GEQP3 achieves only a 6× speedup on Intel and exhibits negligible thread scaling on AMD.

The standout performer is, as expected, GEQRF, with speedups of 43× on Intel and 99× on AMD. Observe also that the performance of GEQRF on the AMD system begins to exceed that on the Intel system at 128 threads for the matrix of size 32,000×32,000. This is in line with what was shown in Fig. 3.9, where GEQRF and BQRRP were showing much better overall performance on an AMD system.

Observe also that the performance of HQRRP begins to stagnate at 64 threads used on an Intel system. I present a thorough investigation of HQRRP thread scaling results for various block sizes in Section 3.8.2.

In the bottom-left plot of Fig. 3.10, the performance of BQRRP approaches that of GEQP3 for a matrix of size 8,000 × 8,000, with only a 1.5× difference. This suggests that a BQRRP block size of $m/32$ may be suboptimal for smaller input matrices.

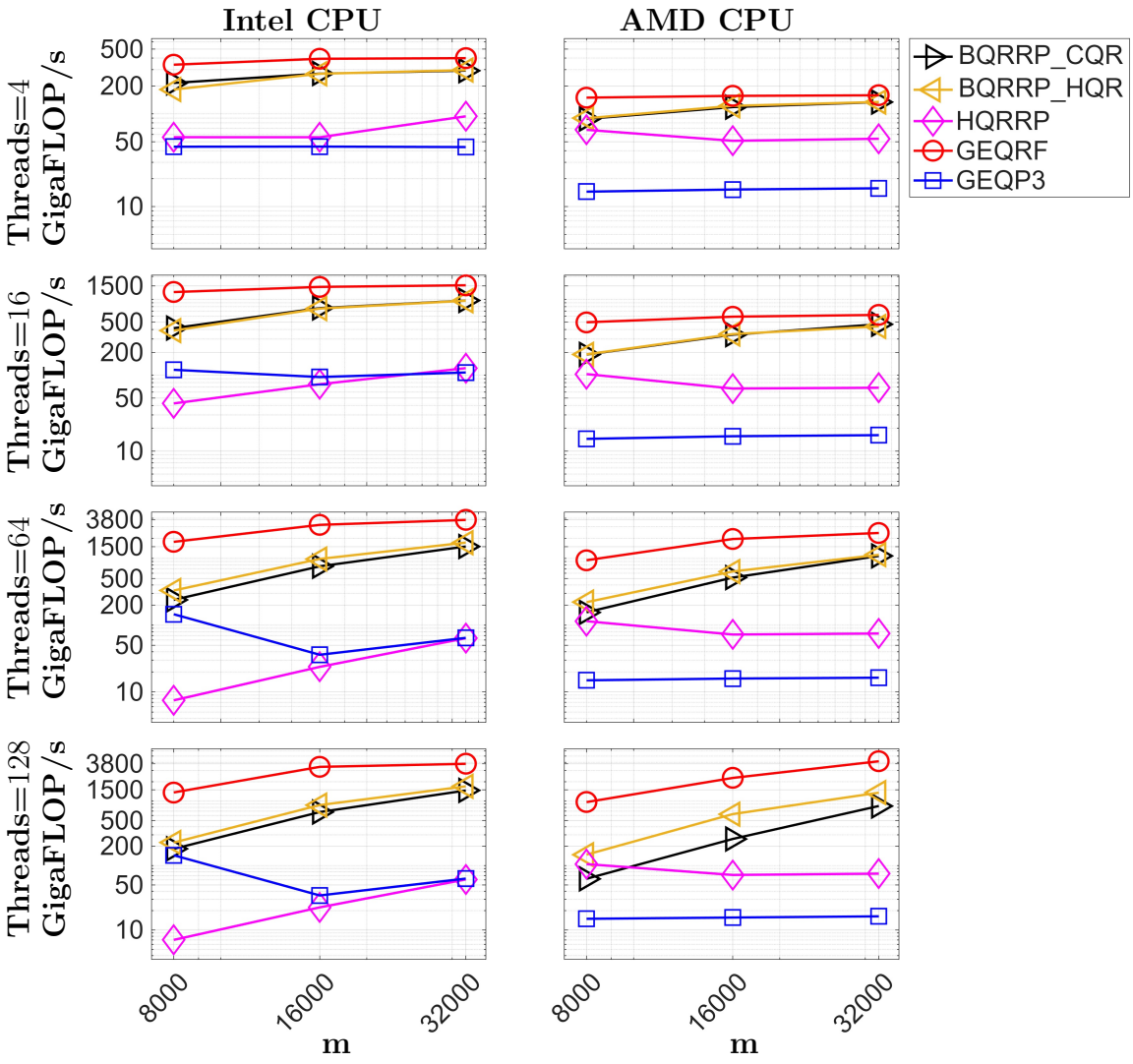


Figure 3.10: Effects of varying the numbers of threads on the FLOP rates in QR and QRCP methods, captured on Intel and AMD systems (see Table 3.1). In these experiments, $m \times m$ matrices are used, where $m \in \{8{,}000,\ 16{,}000,\ 32{,}000\}$; `HQRRP`, `BQRRP_CQR`, and `BQRRP_HQR` block size is set to $b = m/32$. Observe that both versions of `BQRRP_CPU` depict thread scaling on par with `GEQRF` for the larger matrix sizes. Note, however, that there is an apparent slowdown in the performance of `GEQRF` and `BQRRP` on a matrix with $m = n = 8{,}000$ when switching between 64 and 128 threads; this is due to the fact that using 128 threads is an overkill for a matrix of this size. Simultaneously, both `HQRRP` and `GEQP3` exhibit poor thread scaling (especially on the AMD system).

Performance results on smaller inputs. As stated earlier, although the results from Fig. 3.9 show that BQRRP performs best when the block size b is set to $b = n/32$ (given an $m \times n$ input matrix), this may only hold for larger input matrix sizes (as seen in Fig. 3.10, the performance of BQRRP gets suspiciously close to that of GEQRF when the input is of size $8{,}000 \times 8{,}000$). As such, I investigate what block size setting works best for the smaller input matrices.

Fig. 3.11 shows that in all the smaller matrices tested across both systems, using block sizes that are either twice smaller or as large as $m = n$ is the best choice. Fig. 3.11 also suggests that using BQRRP_HQR over BQRRP_CQR is always by far the best option and that at no point does the performance of HQRRP exceed that of GEQP3.

Varying the aspect ratio in the test matrices. Section 3.0.3 briefly explains my choice of test matrix types and sizes. As noted there and earlier in this section, I primarily use square matrices to evaluate QR and QRCP schemes. However, since BQRRP applies to any aspect ratio, I also present results for tall and wide matrices. I do not, however, explore the *extremely* tall or wide cases, where specialized algorithms may be more suitable (Chapter 2, [59, 9]).

Fig. 3.12 depicts the expected results of the performance in the two BQRRP versions, reaching closer to that of GEQRF as the input matrix size increases. Observe that using the BQRRP block size $b = n/32$ is suboptimal when testing smaller matrices.

3.7.2 Performance of GPU Implementations

As noted in Section 3.3, a major challenge in designing GPU versions of algorithms is the lack of GPU implementations for many LAPACK-level functions. Unlike the CPU experiments, the only readily available GPU algorithm to compare with BQRRP_GPU is cuSOLVER's GEQRF[§]. I therefore compare cuSOLVER's GEQRF with two BQRRP_GPU variants: one using Cholesky QR, and another using Householder QR for panel factorization. Experiments are run on $m \times m$ matrices with $m \in 256 \cdot \{8, 16, 32, \ldots, 128\}$ (Fig. 3.13), and block sizes from 32 to 2048.

Fig. 3.13 shows that the implementation of BQRRP_GPU with Householder QR on a panel is able to achieve up to 60% of the performance of the unpivoted QR method offered by cuSOLVER. The performance of BQRRP_GPU relying on Cholesky QR is far from acceptable due to the fact that a slow implementation of ORHR_COL was used.

[§]A GPU QRCP exists in MAGMA [87], [118], but is not widely used.

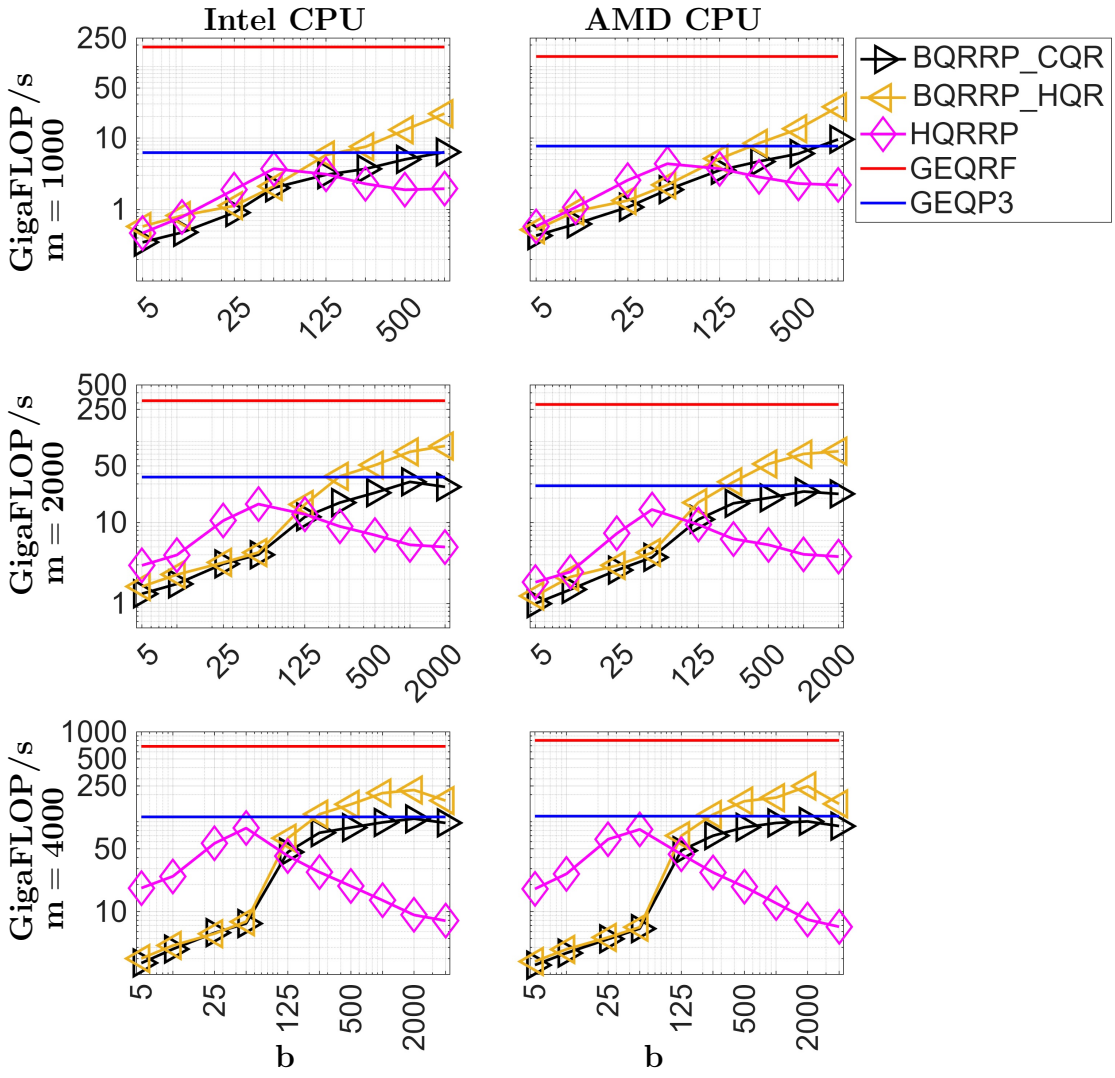


Figure 3.11: Performance of various QR and QRCP methods, captured on Intel and AMD systems (see Table 3.1). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m \times m$ with `BQRRP` and `HQRRP` block size b, where $m \in \{1000, 2000, 4000\}$, and $b \in \{5, 10, 25, 50, 125, 250, 500, 1000, 2000, 4000\}$. The execution of the `GEQRF` and `GEQP3` functions does not depend on the block size parameter b, and hence the performance is depicted as constant. This figure concentrates on assessing how varying the block size parameter in `BQRRP` affects its performance in the context of the smaller input matrices.

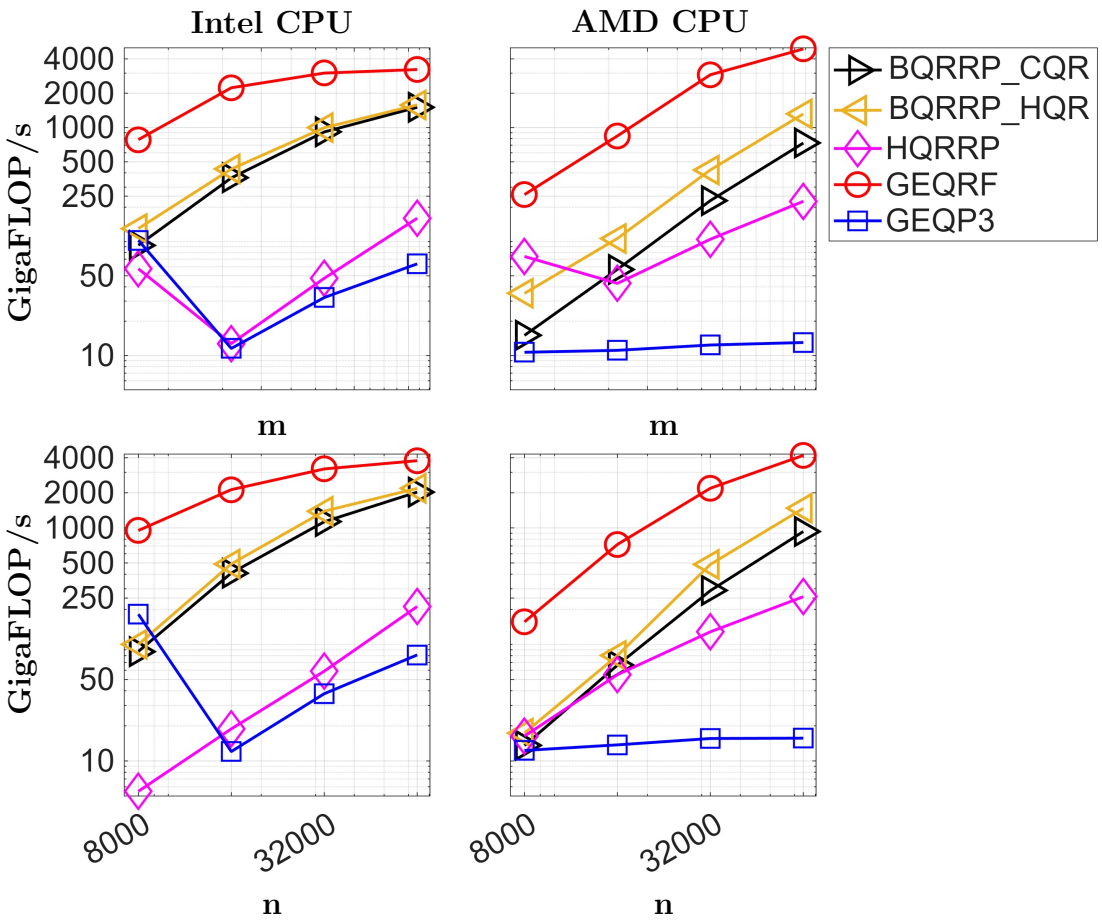


Figure 3.12: Performance of various QR and QRCP methods, captured on Intel and AMD systems (see Table 3.1). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`. The first row depicts tall input matrices, such that $m = 2n$, with $m \in \{8{,}000, 16{,}000, 32{,}000\}$. The second row depicts wide input matrices, such that $n = 2m$, with $n \in \{8{,}000, 16{,}000, 32{,}000\}$. `BQRRP` block size is set to $n/32$, `HQRRP` block size is set to 128.

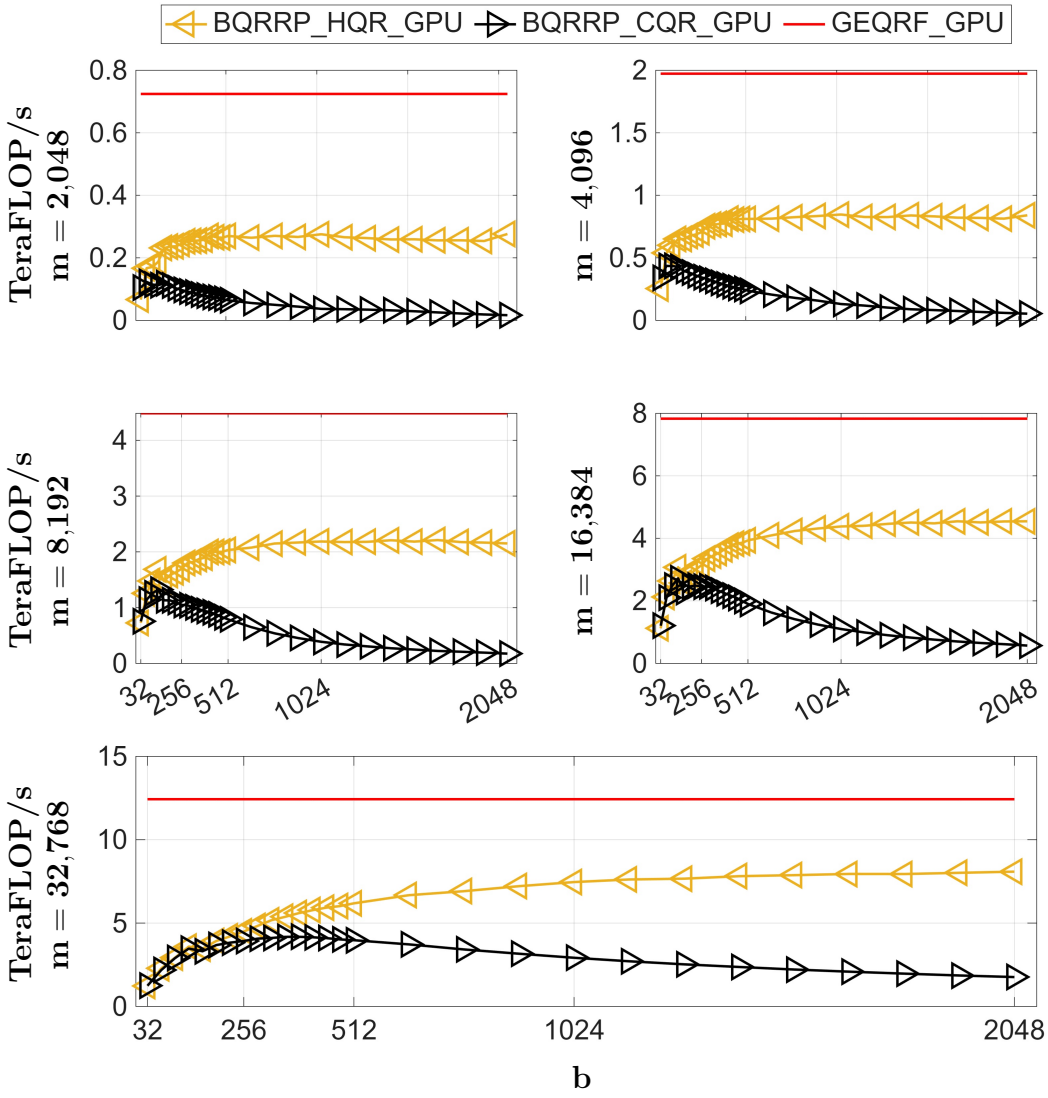


Figure 3.13: Performance of standard QR and two versions of `BQRRP_GPU`, captured on an NVIDIA GPU (for the details on system configuration, refer to Table 3.2). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m \times m$ with `BQRRP` and `HQRRP` block size b, where $m \in \{2{,}048, 4{,}096, 8{,}192, 16{,}384, 32{,}768\}$, and $b \in \{32, \ldots, 2{,}048\}$. The cuSOLVER's `GEQRF` performance is constant, as it is independent of the block size b. `BQRRP` with Householder QR outperforms its Cholesky-based variant, aligning with the profiling results in Fig. 3.6. The relative performance of both `BQRRP_GPU` versions to cuSOLVER remains stable across input sizes. Notably, smaller matrices reach peak performance with smaller block sizes.

Fig. 3.13 also shows that the relative performance of the two `BQRRP_GPU` schemes to cuSOLVER's `GEQRF` scales with the change in the input matrix size.

3.8 Investigating HQRRP performance

Recall from Section 1.1.1 that I discussed prior work on developing a modern QRCP approach, highlighting `HQRRP` [90] as a particularly promising scheme. Nevertheless, comparing the results in Fig. 1.1 with those reported in [90, Fig. 1] and [90, Fig. 5], observe that the performance gap between `GEQRF` and `GEQP3` has increased from under 10× to around 100× on a modern AMD CPU, while the speedup of `HQRRP` over `GEQP3` has not scaled proportionately, remaining at approximately 13× at best. Moreover, from column one in Fig. 1.1, one can see that the performance of `HQRRP` may fall below that of `GEQP3` as the number of OpenMP threads in use increases.

This overall difference in performance is, of course, largely attributable to the drastic differences in hardware platforms: my experiments were conducted on modern CPUs, described in Table 3.1, whereas the experiments from [90] were performed on an Intel Xeon E5-2695 v3 (the Haswell platform) processor featuring "only" 14 cores in a single socket, a platform that is now over a decade old. Nonetheless, to thoroughly assess the `HQRRP` algorithm, I investigate how tuning its *block size* parameter affects performance on modern systems (although [90, Sec. 4.1] suggests that using the block size of 64 or 128 should yield near-best performance, this may not hold true on the hardware systems used). Furthermore, I perform the subroutine performance profiling (similar to Section 3.4) in `HQRRP` to identify any computational bottlenecks. Finally, it is worth analyzing the performance of *each individual* algorithm involved in Fig. 1.1 in order to detect any performance instabilities of each given scheme (of particular interest is the relationship between the performance of `GEQRF` and `GEQP3` on an Intel system). The following subsections address each of these tasks.

3.8.1 Alternative view of Fig. 1.1

Fig. 3.14 presents the performance results of running `HQRRP`, `GEQRF`, and `GEQP3`, measured in terms of the canonical FLOP rate, as opposed to relative speedup (Fig. 1.1). As such, one can assess any performance instabilities of each individual algorithm. Observe that the algorithms perform rather unstably on an Intel CPU across the board, even though, as stated in Section 3.0.3, I perform five runs of

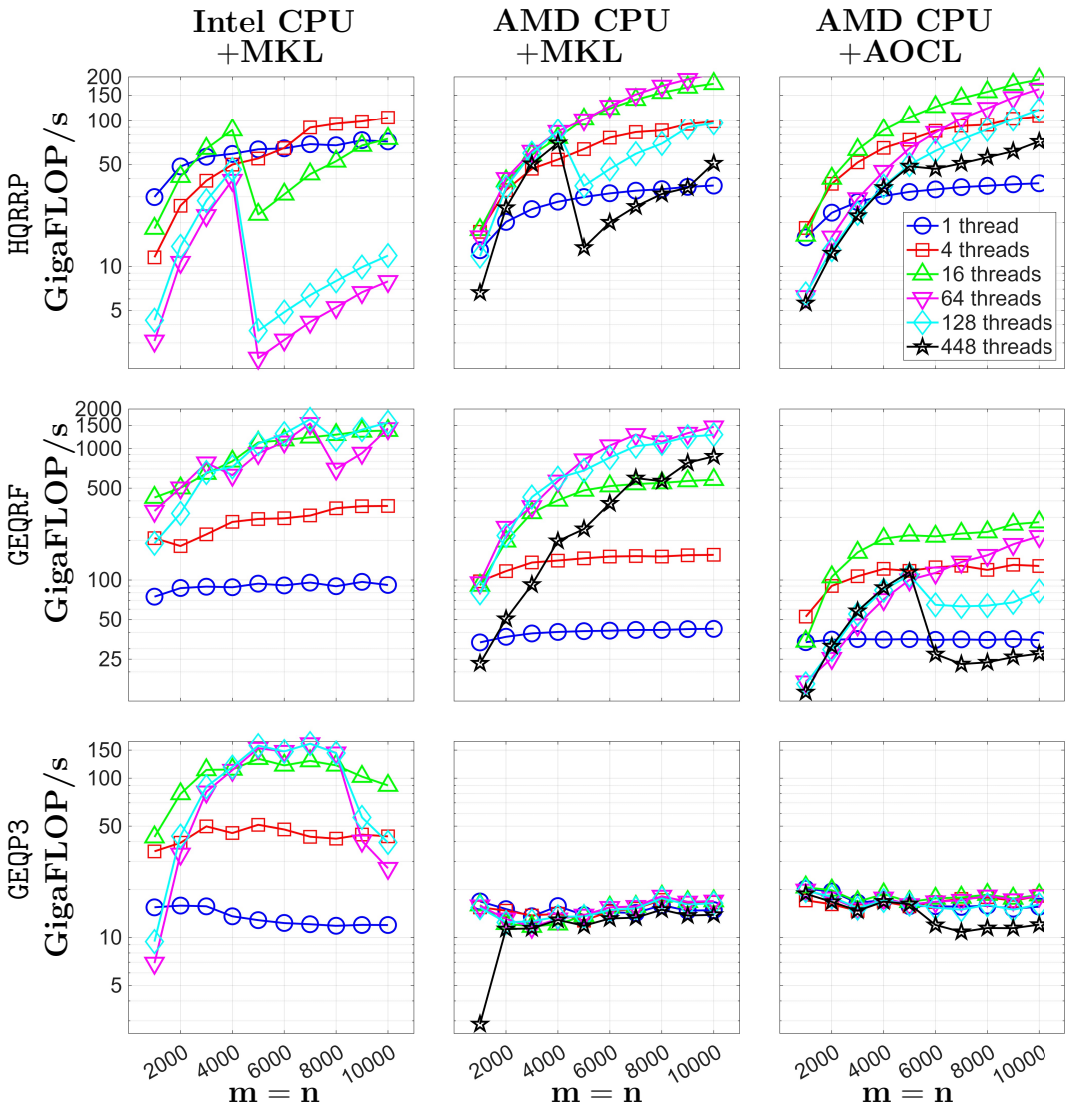


Figure 3.14: Performance of `HQRRP` (first row), `GEQRF` (second row), and `GEQP3` (third row), attained on matrices of sizes between 1,000−by−1,000 to 10,000−by−10,000, when varying the number of OpenMP threads used. In all experiments, `HQRRP` block size was set to 128, as suggested in [90, Sec. 4.1]. Performance is measured in terms of *canonical* FLOP rate, relying on the FLOP count of the standard LAPACK QR function (`GEQRF`). Results were captured on machines, described in Table 3.1. Column three depicts results obtained on an AMD CPU using AOCL 5.0.0 (as opposed to MKL 2025.0, used elsewhere).

each algorithm per given matrix size to address the potential instabilities. Of particular interest is the fact that increasing the number of OpenMP threads used only occasionally has a positive effect on the performance of `HQRRP` on Intel hardware. Additionally, it is worth noting that the performance of `GEQP3` on Intel CPU is far superior to that on AMD CPU, regardless of which vendor library the algorithm is sourced from.

Remark 26. Additional observations can be made regarding the relative performance of algorithms run on AMD hardware when using AOCL versus MKL. As seen from the two rightmost columns in Fig. 3.14, while `HQRRP` performs roughly the same at its peak, the performance of `GEQRF` and `GEQP3` sourced from MKL is far superior to that from AOCL. This observation is the basis for my decision to report AMD system results using MKL instead of AOCL. Furthermore, this observation pushed me to investigate the performance of AOCL relative to other vendor NLA libraries even further. As such, in order to profile some common BLAS and LAPACK functions, I have added the following benchmarking files into RandLAPACK: [19], [20]. After a thorough investigation, I concluded that AOCL LAPACK performance is significantly worse than that of MKL and OpenBLAS on AMD's own hardware, except when running in single-threaded mode. This observation was confirmed by an AMD support engineer in March 2025, and an internal AMD issue report has been submitted.

3.8.2 Varying `HQRRP` block size

As stated previously, in the experiments presented in Fig. 1.1 and Fig. 3.14, the `HQRRP` block size parameter is set to 128, per the suggestion in [90, Sec. 4.1]. Fig. 3.15 shows how varying the block size in `HQRRP` affects its performance for the various numbers of OpenMP threads used.

Results in Fig. 3.10 show that the performance of `HQRRP` indeed peaks at around block size $64 - 128$, as [90] suggests. However, comparing Fig. 3.10 with Fig. 3.15, observe that even at its best, `HQRRP` does not reach the performance of either version of `BQRRP` (except, of course, in a single-threaded case). Furthermore, as seen previously, the performance of `HQRRP` begins to stagnate at 64 OpenMP threads, suggesting that some suboptimal (possibly, Level-2 BLAS) subroutines may have been used in `HQRRP`.

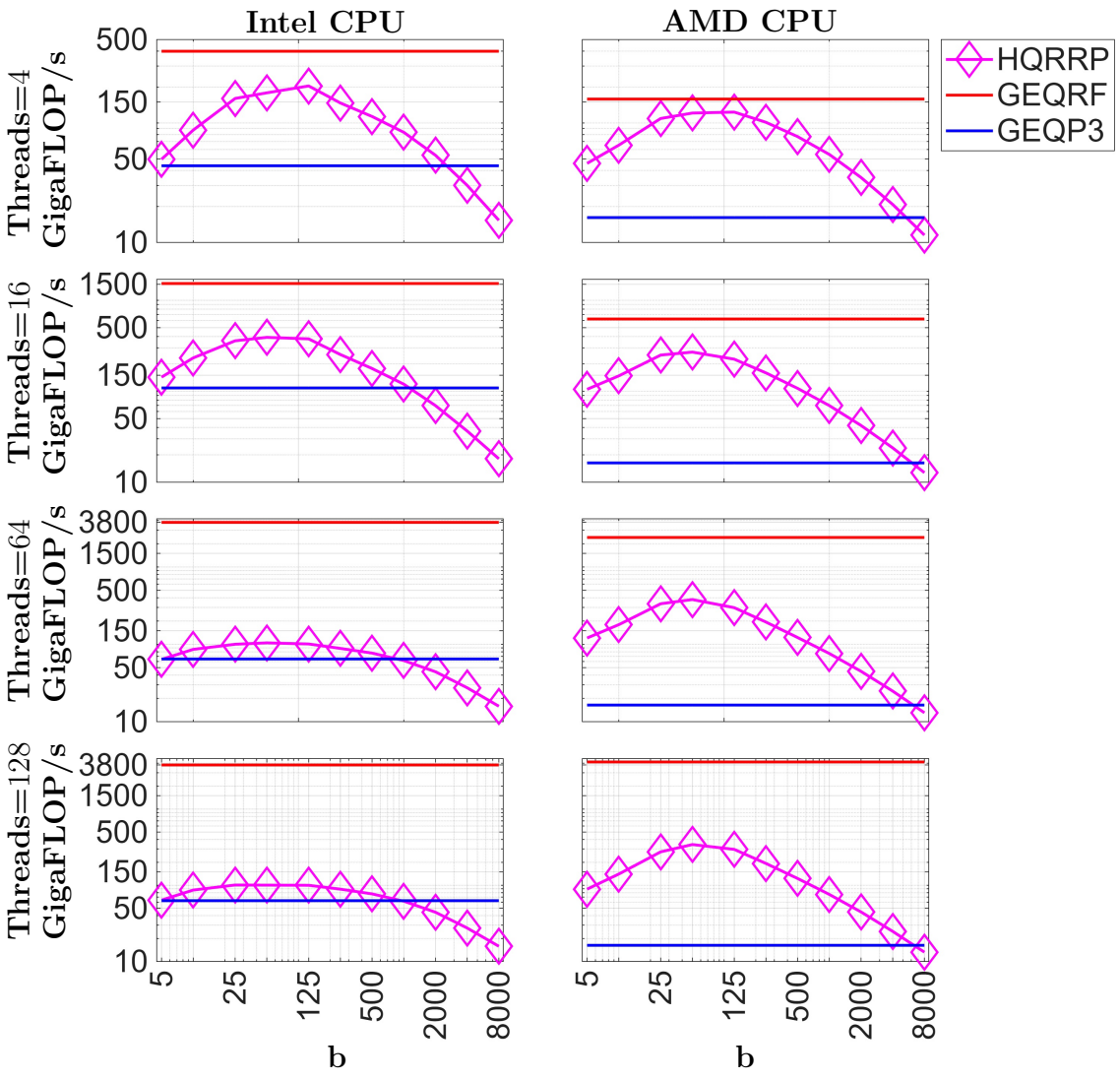


Figure 3.15: Performance comparison of `HQRRP`, `GEQRF`, and `GEQP3`, captured on Intel and AMD systems (see Table 3.1). The experiments are conducted on matrices with 32,000 rows and columns and the `HQRRP` block size parameter $b \in \{5,\ 10,\ 25,\ 50,\ 125,\ 250,\ 500,\ 1000,\ 2000,\ 4000,\ 8000\}$ (x-axis).

3.8.3 HQRRP subroutines profiling

In an effort to better understand the performance gap between BQRRP and HQRRP seen in Fig. 3.10 , I present the subroutines' performance breakdown of HQRRP below (similar to how I did it for BQRRP in Section 3.4). Fig. 3.16 depicts the percentage of runtime that is occupied by a given subcomponent of HQRRP on the y-axis. I use square test matrices with 32,000 rows and columns and the block size parameter $b \in \{5,\ 10,\ 25,\ 50,\ 125,\ 250,\ 500,\ 1000,\ 2000,\ 4000,\ 8000\}$ (x-axis).

As seen in Fig. 3.16, the function that is responsible for updating the input matrix tends to occupy most of the HQRRP runtime when smaller block sizes are in use (and especially for the lower numbers of OpenMP threads used). In the LAPACK-compatible HQRRP implementation, this function is called "NoFLA_Apply_Q_WY_lhfc_blk_var4." It relies on a single LAPACK subroutine, LARFB, which applies a block reflector to a given rectangular matrix. Other notable subroutines come from the QR and QRCP functions within HQRRP, which are performed via calling "NoFLA_QRPmod_WY_unb_var4" in pivoted and unpivoted modes, respectively. There, the most costly functions are LARF (applies a single reflector to a given rectangular matrix). Finally, the column permutation strategy in HQRRP, implemented in "NoFLA_QRP_pivot_G_B_C" becomes costly when larger numbers of OpenMP threads are used (particularly at 64 threads, where the stagnation in HQRRP performance begins).

Overall, I conclude that using LARF within HQRRP is suboptimal, since this function operates on a single reflector at a time, and consequently is largely cast in terms of level 2 BLAS. Furthermore, it could be the case that, depending on the system and the input problem, ORMQR used in BQRRP is superior to LARFB used in HQRRP.

3.9 CQRRPT revisited

This section was written approximately one year after the conclusion of the CQRRPT project, in the way it was presented in [93]. Naturally, over this period, both the RandBLAS and RandLAPACK software libraries have undergone substantial improvements. Additionally, I have gained access to new hardware platforms (compare the hardware in Table 3.1 and Table 2.1 for details). Given these developments, I am now in a position to revisit the performance of CQRRPT under these improved conditions.

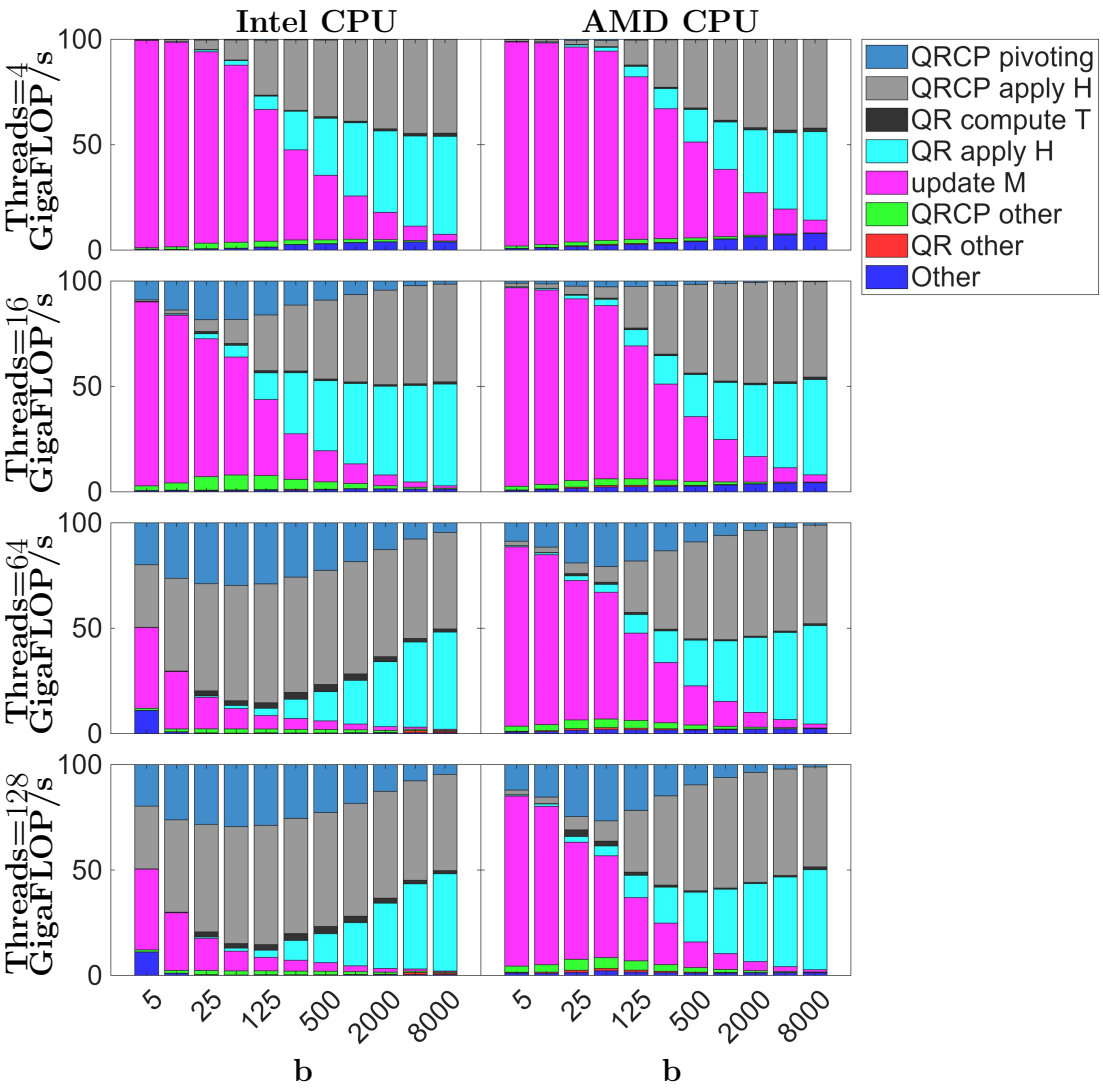


Figure 3.16: Percentages of `HQRRP` runtime, occupied by its respective subroutines. Experminets were conducted on square matrices of size $32{,}000 \times 32{,}000$ with the `HQRRP` block size b taking values $\{5,\ 10,\ 25,\ 50,\ 125,\ 250,\ 500,\ 1000,\ 2000,\ 4000,\ 8000\}$. The results are captured on Intel and AMD systems (see Table 3.1).

As illustrated in Fig. 2.4, the primary computational bottlenecks in CQRRPT are: (1) the generation and application of the sketching operator for inputs with relatively few columns, and (2) the QRCP on the sketch matrix for inputs with a larger number of columns. Recent updates to RandBLAS may have addressed the first of these issues. To mitigate the second, I plan to experiment with both BQRRP (introduced in Chapter 3) and HQRRP (see Section 1.1.1) as alternative methods for the near-square QRCP step.

Since CQRRPT is primarily designed for tall matrices, I will begin by evaluating BQRRP on a range of small square matrices to determine optimal block size configurations. In addition to the results shown in Fig. 3.11, I include further data for 500×500 and 8000×8000 matrices in Fig. 3.17.

Based on my observations, the block size parameter in BQRRP should be selected as a function of the matrix dimension dim according to the following heuristic:

- If $\text{dim} \leq 2000$, set $b = \text{dim}$;
- If $2000 < \text{dim} \leq 8000$, set $b = \text{dim}\,/2$;
- Otherwise, set $b = \text{dim}\,/32$.

In Fig. 3.18, I present the comparison of the performance of algorithms, described in Section 2.5, to three versions of CQRRPT: standard, presented in Chapter 2, fused with HQRRP, as described in Section 2.5.1, and the newest version, where the QRCP on a sketch is done via BQRRP_HQR, described in Section 3.2. In Fig. 3.19, I present the breakdown of BQRRP-based CQRRPT by subroutine, similar to Fig. 2.4. These experiments are performed on matrices with $m = \{64{,}000, 128{,}000, 1{,}000{,}000\}$ and $n = \{50, 125, 250, 500, 1000, 2000, 4000, 8000\}$.

Fig. 3.18 shows a significantly different image than Fig. 2.3. Standard CQRRPT now fails to deliver performance superior to any method except standard GEQP3, unless it is run on the largest matrices in the test set. BQRRP-based CQRRPT, however, shows reasonable performance for all matrices with the number of columns exceeding 1000. It even manages to outperform both GEQRF and GEQR for matrices with $m = 1{,}000{,}000$ and $n = \{4000, 8000\}$. Clearly, CQRRPT is overall best suited for large matrix sizes.

Interestingly enough, Fig. 3.19 shows that for all tested column sizes, QRCP on a sketch, which is done by BQRRP, occupies over 90% of CQRRPT's runtime. Clearly, all other routines that CQRRPT is comprised of are excellently optimized in Intel MKL

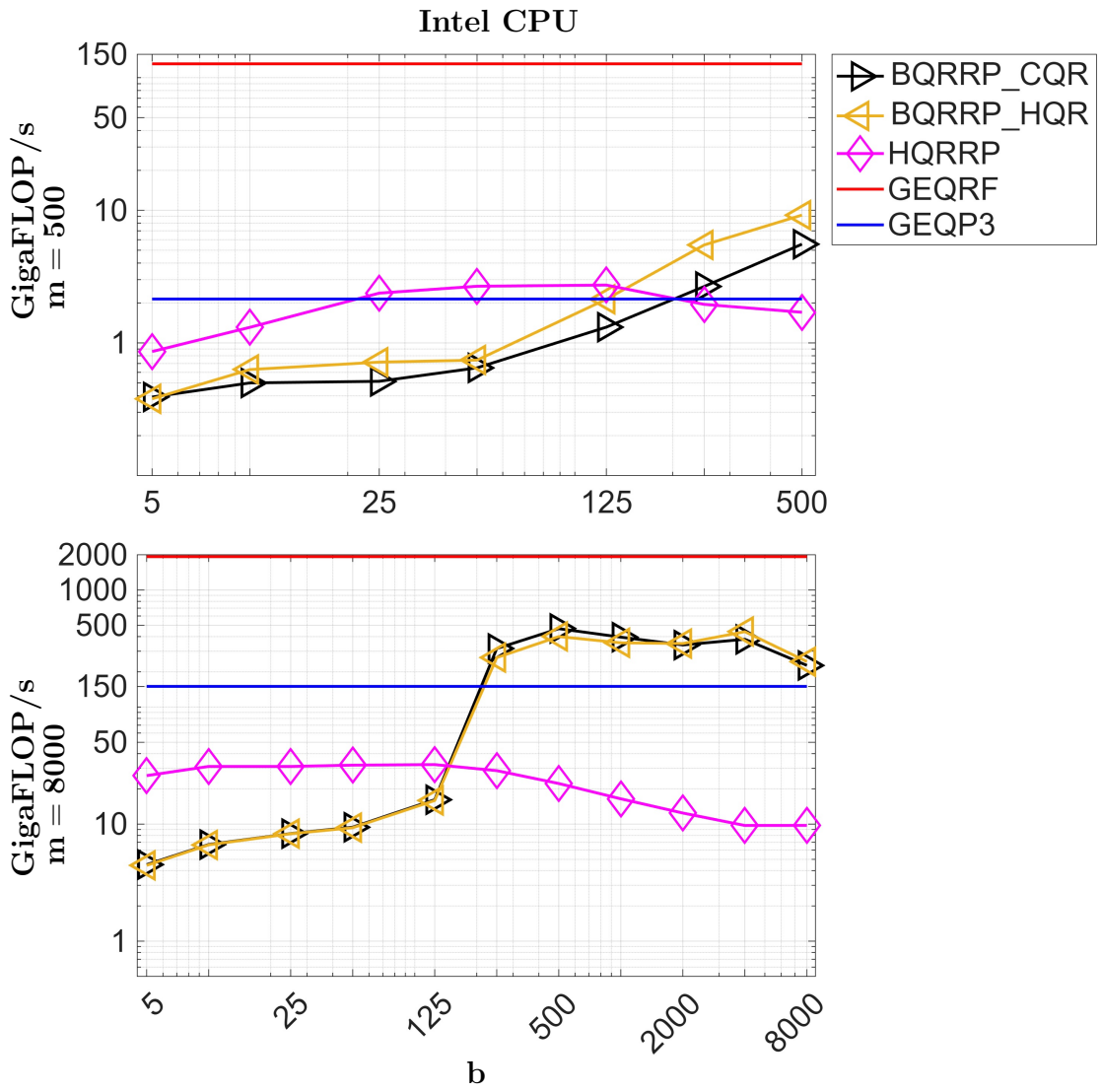


Figure 3.17: Performance of various QR and QRCP methods, captured on Intel system (see Table 3.1). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m \times m$ with `BQRRP` and `HQRRP` block size b, where $m \in \{500, 8000\}$, and $b \in \{5, 10, 25, 50, 125, 250, 500, 1000, 2000, 4000\}$. The execution of the `GEQRF` and `GEQP3` functions does not depend on the block size parameter b, and hence the performance is depicted as constant. This figure concentrates on assessing how varying the block size parameter in `BQRRP` affects its performance in the context of the smaller input matrices.

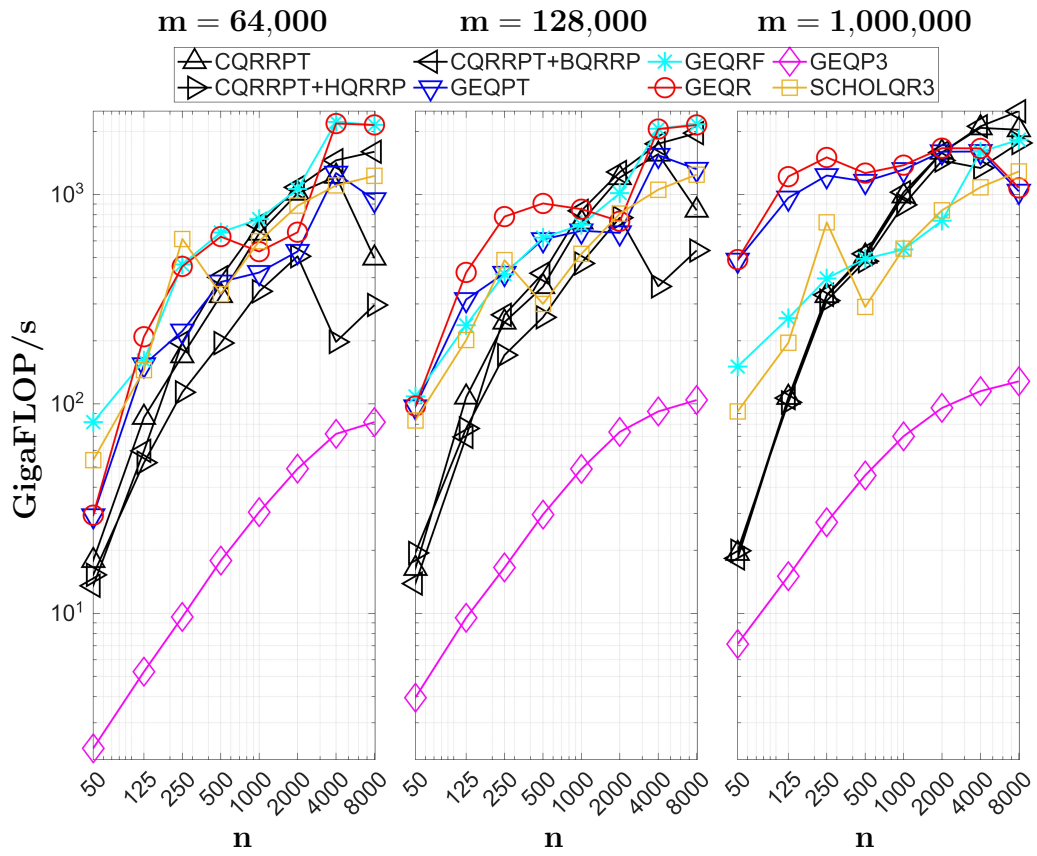


Figure 3.18: Performance of various QR and QRCP methods, captured on an Intel system (see Table 3.1). The performance is measured via canonical FLOP rate, using the FLOP count of LAPACK's `GEQRF`, on matrices of sizes $m \times n$, where $m \in \{64,000, 128{,}000, 1{,}000{,}000\}$, and $b \in \{50, 125, 250, 500, 1000, 2000, 4000, 8000\}$. Note that `BQRRP`-based `CQRRPT` outperforms other versions of `CQRRPT`.

2025.0. This explains well the downgrade in performance of standard CQRRPT in my newest round of experiments.

Although BQRRP_HQR is a rather fast method, as shown in Section 3.7, it is, after all, best suited for large matrices. In principle, one could tune BQRRP perfectly for the particular setting of being used inside CQRRPT, though I believe that such work is beyond the scope of my dissertation.

Why revisit CQRRPT? The inclusion of this section as part of my dissertation raises an important question: since the new CQRRPT performance results presented here differ from those listed in Chapter 2, why did I choose to keep Chapter 2 in line with [93] instead of completely re-conducting CQRRPT benchmarking? Additionally, in light of the new results, can the ones presented in [93] be considered obsolete?

The second question can be answered directly: no, the results shown in [93] and Chapter 2 are not obsolete. They are still relevant for the specific hardware and software configuration that CQRRPT was originally tested with. Moreover, as I point out in Section 2.5.1, an internal QRCP on a sketch becomes the most prominent bottleneck in CQRRPT as the number of columns in the input matrix grows. Considering that with time, the gap in performance between QR and QRCP increases (as pointed out in Section 1.1.1), the results presented as part of Section 3.9 only reinforce the statements made in Chapter 2.

Providing the answer to the first question requires a degree of meta-analysis of this dissertation. I have chosen to present this work as a series of highlights of the most significant research that I conducted during my Ph.D, in chronological order. Each of the presented projects affected the next, with earlier results shaping the direction of subsequent work. Showcasing the results of the CQRRPT work in their original form, therefore, serves the purpose of presenting a snapshot of my state of mind at the time and implicitly illustrates why I chose to pursue the BQRRP project, rather than to devote my efforts elsewhere.

3.10 Chapter conclusion

In this chapter, I introduced BQRRP, a powerful algorithmic framework for QR factorization with column pivoting (QRCP) for general matrices. The framework enables the design of practical QRCP algorithms by allowing users to control key

subroutine choices. I provide a detailed analysis of how these choices can be navigated for modern hardware, presenting formulations of `BQRRP_CPU` and `BQRRP_GPU`. The CPU version is designed for maximally in-place computation, while the GPU version prioritizes performance at the cost of additional storage. Both implementations produce output in the same format as `GEQP3`.

Looking ahead, the relative performance of core subroutines within the BQRRP framework may evolve due to advancements in computational techniques. Nevertheless, this work remains relevant, as it offers a plug-and-play algorithmic framework and a structured approach for analyzing the efficiency of individual subroutines in modern QRCP methods.

My RandLAPACK implementation of `BQRRP_CPU` achieves up to 140× the speed of GEQP3 and up to 60% of the performance of GEQRF. Similarly, `BQRRP_GPU`, also implemented in RandLAPACK, reaches up to 60% of `GEQRF` performance. Given these results and the flexibility of the BQRRP framework, it presents a strong case for inclusion as an alternative to `GEQP3` in LAPACK.

This chapter's CPU benchmarks were conducted on Intel Sapphire Rapids and AMD Zen4c, while GPU results were obtained using an NVIDIA H100. The core subroutine choices were tuned to optimize performance on these architectures. For large matrices, I recommend setting the `BQRRP` block size to 1/32 of the number of columns; while for smaller matrices, it should be set to nearly its maximum feasible value. While this tuning strategy is not rigorously derived, it serves as a general guideline. For more specialized cases, where computations are performed on specific hardware and vendor-optimized libraries, subroutine choices within `BQRRP` should be re-evaluated accordingly.

The adaptability of the presented framework opens the door for future research and practical implementations by engineers looking to integrate `BQRRP` into their software. In particular, it would be valuable to assess `BQRRP`'s performance on ARM-based architectures and consumer-grade hardware such as Apple M-series silicon. Additionally, broader GPU testing is of interest, although current evaluations are limited by RandLAPACK's CUDA-specific GPU support.

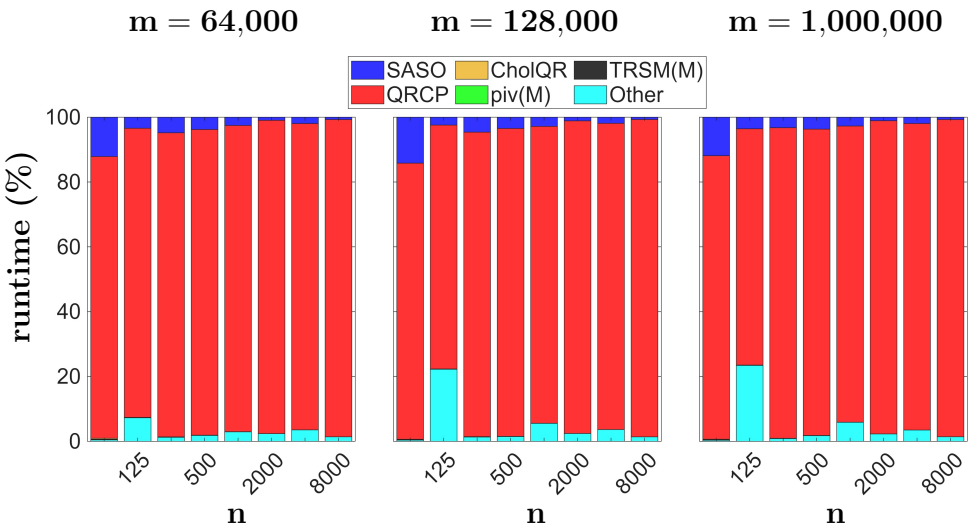


Figure 3.19: Runtime breakdown of `BQRRP`-based `CQRRPT`, captured on an Intel system (see Table 3.1), with matrices of sizes $m \times n$, where $m \in \{64,000, 128{,}000, 1{,}000{,}000\}$, and $b \in \{50, 125, 250, 500, 1000, 2000, 4000, 8000\}$. Note that for all numbers of columns tested, QRCP on a sketch, performed via `BQRRP`, is the most expensive subroutine in `CQRRPT`.

Chapter 4

Accelerating the partial singular value decomposition

In this chapter, I augment the ideas from [120] to deliver new software for computing the partial Singular Value Decomposition (partial SVD). The resulting algorithm is called *ABRIK*[*], which stands for "Adaptive Block Randomized Iterative Krylov method."

4.0.1 Chapter-specific definitions and notation

For the general notation rules, refer to Section 1.2.

The content of this chapter is closely intertwined with the experiments conducted on specific algorithms. Consequently, much of the chapter-specific notation is introduced in context, alongside the algorithms and their corresponding experiments, rather than presented upfront.

In general, I use k to denote the number of singular triplets that the given algorithm aims to approximate. The variable b is used universally to indicate the block size. Finally, p denotes the number of internal iterations within a given algorithm (whether Krylov iterations in `ABRIK` or subspace iterations in `RSVD`, as discussed further).

[*]pronounced "a brick"

4.0.2 Chapter-specific experiments setup

This section outlines the hardware and software setup used in the experiments presented in Section 4.2 and Appendix A.1. For the experimental conventions that are general across all chapters of this dissertation, refer to Section 1.3.1.

Implementation resources. The version of the RandLAPACK software that was used in all experiments in this chapter is documented in [99]. The code for `ABRIK` can be found in `/RandLAPACK/drivers/rl_abrik.hh`. The code for constructing and dispatching the experiments can be found in `/benchmark/bench_ABRIK/*`, as well as `/test/drivers/bench_bqrrp*`. An automated script for re-running all of the experiments shown in this chapter, as well as the MATLAB plotting scripts for replicating the plots, can be found in [23].

Hardware and software configuration. The detailed configurations of the machines that I used for conducting the CPU experiments are shown in Table 4.1.

Test matrices. It is well-understood that the sensitivity of the SVD depends on the spectral properties of the input matrix [114] and that such sensitivity is explicitly related to the performance outcomes of the algorithm [69]. As such, simply generating random inputs via RandBLAS is unwise in this context. Instead, either the real-world datasets or some representative synthetic sets are to be used as input for ABRIK and the alternative methods for computing partial SVD.

With that in mind, I first conducted a round of experiments in Section 4.2.1 using the set of six dense synthetic matrices from [84, Section 7]. Using this dataset should allow us to assess the behavior of `ABRIK` and alternative methods for partial SVD on matrices with a diverse set of spectral properties. This round of experiments is designed to explore the general behavior, or "flavor," of the methods in question. To this end, I generated matrices with relatively modest dimensions, specifically $10{,}000 \times 10{,}000$. Observe Fig. 4.1 for the visualization of the spectra of the dense matrix set.

Remark 27. In [84, Section 7], the parameter "k" is used in the process of test matrix generation to isolate the ranges of the spectra with particular behaviors. This parameter is not to be confused with the notion of *target low numerical rank of the given matrix* (which I am using throughout this chapter), as, in fact, all matrices in the set, except for the fourth matrix (the numerical rank of which is, in fact, "k"),

have full numerical rank. In my experiments, the parameter "k" in the sense of [84, Section 7] is set to 2,000. I apologize in advance for any potential confusion.

I then conducted a second round of experiments on sparse inputs as shown in Section 4.2.2, using matrices from the SuiteSparse Matrix Collection [34]. The selected set represents model reduction problems, consisting of curl-curl-operators of a second-order Maxwell's equations with PEC boundary conditions, with sizes up to 2,380,515×2,380,515 and 26,515,867 nonzeros. These matrices are very ill-conditioned and have slowly decaying spectra with a rapid drop at the tail end. Moreover, curl-curl operators may have *nontrivial nullspace*, which can cause additional trouble to Krylov methods, such as stalled convergence or even stagnation of the Krylov subspace. As such, these matrices can be considered particularly difficult for partial SVD methods.

4.1 Description of algorithms

Algorithm 8 gives pseudocode for `ABRIK`. This formulation is reminiscent of [120, Algorithm A.1], but is presented more concisely.

Remark 28. As noted in [120, Section 6.2 and Appendix A], I assisted the authors in developing the adaptive version of RBKI method (Algorithm A.1), which `ABRIK`'s design most closely resembles. Additionally, I helped finalize the design of the extended RBKI method [120, Algorithm 5.5].

As shown in Algorithm 8, `ABRIK` requires two input parameters: the Krylov block size b and the maximum number of Krylov iterations p. The number of singular triplets approximated by `ABRIK` is given by $\lceil p/2 \rceil b$. It is important to note that the choice of b can have a significant impact on the performance of `ABRIK`.

Algorithm 8 aims to compute the orthonormal bases for the left and right Krylov subspaces, defined respectively as follows:

$$\mathbf{X} = \texttt{orth}([\mathbf{M\Omega}, \cdots, (\mathbf{M}\mathbf{M}^{\mathrm{T}})^{p}\mathbf{M\Omega}]),$$
$$\mathbf{Y} = \texttt{orth}([\mathbf{M}^{\mathrm{T}}\mathbf{M\Omega}, \cdots, (\mathbf{M}^{\mathrm{T}}\mathbf{M})^{p}\mathbf{\Omega}]).$$

Note that the block in the left basis is computed at even iterations of Algorithm 8 (including iteration zero), and the block in the right basis is computed at odd ones.

The blocks in the bases of Krylov subspaces, denoted $\mathbf{X}_i \in \mathbb{R}^{m\times b}$ and $\mathbf{Y}_i \in \mathbb{R}^{n\times b}$, are used to produce structured matrices $\mathbf{S}$ and $\mathbf{R}$, which serve as the *coordinate*

Table 4.1: Key details of the hardware and software configuration of the platform where ABRIK testing was performed.

		Intel Xeon 8462Y+ (2×)
Cores per socket		32
Maximum hardware threads		128
Clock Speed	**Base**	2.80 GHz
	Boost	4.10 GHz
Cache sizes per socket	**L1**	80 KB
	L2	2 MB
	L3	60 MB
RAM		DDR5 1TB
BLAS & LAPACK		MKL 2025.0
FP64 Peak Performance		5.4 TearaFLOPs
Compiler		GCC 13.3.0
CMake		3.31.4
OS		Red Hat Enterprise Linux 8.9

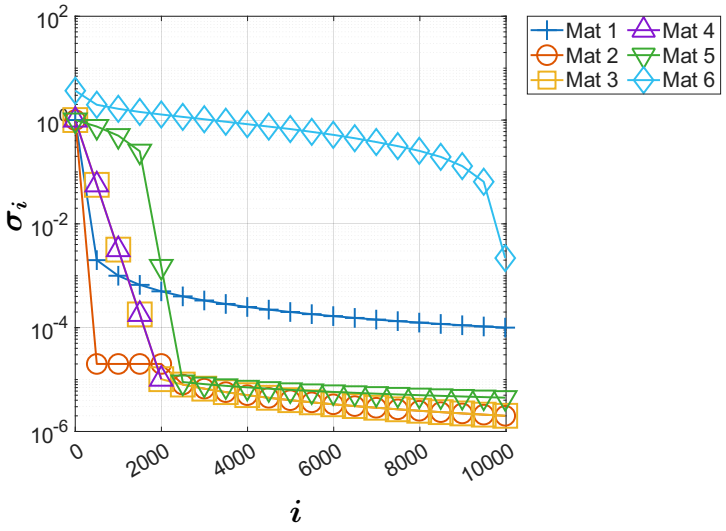


Figure 4.1: Spectra of the six matrices from a synthetic set described in [84, Section 7]. Every 500$^{\text{th}}$ singular value is marked. Note that the fourth matrix in the set is the only low-rank matrix, with just 2000 nonzero singular values.

Algorithm 8 : Adaptive Block Randomized Incremental Krylov method

Required input: Matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, Krylov block size b, maximum number of Krylov iterations $p > 1$.
Optional input: termination criterion.
Output: Orthonormal matrices $\mathbf{U}$ and $\mathbf{V}$ and a diagonal matrix $\mathbf{\Sigma}$ such that $\hat{\mathbf{M}} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}} \approx \mathbf{M}$.

function ABRIK($\mathbf{M}, b, p$)
  Sample a n-by-b sketching operator $\mathbf{\Omega}$ from a Gaussian distribution
  Let $\mathbf{R}$ =[ ]; $\mathbf{S}$ = [ ]
  // Empty matrices decloration
  for $i = 0{:}p$ **do**
    // Perform iterations of Krylov method
    if $i = 0$ **then**
      $[\mathbf{X}_0, \sim] = \texttt{qr_econ}(\mathbf{M}\Omega)$
      // At $i = 0$, $\mathbf{S}$ remains empty
    else if $i = 1$ **then**
      $[\mathbf{Y}_1, \mathbf{R}] = \texttt{qr_econ}(\mathbf{M}^{\mathrm{T}}\mathbf{X}_0)$
      // At $i = 1$, $\mathbf{R}$ is computed
    else if i is even **then**
      $[\mathbf{X}_i, \mathbf{S}] = \texttt{qr_add}(\begin{bmatrix}\mathbf{X}_0, & \mathbf{X}_2, & \cdots, & \mathbf{X}_{i-2}\end{bmatrix}, \mathbf{M}\mathbf{Y}_{i-1}, \mathbf{S})$
      // Done via Algorithm 9; input $\mathbf{S}$ is empty or upper-Hessenberg
    else
      $[\mathbf{Y}_i, \mathbf{R}] = \texttt{qr_add}(\begin{bmatrix}\mathbf{Y}_1, & \mathbf{Y}_3, & \cdots, & \mathbf{Y}_{i-2}\end{bmatrix}, \mathbf{M}^{\mathrm{T}}\mathbf{X}_{i-1}, \mathbf{R})$
      // Done via Algorithm 9; input $\mathbf{R}$ is upper-triangular
    if termination criterion is met **then**
      break
  if i is even **then**
    $[\hat{\mathbf{U}}, \mathbf{\Sigma}, \hat{\mathbf{V}}] = \texttt{svd}(\mathbf{S})$
    // $\mathbf{S}$ is upper-Hessenberg, of size $(i/2+1)b \times (i/2)b$
    $\mathbf{U} = \begin{bmatrix}\mathbf{X}_0, & \mathbf{X}_2, & \cdots, & \mathbf{X}_i\end{bmatrix} \hat{\mathbf{U}}$
    $\mathbf{V} = \begin{bmatrix}\mathbf{Y}_1, & \mathbf{Y}_3, & \cdots, & \mathbf{Y}_{i-1}\end{bmatrix} \hat{\mathbf{V}}$
  else
    $[\hat{\mathbf{U}}, \mathbf{\Sigma}, \hat{\mathbf{V}}] = \texttt{svd}(\mathbf{R}^{\mathrm{T}})$
    // $\mathbf{R}$ is upper-triangular, of size $\lceil i/2 \rceil b \times \lceil i/2 \rceil b$
    $\mathbf{U} = \begin{bmatrix}\mathbf{X}_0, & \mathbf{X}_2, & \cdots, & \mathbf{X}_{i-1}\end{bmatrix} \hat{\mathbf{U}}$
    $\mathbf{V} = \begin{bmatrix}\mathbf{Y}_1, & \mathbf{Y}_3, & \cdots, & \mathbf{Y}_i\end{bmatrix} \hat{\mathbf{V}}$
  return $\mathbf{U}$, $\mathbf{V}$, $\mathbf{\Sigma}$

representations of the Krylov basis vectors, i.e., $\mathbf{S}$ and $\mathbf{R}$ represent the compression of $\mathbf{M}$ into the given subspaces. Pseudocode Algorithm 9 presents `qr_add`, which is used to orthogonalize the current block of the Krylov subspace and compute the new portion of matrices $\mathbf{S}$ or $\mathbf{R}$.

Remark 29. In Algorithm 8, $\mathbf{S}$ is upper-Hessenberg, and $\mathbf{R}$ is upper-triangular. This structural asymmetry comes from the particular design decision in Algorithm 8: during the Krylov iteration $i = 0$, I avoid computing any part of the matrix $\mathbf{S}$. If I were to force the symmetry by performing such a computation, then iteration $i = 2$ would have resulted in an insufficient number of blocks in the right Krylov subspace (only $\mathbf{Y}_1$ would have been computed at that time), while the shape of $\mathbf{S}$ would have implied needing two blocks in the right subspace. Similarly, forcing both $\mathbf{S}$ and $\mathbf{R}$ to be upper-triangular or upper-Hessenberg results in complications with the number of Krylov blocks available at each given iteration.

An optional termination criterion for Algorithm 8 was described as part of [120, Algorithm A.1, steps 16-17, 25-26] and involved performing a direct SVD on factors $\mathbf{S}$ and $\mathbf{R}$ at every iteration of the algorithm's inner loop. While this strategy can be advantageous in specific application scenarios, my overall assessment is that its performance overhead outweighs the potential benefits.

The `qr_econ` function, used repeatedly in both Algorithm 8 and Algorithm 9, performs a tall-and-skinny QR factorization. In an implementation context, a pivot-free variant of the approach described in Chapter 2 may offer improved efficiency. In an implementation of `ABRIK`, the `svd` function in Algorithm 8 is represented via LAPACK's fastest direct SVD method, `GESDD`.

4.1.1 Alternative methods under evaluation

This section briefly presents algorithms that I compare `ABRIK` against.

SVDS. This method is represented by a low-rank SVD from the Spectra package [107], specifically, the implementation found in [85]. Spectra leverages Eigen [67] for its core linear algebra operations, and thus supports both dense and sparse input matrices. The algorithms in Spectra are inspired by those in ARPACK [83]; like ARPACK, Spectra implements iterative eigensolvers. The `SVDS` routine internally relies on [86], which implements the Implicitly Restarted Lanczos Method. The

functionality of both Spectra and Eigen is directly embedded into RandLAPACK via CMake's `FetchContent` mechanism solely for benchmarking purposes.

`SVDS` exposes two main tuning parameters: nev, the number of eigenvalues requested, and `ncv`, which sets the size of the Krylov subspace to compute and controls the convergence speed of the underlying eigensolver. The parameters must satisfy $nev < ncv \leq n$, with the recommended setting $ncv \geq 2 \cdot nev$; in my benchmarking setup, I follow this recommendation. Since `SVDS` is an iterative method, choosing nev larger than the target rank k is expected to generally improve the quality of the top k singular triplets compared to setting $nev = k$.

RSVD. This method is implemented in RandLAPACK as a blocked randomized low-rank SVD, found at [22]. This particular implementation is based on an augmented version of the QB decomposition technique [126, Algorithm 2], and operates either with a user-specified approximation error tolerance or a fixed target rank, depending on the input configuration. The algorithm requires a user-defined block size to construct the **Q** and **B** factors that form the final low-rank approximation. The implementation of RSVD allows users to control the number of internal *subspace iterations* to refine the output approximation quality. The number of subspace iterations is controlled through the parameter p, and the widely-accepted default choice for this parameter is $p = 2$ [69].

The current implementation of RSVD in RandLAPACK does not support sparse input matrices. As such, the discussion of this algorithm is excluded from Section 4.2.2.

When using a fixed small target rank k, suppose i iterations of the QB procedure are sufficient to recover k singular triplets. In this case, performing more than i iterations will not improve the quality of those k triplets. This expected behavior contrasts with methods like `ABRIK` and `SVDS`, where additional iterations or a larger search space can enhance approximation quality.

Remark 30. Descriptions of the `RSVD` method and the underlying computational routines (power iteration and QB2) are provided in [101, Sections 4.2.1, 4.3.1, 4.3.2]. The material in [101, Section 4] was initially drafted by me and later redacted by coauthors.

4.1.2 Common tuning parameters and metrics

Efficiency metric. In Section 4.2.1 and Section 4.2.2, I use wall clock time as a direct measure of algorithms' efficiency.

Factorization accuracy metric. The metric that I use assesses the accuracy of the k leading singular triplets $(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{v}}_i, \hat{\sigma}_i)$, $i \in 1, ..., k$ of the input $\mathbf{M}$. The residual error of the leading singular triplets is computed as follows:

$$r = (\|\mathbf{A}\hat{\mathbf{V}} - \hat{\mathbf{\Sigma}}\hat{\mathbf{U}}\|^2 + \|\mathbf{A}^{\mathrm{T}}\mathbf{U} - \hat{\mathbf{V}}\hat{\mathbf{\Sigma}}\|^2)^{1/2}, \tag{4.1}$$

where $\hat{\mathbf{U}}, \hat{\mathbf{V}}, \hat{\mathbf{\Sigma}}$ are comprised of the entries in the leading singular triplets. The discussion of this metric can be found in [120, Section 6.2].

Tuning parameters considerations. All three algorithms in Section 4.1.1 aim to compute k singular triplets. However, these algorithms differ significantly in the configurable tuning parameters they expose to the user. Both `ABRIK` and `RSVD` allow for control over the block size, while `SVDS` does not expose such a parameter. Additionally, `RSVD` lacks an explicit notion of Krylov iterations, and the number of iterations performed by `SVDS` is determined internally and not user-configurable.

To enable meaningful and readable visual comparisons in Section 4.2.1 and Section 4.2.2, I adopt a unified presentation strategy across all three algorithms. In particular, I ensure that the plots reflect how parameter choices impact both performance and factorization quality, two metrics that are not directly controlled by the user.

For each matrix tested, I present a figure containing six (Section 4.2.1) or four (Section 4.2.2) subplots, arranged in three (Section 4.2.1) or two (Section 4.2.2) columns (each column relating to a single algorithm) and two rows. The x-axis of the top row shows the total number of singular triplets computed, while the x-axis of the bottom row reports the best achieved wall clock time in seconds. The y-axis in each subplot reports the number of accurate digits of the singular triplets' approximation accuracy achieved according to the selected factorization quality metric. The reported accuracy corresponds to the experiment that achieved the best wall clock time across five consecutive runs. This layout highlights the trade-off between accuracy and speed as the number of computed singular triplets increases.

Legend entries correspond to different block sizes. Since `SVDS` lacks an explicit block size parameter, it is assigned its own distinct legend entry.

Note that only the data points where a given algorithm achieved between one and sixteen digits of accuracy are shown in the plots; all other outcomes resulting from arbitrary tuning parameter choices are excluded.

4.2 Performance results

The experiments in this section compare the performance and factorization quality tradeoffs of the three algorithms for low-rank SVD, as described in Section 4.1.

Remark 31. As mentioned earlier in Section 4.1, using a pivoting-free version of `CQRRPT` (Chapter 2) in place of `qr_econ` calls in Algorithm 8 can be beneficial. For that reason, I implemented a `CQRRT` driver, available at [21]. In short, this function is beneficial to use in `ABRIK` when inputting larger matrices, of dimension over 10,000. As such, I am using it in the benchmarks on sparse inputs, described in Section 4.2.2.

4.2.1 Experiments on dense data

Recall that the set of dense matrices experimented on in this section is described in Section 4.0.2. As stated in [120], `RSVD` is expected to perform poorly on inputs with slowly-decaying spectra; in the case with the chosen dataset, a slowly-decaying spectrum is seen in the matrices five and six. The hypothetical best-case scenario for `RSVD` would be matrix one with a rapidly-decaying spectrum.

In the benchmarks, I set the block size parameter b to $\{4, 8, 16, 32, 64, 128\}$, and the maximum number of Krylov iterations parameter p to $\{8, 16, 32\}$. Consequently, the total number of singular triplets found would be $\#triplets = \{16, 32, 64, 128, 256, 512, 1024, 2048\}$. In all experiments in this section, I will be assessing the quality of approximating the first ten singular triplets of the given matrix.

See Fig. 4.2 for results on a matrix with a rapidly decaying spectrum, and Fig. 4.3 for results on a matrix with a slowly decaying spectrum. Additional experiments on the remaining matrices from the selected set can be found in Appendix A.1.

Figures 4.2 and 4.3 show that in these particular settings, `ABRIK` exhibits the best approximation quality to performance tradeoff. Notably, increasing the number of

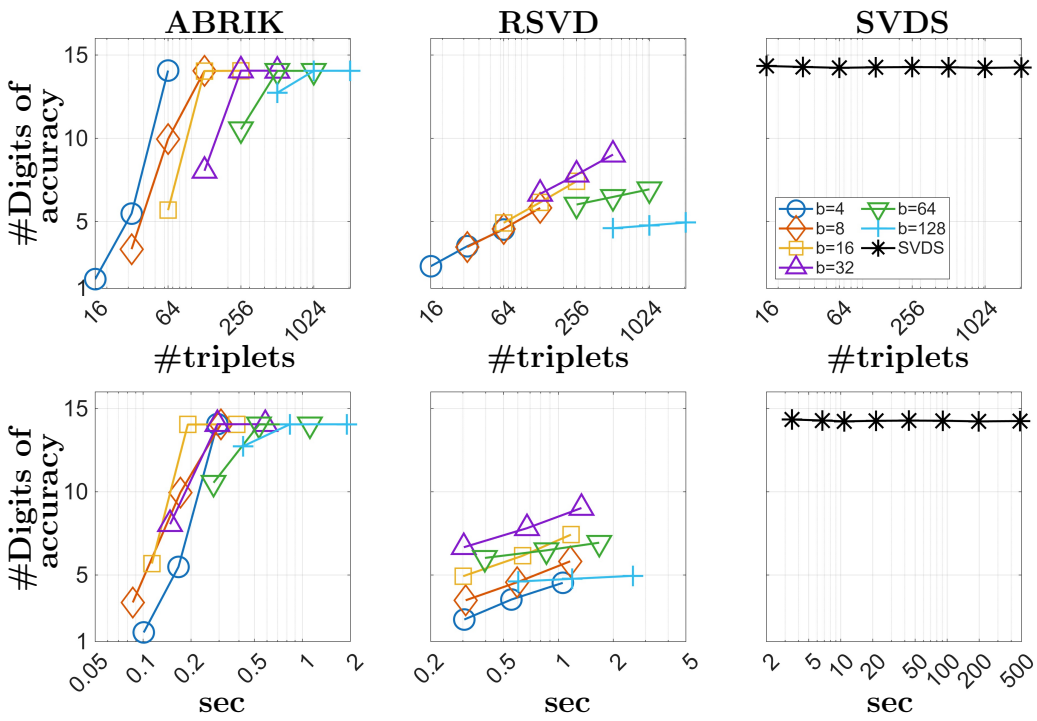


Figure 4.2: Speed (in seconds) and accuracy of `ABRIK`, `RSVD`, and `SVDS`, evaluated across various algorithm tuning parameters (see Section 4.1.2) using the input matrix with rapid spectral decay (the first test matrix from the dense synthetic dataset; see Section 4.0.2). Note that the x-axes of the subplots in the bottom row have different values. `ABRIK` attains full near-15-digit accuracy for several block sizes and achieves the best performance-to-accuracy trade-off when $b = 16$ is used to compute the leading 128 singular triplets. `RSVD` performs best with $b = 32$, though it falls short of reaching full 15-digit accuracy. `SVDS` achieves full accuracy but lags behind in performance.

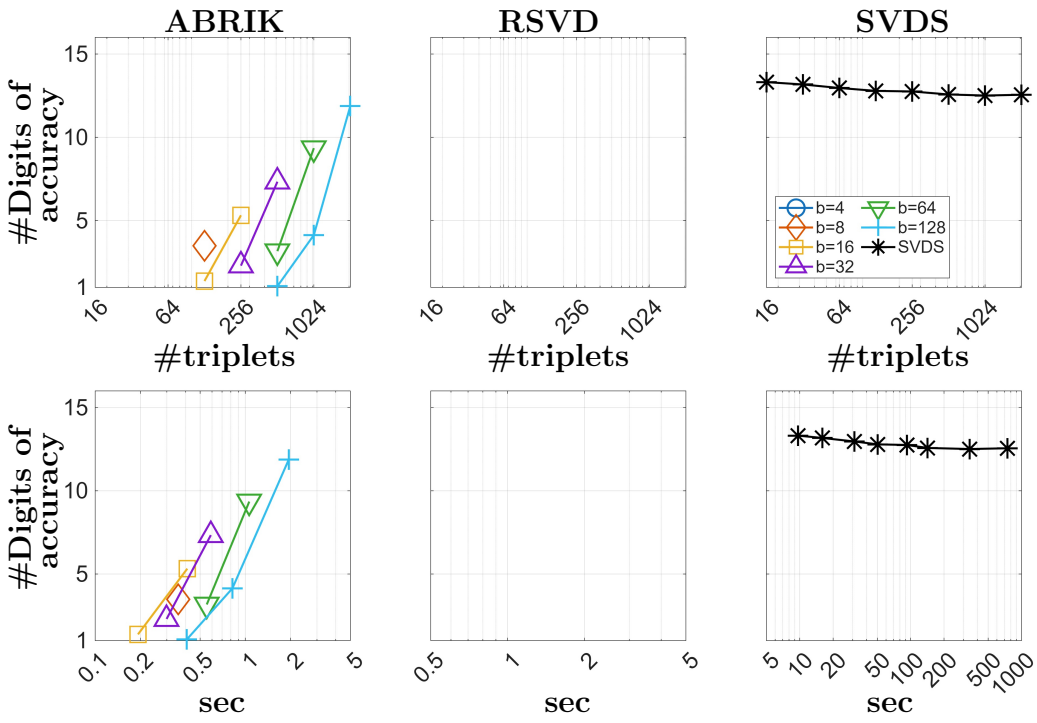


Figure 4.3: Performance and accuracy of `ABRIK`, `RSVD`, and `SVDS`, evaluated across various algorithm tuning parameters (see Section 4.1.2) using the input matrix with slow spectral decay (the sixth test matrix from the dense synthetic dataset; see Section 4.0.2). Note that the x-axes of the subplots in the bottom row have different values. `ABRIK` achieves 12 digits of accuracy with $b = 128$ and 2048 singular triplets, but depicts suboptimal performance in that case. `RSVD` fails to achieve a single digit of accuracy for any combination of input parameters. `SVDS` reaches 13-digit accuracy but shows very poor performance.

target singular triplets does not improve the accuracy of the leading ten singular triplets computed by `SVDS`, likely because the best attainable accuracy is already achieved with the minimal set of parameters.

Performance profiling of major computational kernels. I now examine how the internal components of `ABRIK` contribute to overall performance, with the goal of identifying bottlenecks for potential optimization. Fig. 4.4 presents runtime breakdowns for `ABRIK` runs with $b = 16$, the configuration that offered the most balanced performance-to-accuracy trade-off in Fig. 4.2 and Fig. 4.3, across varying numbers of target singular triplets. The figure shows the percentage of total runtime attributed to each subroutine. The corresponding component kernels and (where applicable) their line references in Algorithm 8 and Algorithm 9 are outlined below:

- Data allocation - largely amounts to the `realloc` function calls for iteratively growing the sizes of the bases for the Krylov subspaces.
- SVD and factors reconstruction - represented by Algorithm 8 Lines 16-18 and 20-22.
- `ORGQR` - retrieving the explicit representation of the $\mathbf{Q}$-factor returned by the `qr_econ`, which is necessary for constructing the bases for the Krylov subspaces. This takes place right after Algorithm 8 Lines 6, 8, and Algorithm 9 Line 5.
- Reorthogonalization-represented by the dual modified Gram-Schmidt process on Algorithm 9 Lines 3 and 4.
- QR - the direct calls to `qr_econ`, seen on Algorithm 8 Lines 6, 8, and Algorithm 9 Line 5.
- `GEMM`($\mathbf{M}$) - "large" multiplications with the input matrix, seen on Algorithm 8 Lines 6, 8, 10, 12. Note that Lines 6, 8 are only triggered in the first two iterations of the main loop in Algorithm 8. As such, the advantage of potentially using a sparse operator on Line 6 would diminish as the number of Krylov iterations used grows.

Algorithm 9 : Add to QR decomposition

Input: Iteration $i \in 2{:}p$ (from Algorithm 8); if $i = 2$, $\mathbf{R}$ is empty; if i is otherwise even, $\ell = m$ and $\mathbf{R} \in \mathbb{R}^{(i/2)b \times (i/2-1)b}$ is upper-Hessenberg; if i is odd, $\ell = n$ and $\mathbf{R} \in \mathbb{R}^{\lfloor i/2 \rfloor b \times \lfloor i/2 \rfloor b}$ is upper-triangular; orthonormal $\mathbf{Q} \in \mathbb{R}^{\ell \times \lfloor i/2 \rfloor b}$, and a general matrix $\mathbf{Q}_i \in \mathbb{R}^{\ell \times b}$.

1: **function** QR_ADD($\mathbf{Q}, \mathbf{Q}_i, \mathbf{R}$)
2: $\quad \mathbf{R}_1 = \mathbf{Q}^{\mathrm{T}}\mathbf{Q}_i$
3: $\quad \mathbf{Q}_i = \mathbf{Q}_i - \mathbf{Q}\mathbf{R}_1$
4: $\quad \mathbf{Q}_i = \mathbf{Q}_i - \mathbf{Q}\mathbf{Q}^{\mathrm{T}}\mathbf{Q}_i$
$\quad$ // Orthogonalize w.r.t. the previous iterates twice
5: $\quad [\mathbf{Q}_i, \mathbf{R}_2] = \texttt{qr_econ}(\mathbf{Q}_i)$
6: $\quad \mathbf{R} = \begin{bmatrix} \mathbf{R} & \mathbf{R}_1 \\ 0 & \mathbf{R}_2 \end{bmatrix}$
$\quad$ // If input $\mathbf{R}$ is empty, vertically append $\mathbf{R}_1$ and $\mathbf{R}_2$, constructing an Upper-Hessenberg matrix
7: $\quad$ **return** $\mathbf{Q}_i$, $\mathbf{R}$

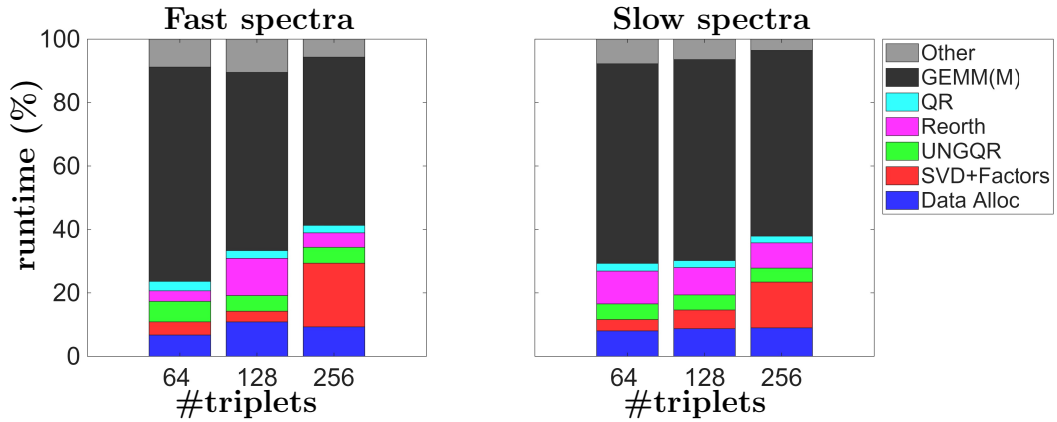


Figure 4.4: Breakdown of ABRIK runtime by subroutine when executed with $b = 16$ and targeting the leading $64, 128, 256$ singular triplets. The left subplot corresponds to matrix one from the dense set, which has a fast-decaying spectrum, while the right subplot shows results for matrix six, characterized by a slowly decaying spectrum [84, Section 7]. In both cases, the majority of the runtime is spent on large matrix multiplications with the input matrix. As expected, the cost of computing the SVD and reconstructing the factors increases with the number of target singular triplets.

4.2.2 Experiments on sparse data

Recall that the set of sparse matrices experimented on in this section is described in Section 4.0.2. As stated previously, these matrices can be considered difficult for the algorithms in question, due to poor singular value separation, nontrivial nullspace, and high condition numbers.

Remark 32. An additional comment regarding the matrices in the selected sparse set is in order. The first matrix in this set is of dimension 11,083, which can be considered rather small. Furthermore, starting with the third matrix, running `SVDS` becomes prohibitively slow, requiring several hours for a single execution. Consequently, the efficiency of `ABRIK` in this context is underscored by the drastic inefficiency of `SVDS` on these larger and more challenging inputs. As such, I chose to present the results for matrix two (of dimension 226,451), and for its leading (top-left) submatrix of dimension 113,225.

Figures 4.5 and 4.6 show the experimental results with `ABRIK` and `SVDS` on the leading submatrix and on the full matrix number two from the selected set.

Remark 33. Taking into account the figures presented in this section, along with the results from Section 4.2.1 and Appendix A.1, it remains unclear whether a universal rule can be established for setting the parameters in `ABRIK`. Further investigation, potentially focused on specific problem classes, appears necessary to develop such a guideline.

Performance profiling of major computational kernels. Similarly to Section 4.2.1, I examine the performance of the internal components of `ABRIK`, now attained during the runs on sparse input data. Fig. 4.7 presents runtime breakdowns for `ABRIK` runs with $b = 16$ from Fig. 4.5, and $b = 32$ from Fig. 4.6, representing the configurations that offered the most balanced performance-to-accuracy trade-off across varying numbers of target singular triplets. The figure shows the percentage of total runtime attributed to each subroutine.

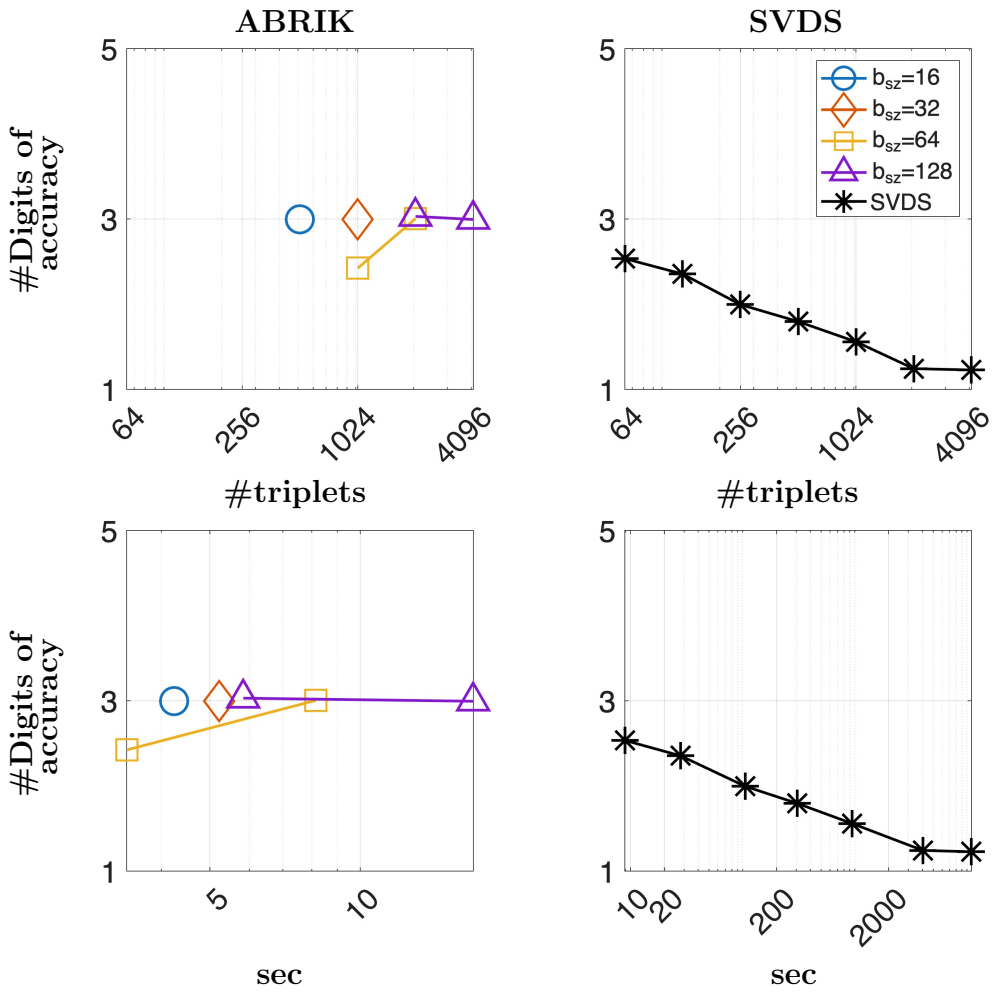


Figure 4.5: Speed (in seconds) and accuracy of `ABRIK` and `SVDS` evaluated across various algorithm tuning parameters (see Section 4.1.2) using the leading submatrix of the second test matrix from the sparse model reduction dataset (see Section 4.0.2). In this experiment, `ABRIK` attains the maximum achievable accuracy of three digits of accuracy, regardless of the input parameters configuration. `SVDS` fails to achieve even three digits of accuracy, furthermore showing stagnation in approximation accuracy with the increase of the Krylov subspace size (I anticipated the possibility of such behavior, as explained in Section 4.0.2)

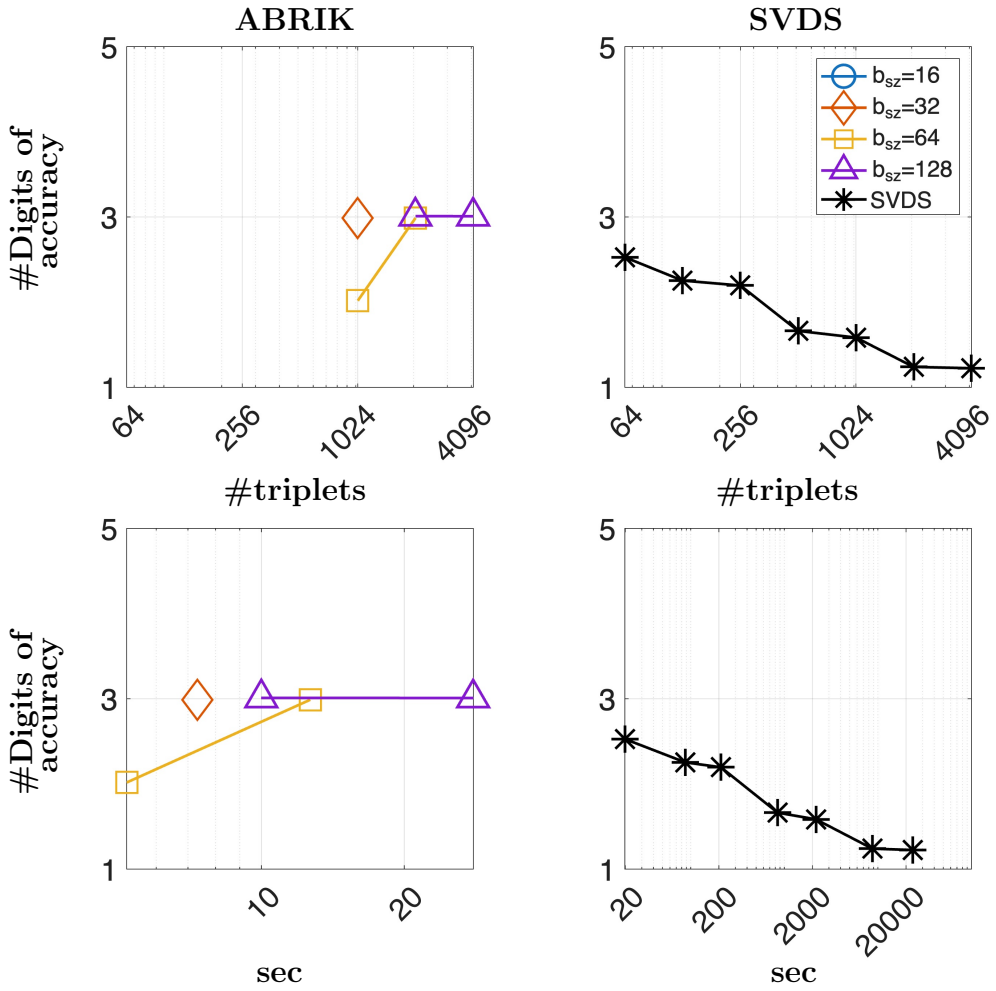


Figure 4.6: Speed (in seconds) and accuracy of `ABRIK` and `SVDS` evaluated across various algorithm tuning parameters (see Section 4.1.2) using the second test matrix from the sparse model reduction dataset (see Section 4.0.2). Observed results are similar to Fig. 4.5, with the exception of `ABRIK` now failing to achieve even a single digit of accuracy with $b = 16$ and `SVDS` exhibiting considerably worse performance.

4.3 Chapter conclusion

In this chapter, I introduced `ABRIK`, a new software for partial Singular Value Decomposition. `ABRIK` is a practical, high-performance extension of the algorithms discussed in [120], which I assisted in developing.

The implementation of `ABRIK` in RandLAPACK marked the long-anticipated introduction of abstract linear operator support, an essential abstraction for enabling consistent treatment of diverse matrix input types across the library's methods. As demonstrated by experiments on an Intel Sapphire Rapids CPU, the resulting implementation is highly efficient on the dense synthetic dataset from [84], outperforming both RandLAPACK's `RSVD` method and Spectra's `SVDS` implementation. Moreover, `ABRIK` delivers significantly better performance than `SVDS` on real-world sparse model reduction problems, which are known to be particularly challenging for Krylov-based methods.

Nonetheless, the work on this project has a few potential extensions. First, a broader comparison between `ABRIK` and other existing partial SVD methods, particularly on carefully selected real-world datasets, would greatly enhance the assessment of `ABRIK`'s practical effectiveness. Second, the current implementation of `ABRIK` could benefit from the addition of a user-controlled adaptive stopping criterion, as well as from the development of a specialized GPU-enabled version. Finally, as previously mentioned, RandLAPACK's current `RSVD` implementation lacks support for sparse inputs. This limitation is not unique to `RSVD`; it stems from the fact that abstract linear operator support, as mentioned earlier, was only recently introduced into this library. I plan on addressing this issue as part of the upcoming overhaul of RandLAPACK.

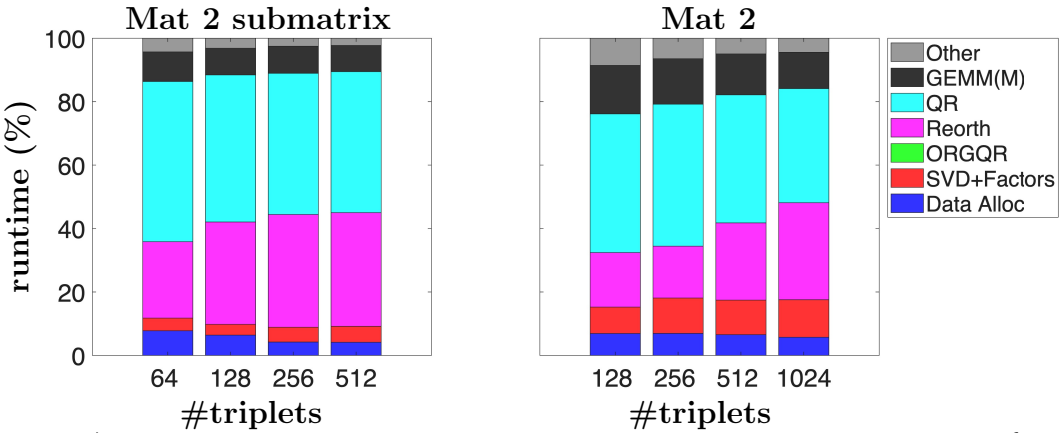


Figure 4.7: Breakdown of `ABRIK` runtime by subroutine when executed with $b = \{16, 32\}$ and targeting the leading $\{64, 128, 256, 512, 1024\}$ singular triplets. Note that in both subplots, only the runs represented by the rightmost column resulted in `ABRIK` achieving over one digit of accuracy. `ORGQR` runtime is not presented in the plot, as this step is not performed as part of `CQRRT`, which outputs the **Q**-factor explicitly. Observe that, by contrast to the dense performance breakdown Fig. 4.4, most of the runtime here is spent on QR factorization and reorthogonalization of the Krylov bases. This is the case even though `CQRRT`, the faster method for performing QR factorization, was used, as stated earlier.

Chapter 5

Epilogue

Sections 2.6, 3.10, and 4.3 provided chapter-specific conclusions, re-emphasizing the most prominent features of each of the methods I propose. This chapter serves a distinct purpose. Readers seeking concise highlights of previous chapters should refer to the earlier section.

Writing this dissertation, I did not intend to include a generalized conclusion, assuming that the quality and rigor of the main chapters would suffice to convey all the right messages. However, this assumption of mine was yet another example of how authors tend to enclose themselves in their own perspective and project their reasoning onto others, presuming that the audience's interpretation of their work would match their own. The fact is that despite the extensive body of high-quality RandNLA research, much of which my collaborators and I highlighted in [101], RandNLA methods have yet to become part of mainstream NLA software. This exemplifies the common pattern that we see in just about any domain of our lives: opportunities remaining unrecognized, or even deliberately ignored, unless someone demonstrates their value by seizing them, leaving others questioning themselves: *"Why them and not me?"* The inclusion of a generalized conclusion, here framed as the "Epilogue," is a modest attempt to break this pattern.

The core purpose of this dissertation has been to convince the reader that RandNLA can, in fact, be regarded as a serious and viable approach within NLA. This idea is, of course, not unique to my work; in fact, the opposite is true. However, *explicit* statements in this regard are rare, which, in my opinion, is a crucial oversight. With that in mind, I decided to write this chapter.

5.1 The persuasion

The core claims behind my persuasion are the following:

1. RandNLA methods can be made faster than the classical NLA methods.
2. RandNLA methods can be as accurate as the classical NLA methods.

This dissertation supports the above claims by presenting arguments that I myself found convincing, following the path of my research through the years. As such, each chapter of this dissertation conveys important messages in support of the stated claims:

- The purpose of the `CQRRPT` project, discussed in Chapter 2 was to show that randomization in RandNLA algorithms is a well-understood tool. This tool, in many cases, is used only once in a given method - upon generating a random *sketching operator*. The sketching operator is applied to the input data to reduce the dimension of the problem that we're processing. One can rigorously analyze the algorithm to show which user-controlled properties a sketching operator should have for the randomized component to have no negative effect on the algorithm's behavior with high probability, as established in Theorem 2.4. Once that is ensured, the method can be studied deterministically, under the assumption that the randomized step behaves as intended. As such, the analysis carried out in Section 2.2 (elaborated on more explicitly in [93, Section 3]) shows that the presented method has near-deterministic guarantees, i.e., that the quality of the output factors of a randomized method matches that of the classical deterministic method, which is further supported by the empirical results in Sections 2.3 and 2.4.

- The purpose of the `BQRRP` project, discussed in Chapter 3, was to demonstrate how powerful the well-implemented RandNLA methods can be when compared to traditional approaches on both CPU and GPU architectures, even without any specialized tuning, aside from some basic decision-making process of which one of the widely available algorithms is better in which context. This is illustrated in Fig. 3.9 and Fig. 3.13.

- Part of the purpose of `ABRIK` project, discussed in Chapter 4, was to emphasize the applicability of randomized methods to the "notorious" practical problems,

as seen in the discussion of the sparse dataset in Section 4.0.2. However, most importantly, this chapter was used to show that some RandNLA methods have been present in the mainstream of NLA through the example of Krylov methods, which are undoubtedly viewed as "classic" NLA algorithms.

5.2 The future

My journey through the PhD included first getting acquainted with RandNLA, then contributing to the consolidation of the knowledge in this field in the "RandNLA book" [101], and, finally, making an effort at strengthening the position of RandNLA in the field of HPC through the work featured in my dissertation, and, most importantly, through the RandLAPACK software.

In developing the "RandNLA book," my collaborators and I brought together many experts to deliver what remains the most comprehensive volume on the subject. I now set myself a parallel goal with software to establish RandLAPACK as a tool for advancing the popularity of RandNLA in HPC. As with the book, collaboration is key to the success of such an endeavor. This idea, while seemingly obvious, only became apparent to me in the early 2020s, when much of my research was commenced under the sorry conditions of being disconnected from others in the midst of a worldwide pandemic.

The first concrete step toward the said goal is my upcoming *"Birds of a Feather"* session titled *"Randomized Numerical Linear Algebra in HPC: Toward a Sustainable, Scalable Software Ecosystem "*[96], which will take place at the SC25 conference this coming November [3]. My proposal for this session was accepted by the conference review committee in August of 2025.

There exists a myriad of questions related to my goal (far too many to list here) that I, together with the RandNLA and HPC communities, will have to address in the years ahead. Neither I nor (I suspect) my readers can tell the future. But what we can do is, together, forge it.

Bibliography

[1] Randomized Block Krylov Methods for Stronger and Faster Approximate Singular Value Decomposition, author=Cameron Musco and Christopher Musco, 2015. 7

[2] Accelerate AI Workloads with Intel AMX. Technical report, Intel Corporation, Dec. 2022. Solution Brief. 1

[3] *SC25: The International Conference for High Performance Computing, Networking, Storage, and Analysis*, St. Louis, Missouri, USA, Nov. 2025. November 16–21, 2025. 122

[4] G. Abraham. FlashPCA: A Fast Principal Component Analysis for Large-Scale Datasets, 2021. Accessed: 2025-02-24. 7

[5] AMD Inc. AMD Optimizing CPU Libraries (AOCL). https://www.amd.com/en/developer/aocl.html, 2025. Accessed: 2025-09-22. 10

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 1999. 3

[7] E. Anderson, J. Dongarra, and S. Ostrouchov. LAPACK Working Note 41 Installation Guide for LAPACK. LAPACK Working Note 37996-1301, Oct. 1994. 10, 13, 14, 22, 23, 62

[8] Apple Inc. Accelerate. https://developer.apple.com/documentation/accelerate, 2025. Accessed: 2025-09-22. 10

[9] R. Armstrong and A. Damle. Collect, Commit, Expand: Efficient CPQR-Based Column Selection for Extremely Wide Matrices. *arXiv preprint arXiv:2501.18035*, 2025. Accessed: 2025-03-21. 86

[10] H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging LAPACK's Least-Squares Solver. *SIAM Journal on Scientific Computing*, 32(3):1217–1236, Jan. 2010. 15, 25

[11] O. Balabanov. Randomized Cholesky QR factorizations, 2022. 15

[12] O. Balabanov and L. Grigori. Randomized block Gram-Schmidt process for solution of linear systems and eigenvalue problems, 2021. 15

[13] O. Balabanov and L. Grigori. Randomized Gram–Schmidt process with application to GMRES. *SIAM Journal on Scientific Computing*, 44(3):A1450–A1474, 2022. 15

[14] O. Balabanov and A. Nouy. Randomized linear algebra for model reduction. Part I: Galerkin methods and error estimation. *Advances in Computational Mathematics*, 45(5):2969–3019, 2019. 26

[15] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, N. Knight, and H. Nguyen. Reconstructing Householder vectors from Tall-Skinny QR. *Journal of Parallel and Distributed Computing*, 85:3–31, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms. 58, 72

[16] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing Communication in Numerical Linear Algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011. 1

[17] BallisticLA. BQRRP-benchmark. https://github.com/BallisticLA/RandLAPACK/releases/tag/BQRRP-benchmark, 2024. GitHub release. 49

[18] BallisticLA. CQRRPT-benchmark-update. https://github.com/BallisticLA/RandLAPACK/releases/tag/CQRRPT-benchmark-update, 2024. GitHub release. 18

[19] BallisticLA. GEMM_flop_count.cc. https://github.com/BallisticLA/RandLAPACK/blob/main/benchmark/bench_general/GEMM_flop_count.cc, 2024. Source code file, RandLAPACK benchmark. 92

[20] BallisticLA. LAPACK_flop_count.cc. https://github.com/BallisticLA/RandLAPACK/blob/main/benchmark/bench_general/LAPACK_flop_count.cc, 2024. Source code file, RandLAPACK benchmark. 92

[21] BallisticLA. rl_cqrrt.hh. https://github.com/BallisticLA/RandLAPACK/blob/02e4d995ba07de2522cea8faf074ebd9b6647e64/RandLAPACK/drivers/rl_cqrrt.hh, 2024. Source code file, RandLAPACK (commit 02e4d99). 110

[22] BallisticLA. rl_rsvd.hh. https://github.com/BallisticLA/RandLAPACK/blob/main/RandLAPACK/drivers/rl_rsvd.hh, 2024. Source code file, RandLAPACK drivers. 108

[23] BallisticLA. ABRIK_benchmarking. https://github.com/BallisticLA/ABRIK_benchmarking, 2025. GitHub repository. 103

[24] BallisticLA. Dissertation-benchmarking. https://github.com/BallisticLA/BQRRP_benchmarking/releases/tag/Dissertation-benchmarking, 2025. GitHub release. 49

[25] C. Bischof and G. Quintana-Ortí. Algorithm 782: Codes for Rank-Revealing QR Factorizations of Dense Matrices. *ACM Transactions on Mathematical Software*, 24:254–257, /07/ 1998. 3

[26] C. Bischof and G. Quintana-Ortí. Computing Rank-Revealing QR Factorizations of Dense Matrices. *ACM Trans. Math. Softw.*, 24:226–253, /06/ 1998. 3

[27] C. Bischof and C. van Loan. The WY Representation for Products of Householder Matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):2–13, Jan. 1987. 58

[28] Å. Björk. *Numerical methods for Least Squares Problems*. SIAM, 1996. ISBN 0-89871-360-9. 3

[29] S. Blackford et al. Naming Scheme (LAPACK User's Guide, Node 2.4). https://www.netlib.org/lapack/lug/node24.html, 1999. LAPACK Users' Guide, Netlib. 9

[30] C. Boutsidis and A. Gittens. Improved Matrix Algorithms via the Subsampled Randomized Hadamard Transform. *SIAM Journal on Matrix Analysis and Applications*, 34(3):1301–1340, Jan. 2013. 26

[31] P. Businger and G. H. Golub. Linear least squares solutions by householder transformations. *Numerische Mathematik*, 7(3):269–276, June 1965. 3

[32] M. Cohen. Nearly Tight Oblivious Subspace Embeddings by Trace Inequalities. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Dec. 2016. 26

[33] cppreference.com. Algorithms library, 2025. Accessed: 2025-09-22. 8

[34] T. A. Davis. SuiteSparse Matrix Collection: Bodendiek group. https://sparse.tamu.edu/Bodendiek, 2012. Accessed: 2025-06-24. 104

[35] J. Demmel. Nearly optimal block-Jacobi preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 44(1):408–413, Mar. 2023. 14

[36] J. Demmel, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and M. Mahoney. Prospectus for the next LAPACK and ScaLAPACK libraries: Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC). https://icl.utk.edu/ballistic/, 7 2020. 9

[37] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012. 1, 12, 14

[38] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication Avoiding Rank Revealing QR Factorization with Column Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015. 1

[39] M. Derezinski and M. W. Mahoney. Determinantal Point Processes in Randomized Numerical Linear Algebra. *Notices of the AMS*, 68(1):34–45, 2021. 30

[40] L. Developers. DGEMQRT. Accessed: 2024-06-13; Direct link: LAPACK: DGEMQRT. 62

[41] L. Developers. DORMQR. Accessed: 2024-06-13; Direct link: LAPACK: DORMQR. 62

[42] L. Developers. GEQRF. Accessed: 2024-11-01; Direct link: LAPACK: GEQRF. 58

[43] L. Developers. GEQRT. Accessed: 2024-10-29; Direct link: LAPACK: GEQRT. 58

[44] L. Developers. ILAENV. Accessed: 2024-11-01; Direct link: LAPACK: ILAENV. 58

[45] L. Developers. LATSQR. Accessed: 2024-11-01; Direct link: LAPACK: LATSQR. 58

[46] J. Dongarra, J. Gunnels, H. Bayraktar, A. Haidar, and D. Ernst. Hardware Trends Impacting Floating-Point Computations In Scientific Applications, 2024. 1

[47] J. Dongarra and F. Sullivan. Guest Editors Introduction to the top 10 algorithms. *Computing in Science & Engineering*, 2(01):22–23, jan 2000. 1

[48] P. Drineas, R. Kannan, and M. W. Mahoney. Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication. *SIAM Journal on Computing*, 36(1):132–157, 2006. 7

[49] P. Drineas and M. W. Mahoney. On the Nystrom Method for Approximating a Gram Matrix for Improved Kernel-Based Learning. *Journal of Machine Learning Research*, 6(72):2153–2175, 2005. 7

[50] P. Drineas and M. W. Mahoney. RandNLA: Randomized Numerical Linear Algebra. *Communications of the ACM*, 59:80–90, 2016. 30

[51] P. Drineas, M. W. Mahoney, and S. Muthukrishnan. Sampling Algorithms for ℓ_2 Regression and Applications. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1127–1136, 2006. 15, 25

[52] P. Drineas, M. W. Mahoney, S. Muthukrishnan, and T. Sarlós. Faster Least Squares Approximation. *Numerische Mathematik*, 117(2):219–249, 2011. 15, 25

[53] Z. Drmac. Implementation of Jacobi Rotations for Accurate Singular Value Computation in Floating Point Arithmetic. *SIAM Journal on Scientific Computing*, 18(4):1200–1222, 1997. 7

[54] J. Duersch and M. Gu. Randomized QR with Column Pivoting. *SIAM Journal on Scientific Computing*, 39(4):C263–C291, Jan. 2017. 3, 4, 43, 54, 57, 62

[55] C. Eckart and G. Young. The Approximation of One Matrix by Another of Lower Rank. *Psychometrika*, 1(3):211–218, 1936. 6

[56] Y. Fan, Y. Guo, and T. Lin. A Novel Randomized XR-Based Preconditioned CholeskyQR Algorithm, 2021. 15

[57] J. G. F. Francis. The QR Transformation A Unitary Analogue to the LR Transformation—Part 1. *The Computer Journal*, 4(3):265–271, 01 1961. 2

[58] T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto, and Y. Yanagisawa. Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices. *SIAM Journal on Scientific Computing*, 42(1):A477–A503 (cit. on pp. 2, 19, 20), 2020. 12, 14, 36

[59] T. Fukaya, Y. Nakatsukasa, and Y. Yamamoto. A Cholesky QR Type Algorithm for Computing Tall-Skinny QR Factorization with Column Pivoting. In *Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 63–75, 2024. 86

[60] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto. CholeskyQR2: A Simple and Communication-Avoiding Algorithm for Computing a Tall-Skinny QR Factorization on a Large-Scale Parallel System. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 31–38, 2014. 14

[61] M. Gates, A. YarKhan, D. Sukkari, K. Akbudak, S. Cayrols, D. Bielich, A. Abdelfattah, M. Al Farhan, and J. Dongarra. Portable and efficient dense linear algebra in the beginning of the exascale era. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 36–46. IEEE, 2022. 9

[62] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 4 edition, Feb. 2013. 6, 22

[63] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970. 7

[64] J. Gram. Ueber die Entwickelung reeller Functionen in Reihen mittelst der Methode der kleinsten Quadrate. *Journal für die reine und angewandte Mathematik*, 94:41–73, 1883. 2

[65] M. Gu and S. Eisenstat. Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, July 1996. 17

[66] M. Gu and S. C. Eisenstat. A Divide-and-Conquer Algorithm for the Symmetric Tridiagonal Eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 16(1):172–191, 1995. 7

[67] G. Guennebaud, B. Jacob, et al. Eigen v3. https://eigen.tuxfamily.org, 2010. Accessed: 2025-06-24. 107

[68] F. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997. 58

[69] N. Halko, P.-G. Martinsson, Y. Shkolnisky, and M. Tygert. An algorithm for the principal component analysis of large data sets. *SIAM Journal on Scientific computing*, 33(5):2580–2594, 2011. 103, 108

[70] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2):217–288, Jan. 2011. 7

[71] A. J. Higgins, D. B. Szyld, E. G. Boman, and I. Yamazaki. Analysis of Randomized Householder-Cholesky QR Factorization with Multisketching, 2023. arXiv:2309.05868. 15

[72] N. Higham. What Is a Rank-Revealing Factorization? https://nhigham.com/2021/05/19/what-is-a-rank-revealing-factorization/, May 2021. [Accessed 03-Apr-2023]. 30

[73] N. Higham. The Big Six Matrix Factorizations— nhigham.com. https://nhigham.com/2022/05/18/the-big-six-matrix-factorizations/comment-page-1, May 2022. [Accessed 15-Mar-2023]. 2

[74] A. S. Householder. Unitary Triangularization of a Nonsymmetric Matrix. *J. ACM*, 5(4):339–342, Oct. 1958. 2

[75] HQRRP Development Team. LAPACK-Compatible Sources for HQRRP, 2020. Accessed: 2025-05-12; Direct link: HQRRP_sources. 82

[76] Intel Corporation. Intel oneAPI Math Kernel Library (oneMKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2025. Accessed: 2025-09-22. 10, 46

[77] I. Kozachenko. ORHR_COL, 2019. Accessed: 2024-06-13; Direct link: LAPACK: ORHR_COL. 58

[78] V. Kublanovskaya. On some algorithms for the solution of the complete eigenvalue problem. *USSR Computational Mathematics and Mathematical Physics*, 1(3):637–657, 1962. 2

[79] LAPACK Developers. LAPACK: xGESVDQ – QR-Preconditioned QR SVD Driver. https://www.netlib.org/lapack/explore-html/d4/db5/group__gesvdq.html. Accessed: 2025-09-22. 7

[80] LAPACK Developers. LAPACK: xGESVDX – Spectrum-Slicing Partial SVD Driver, 2025. Accessed: 2025-09-22. 7

[81] R. M. Larsen. PROPACK: A Software Package for the Symmetric Eigenvalue Problem and SVD on Large Sparse Matrices. http://sun.stanford.edu/~rmunk/PROPACK/, 1998. Accessed: 2025-09-02. 7

[82] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998. 7

[83] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1998. 107

[84] H. Li, G. Linderman, A. Szlam, K. Stanton, Y. Kluger, and M. Tygert. Algorithm 971: An Implementation of a Randomized Algorithm for Principal Component Analysis. *ACM Transactions on Mathematical Software*, 43:1–14, 01 2017. 103, 104, 105, 114, 118

[85] Y. Li and Contributors. PartialSVDSolver.h. https://github.com/yixuan/spectra/blob/master/include/Spectra/contrib/PartialSVDSolver.h, 2024. Source code file, Spectra library, "contrib" section. 107

[86] Y. Li and Contributors. SymEigsSolver.h. https://github.com/yixuan/spectra/blob/master/include/Spectra/SymEigsSolver.h, 2024. Source code file, Spectra library. 107

[87] MAGMA Development Team. zgeqp3_gpu.cpp – GPU-accelerated QR factorization with column pivoting, 2020. Accessed: 2025-05-12; Direct link: MAGMA_sources. 86

[88] M. W. Mahoney. *Randomized algorithms for matrices and data*. Foundations and Trends in Machine Learning. NOW Publishers, Boston, 2011. 15, 30

[89] P.-G. Martinsson. Blocked rank-revealing QR factorizations: How randomized sampling can be used to avoid single-vector pivoting. *arXiv preprint arXiv:1505.08115*, 2015. 3, 43

[90] P.-G. Martinsson, G. Quintana-Ortí, N. Heavner, and R. van de Geijn. Householder QR Factorization With Randomization for Column Pivoting (HQRRP). *SIAM Journal on Scientific Computing*, 39(2):C96–C115, Jan. 2017. 3, 4, 5, 16, 39, 42, 43, 82, 90, 91, 92

[91] P.-G. Martinsson, G. Quintana-Ortí, N. Heavner, and R. van de Geijn. HQRRP: Householder QR Factorization with Randomization for Column Pivoting. https://github.com/flame/hqrrp, 2017. GitHub repository. 4

[92] M. Melnichenko. RandLAPACK: rl_rsvd.hh, 2025. Accessed: February 24, 2025. 7

[93] M. Melnichenko, O. Balabanov, R. Murray, J. Demmel, M. W. Mahoney, and P. Luszczek. CholeskyQR with Randomization and Pivoting for Tall Matrices (CQRRPT). *SIAM Journal on Matrix Analysis and Applications*, 46(3):1701–1734, 2025. v, 16, 21, 26, 28, 33, 43, 94, 99, 121

[94] M. Melnichenko, R. Murray, W. Killian, J. Demmel, M. W. Mahoney, P. Luszczek, and M. Gates. Anatomy of High-Performance Column-Pivoted QR Decomposition. https://arxiv.org/abs/2507.00976, 2025. v, 47

[95] M. Melnichenko, R. Murray, B. Loring, and P. Nobel. RandLAPACK Library. https://github.com/BallisticLA/RandLAPACK, 2 2025. 2, 9

[96] M. Melnichenko, R. Murray, V. Patel, W. D. S. Pereira, S. Li, P. Drineas, and V. Georgiou. Randomized numerical linear algebra in hpc: Toward a sustainable, scalable software ecosystem. Birds of a Feather session, SC25, Nov. 2025. 18 November 2025, 12:15pm – 1:15pm CST, SC25. 122

[97] X. Meng, M. Saunders, and M. Mahoney. LSRN: A Parallel Iterative Solver for Strongly Over- or Underdetermined Systems. *SIAM Journal on Scientific Computing*, 36(2):C95–C118, Jan. 2014. 15, 25

[98] B. Miller. KAHAN: Upper Trapezoidal Matrix for Testing Condition and Rank Estimation. https://math.nist.gov/MatrixMarket/data/MMDELI/kahan/kahan.html, 2004. Accessed: 2025-04-24. 79

[99] mmelnich. Equalizing the behavior of ABRIK and SVDS (Pull Request #106). https://github.com/BallisticLA/RandLAPACK/pull/106, 2025. GitHub pull request. 103

[100] D. Mori, Y. Yamamoto, and S.-L. Zhang. Backward error analysis of the AllReduce algorithm for Householder QR decomposition. *Japan Journal of Industrial and Applied Mathematics*, 29:111–130, 2012. 14

[101] R. Murray, J. Demmel, M. Mahoney, N. Erichson, M. Melnichenko, O. Malik, L. Grigori, P. Luszczek, M. Derezinski, M. Lopes, T. Liang, H. Luo, and J. Dongarra. Randomized Numerical Linear Algebra: A Perspective on the Field with an Eye to Software. Technical report, U California Berkeley, U Tennessee Knoxville, U Colorado Denver, Apr 2023. arXiv:2302.11474:v2. 13, 23, 24, 54, 108, 120, 122

[102] R. Murray and B. Loring. RandBLAS: Sketching for randomized numerical linear algebra. https://randblas.readthedocs.io/en/latest/, 2024. Documentation website. 10

[103] R. Murray, B. Loring, K. He, M. Melnichenko, R. Weaver, and P. Nobel. RandBLAS: A header-only C++ library for sketching in randomized numerical linear algebra. https://github.com/BallisticLA/RandBLAS, 2024. Version 1.0.1 (latest release Sep 29, 2024), accessed 2025-06-17. 9

[104] H. D. Nguyen and J. Demmel. Reproducible tall-skinny QR. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 152–159. IEEE, 2015. 23

[105] NVIDIA Corporation. cuSOLVER — NVIDIA Developer. https://developer.nvidia.com/cusolver, 2025. Accessed: 2025-09-23. 46

[106] B. N. Parlett. *The contribution of J. H. Wilkinson to numerical analysis*, page 17–30. Association for Computing Machinery, New York, NY, USA, 1990. 2

[107] Y. Qiu. Spectra: A C++ Library for Large Scale Eigenvalue Problems. https://github.com/yixuan/spectra, 2016. Accessed: 2025-06-24. 107

[108] V. Rokhlin and M. Tygert. A fast randomized algorithm for overdetermined linear least-squares regression. *Proceedings of the National Academy of Sciences*, 105(36):13212–13217, Sept. 2008. 15, 25

[109] J. Salmon, M. Moraes, D. Shaw, et al. Random123: Counter-based random number generators for C, C++ and CUDA. https://github.com/DEShawResearch/random123, 2011. GitHub repository; latest release v1.14.0 (Jan 29, 2021). 10

[110] T. Sarlos. Improved Approximation Algorithms for Large Matrices via Random Projections. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, FOCS '06, page 143–152, USA, 2006. IEEE Computer Society. 15, 25

[111] E. Schmidt. Zur Theorie der linearen und nichtlinearen Integralgleichungen. I. Teil: Entwicklung willkürlicher Funktionen nach Systemen vorgeschriebener. *Mathematische Annalen*, 63:433–476, 1907. 2

[112] A. V. D. Sluis. Condition numbers and equilibration of matrices. *Numerische Mathematik*, (14):14–23, 1969. 14

[113] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23:2165–2182, 2002. 3

[114] G. W. Stewart. Perturbation theory for the singular value decomposition. In Y. C. Eldar and A. V. Oppenheim, editors, *SVD and Signal Processing, II: Algorithms, Analysis and Applications*, pages 99–109. Elsevier, 1991. 103

[115] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli. Summarizing CPU and GPU Design Trends with Product Data, 2020. 1

[116] T. Terao, K. Ozaki, and T. Ogita. LU-Cholesky QR algorithms for thin QR decomposition. *Parallel Computing*, 92:102571, 2020. 14

[117] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. 24

[118] S. Tomov, R. Nath, P. Du, P. Luszczek, and J. Dongarra. MAGMA: Matrix Algebra on GPU and Multicore Architectures. *International Journal of High Performance Computing Applications*, 24(3):275–293, 2010. 86

[119] J. Tropp. Improved analysis of the subsampled randomized Hadamard transform. *Advances in Adaptive Data Analysis*, 03(01n02):115–126, Apr. 2011. 26

[120] J. A. Tropp and R. J. Webber. Randomized algorithms for low-rank matrix approximation: Design, analysis, and applications, 2023. 3, 7, 102, 104, 107, 109, 110, 118

[121] J. H. Wilkinson. Modern Error Analysis. *SIAM Review*, 13(4):548–568, 1971. 2

[122] J. Xiao, M. Gu, and J. Langou. Fast Parallel Randomized QR with Column Pivoting Algorithms for Reliable Low-Rank Matrix Approximations. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 233–242, 2017. 3, 43

[123] Y. Yamamoto, Y. Nakatsukasa, Y. Yanagisawa, and T. Fukaya. Roundoff error analysis of the CholeskyQR2 algorithm. *Electron. Trans. Numer. Anal*, 44(01):306–326, 2015. 14

[124] I. Yamazaki, S. Tomov, and J. Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPUs with multiple GPUs. *SIAM J. Scientific Computing*, 37:C307–C330, 2015. 14

[125] D. Yan, W. Wang, and X. Chu. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020. 1

[126] W. Yu, Y. Gu, and Y. Li. Efficient Randomized Algorithms for the Fixed-Precision Low-Rank Matrix Approximation. *SIAM Journal on Matrix Analysis and Applications*, 39(3):1339–1359, Jan. 2018. 108

[127] F. G. V. Zee, E. Chan, R. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. The libflame library for dense matrix computations. Technical report, 2009. 4

Appendix A

ABRIK additional content

A.1 Experiments on the remaining dense matrices

This section presents the experimental results on matrices two through five from the dense set described in Section 4.0.2. Observe figures Fig. A.1, Fig. A.2, Fig. A.3, Fig. A.4.

Overall, we see `ABRIK` achieving near-full 15 digits of accuracy for all test matrices with some configuration of input parameters, while the same cannot be said about `RSVD` and `SVDS`.

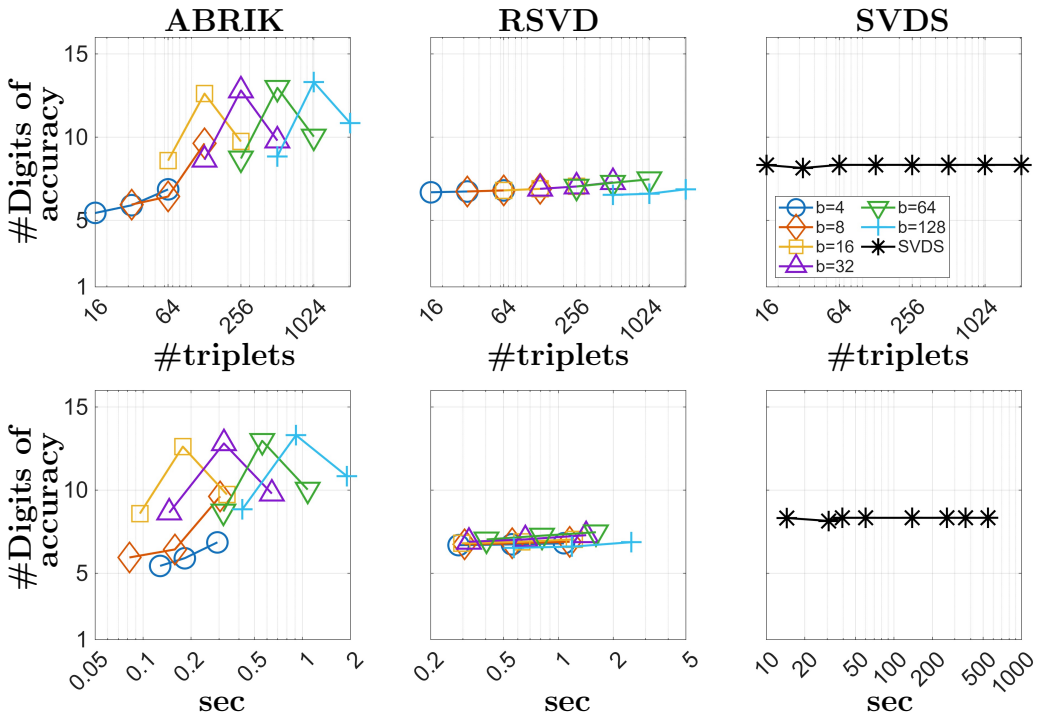


Figure A.1: Speed (in seconds) and accuracy of `ABRIK`, `RSVD`, and `SVDS` are evaluated across various algorithm tuning parameters (see Section 4.1.2) using the first test matrix from the dense synthetic dataset (see Section 4.0.2). Note that the x-axes of the subplots in the bottom row differ.

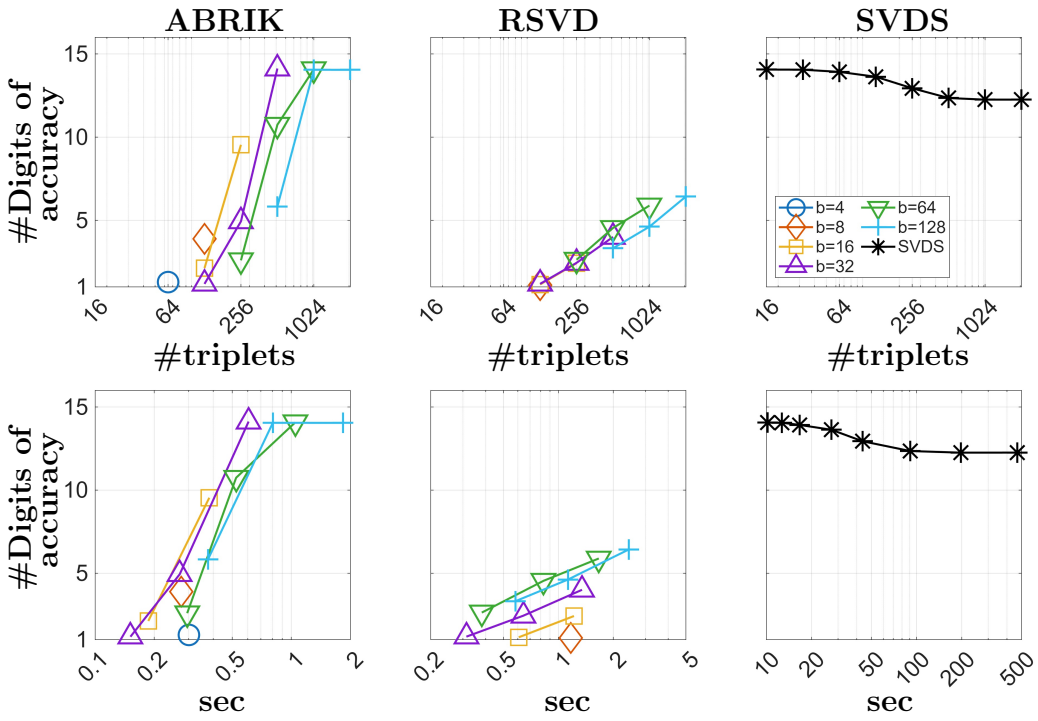


Figure A.2: Speed (in seconds) and accuracy of `ABRIK`, `RSVD`, and `SVDS` are evaluated across various algorithm tuning parameters (see Section 4.1.2) using the first test matrix from the dense synthetic dataset (see Section 4.0.2). Note that the x-axes of the subplots in the bottom row differ.

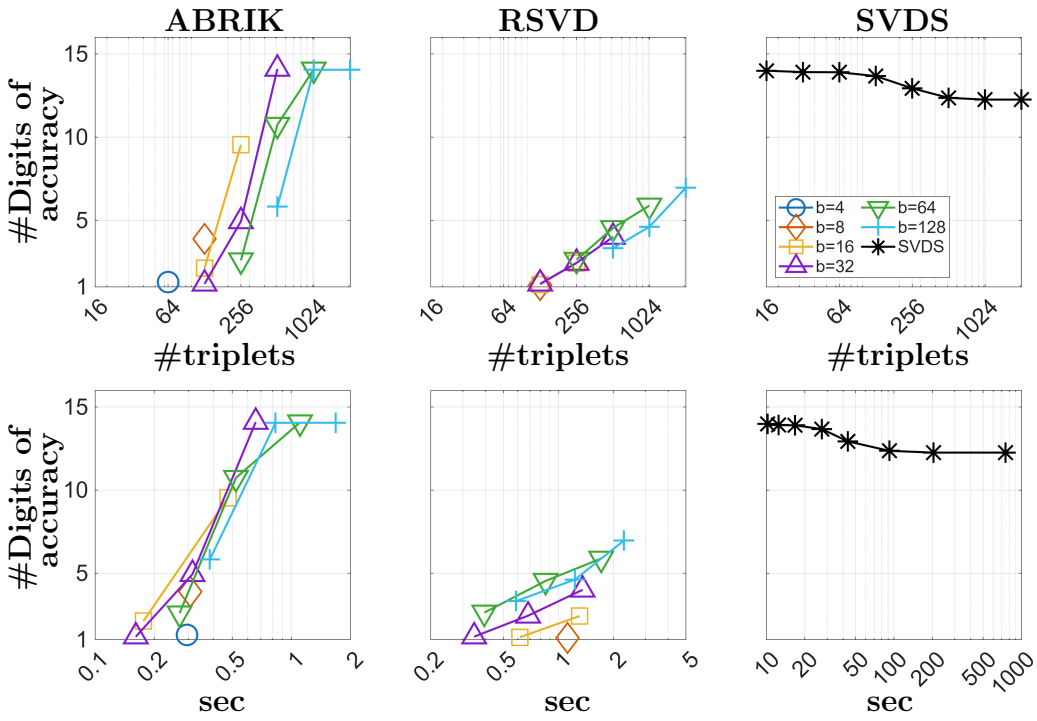


Figure A.3: Speed (in seconds) and accuracy of `ABRIK`, `RSVD`, and `SVDS` are evaluated across various algorithm tuning parameters (see Section 4.1.2) using the first test matrix from the dense synthetic dataset (see Section 4.0.2). Note that the x-axes of the subplots in the bottom row differ.

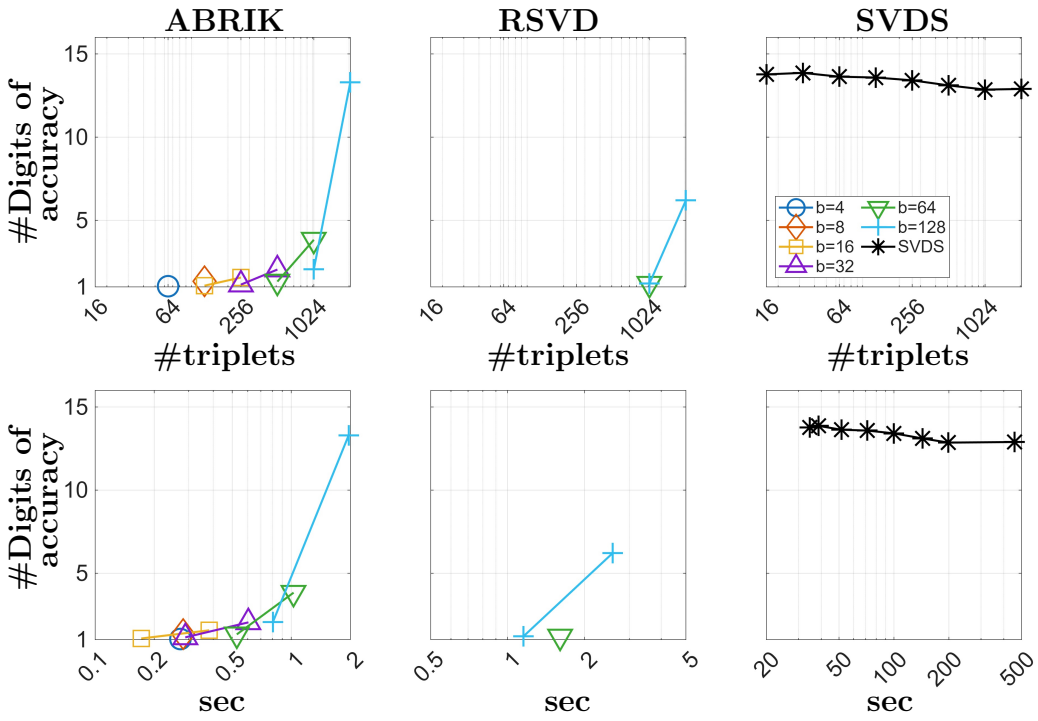


Figure A.4: Speed (in seconds) and accuracy of `ABRIK`, `RSVD`, and `SVDS` are evaluated across various algorithm tuning parameters (see Section 4.1.2) using the first test matrix from the dense synthetic dataset (see Section 4.0.2). Note that the x-axes of the subplots in the bottom row differ.

Vita

Maksim Melnichenko was born in Kaliningrad, Russia on November 8th 1998. During his undergraduate years at Millersville University of Pennsylvania, he was advised by Dr. William Killian and Dr. Baoling Ma. In May of 2020, he graduated with a dual B.S. Degree in Applied Mathematics and Computer Science. Maksim then enrolled as a doctoral student at the University of Tennessee, Knoxville, and was employed as a Graduate Research Assistant at the Innovative Computing Laboratory. There, he worked under the guidance of Dr. Jack Dongarra and Dr. Piotr Luszczek. Maksim is expected to earn his Ph.D degree, as well as his concurrent Master's degree, in December of 2025. He specializes in Randomized Numerical Linear Algebra (RandNLA) and has collaborated with UC Berkeley, LBNL, and SNL on high-performance implementations of randomized algorithms. He is the main contributor to RandLAPACK, a C++ library for implementing, testing, and benchmarking of various RandNLA algorithms.